



Xtensa[®] Instruction Set Architecture (ISA) Summary

For all Xtensa LX Processors

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Copyright © 2013, 2022 Cadence Design Systems, Inc.. All Rights Reserved
Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522. All other trademarks are the property of their respective holders.

Patents: Licensed under U.S. Patent Nos. 7,526,739; 8,032,857; 8,209,649; 8,266,560; 8,650,516

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

- * The publication may be used solely for personal, informational, and noncommercial purposes;
- * The publication may not be modified in any way;
- * Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement,
- * The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration; and
- * Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence's customer in accordance with, a written agreement between Cadence and its customer. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

For further assistance, contact Cadence Online Support at <https://support.cadence.com/>.

Product Release:RI-2021.8

Last Updated:04/2022

Modification: 737871

Cadence Design Systems, Inc.
2655 Seely Ave.
San Jose, CA 95134
www.cadence.com

Contents

List of Tables.....	xix
List of Figures.....	xxvii
Preface.....	xxix
1 Introduction.....	31
1.1 The Xtensa Instruction Set Architecture.....	32
1.1.1 Configurability.....	32
1.1.2 Extensibility.....	34
1.1.2.1 State Extensions.....	34
1.1.2.2 Register File Extensions.....	34
1.1.2.3 Instruction Extensions.....	34
1.1.2.4 Coprocessor Extensions.....	34
1.1.3 Time-to-Market.....	35
1.1.4 Code Density.....	35
1.1.5 Low Implementation Cost.....	35
1.1.6 Low-Power.....	36
1.1.7 Performance.....	36
2 Notation.....	37
2.1 Bit and Byte Order.....	38
2.2 Expressions.....	39
2.3 Unsigned Semantics.....	42
2.4 Case.....	42
2.5 Statements.....	42
2.6 Instruction Fields.....	43
3 Core Architecture.....	45
3.1 Overview of the Core Architecture.....	46
3.2 Processor-Configuration Parameters.....	46
3.3 Registers.....	47
3.3.1 General (AR) Registers.....	47
3.3.2 Shifts and the Shift Amount Register (SAR).....	47
3.3.3 Reading and Writing the Special Registers.....	49
3.4 Data Formats and Alignment.....	49
3.5 Memory.....	50
3.5.1 Memory Addressing.....	50
3.5.2 Addressing Modes.....	51
3.5.3 Program Counter.....	51
3.5.4 Instruction Fetch.....	51
3.5.4.1 Little-Endian Fetch Semantics.....	52
3.5.4.2 Big-Endian Fetch Semantics.....	53

3.6	Reset.....	54
3.7	Exceptions and Interrupts.....	54
3.8	Instruction Summary.....	55
3.8.1	Load Instructions.....	56
3.8.2	Store Instructions.....	58
3.8.3	Memory Access Ordering.....	60
3.8.4	Jump and Call Instructions.....	61
3.8.5	Conditional Branch Instructions.....	62
3.8.6	Move Instructions.....	65
3.8.7	Arithmetic Instructions.....	66
3.8.8	Bitwise Logical Instructions.....	68
3.8.9	Shift Instructions.....	68
3.8.10	Processor Control Instructions.....	70
4	Architectural Options.....	73
4.1	Option Introduction.....	74
4.1.1	Purpose of Options.....	74
4.1.2	Overview of Options.....	74
4.2	Core Architecture.....	77
4.3	Options for Additional Instructions.....	82
4.3.1	Code Density Option.....	82
4.3.1.1	Code Density Option Architectural Additions.....	82
4.3.1.2	Branches.....	84
4.3.2	Loop Option.....	84
4.3.2.1	Loop Option Architectural Additions.....	84
4.3.2.2	Restrictions on Loops.....	85
4.3.2.3	Loops Disabled During Exceptions.....	86
4.3.2.4	Loopback Semantics.....	86
4.3.3	Extended L_{32R} Option.....	86
4.3.3.1	Extended L_{32R} Option Architectural Additions.....	87
4.3.3.2	The Literal Base Register.....	87
4.3.4	16-bit Integer Multiply Option.....	87
4.3.4.1	16-bit Integer Multiply Option Architectural Additions.....	88
4.3.5	32-bit Integer Multiply Option.....	88
4.3.5.1	32-bit Integer Multiply Option Architectural Additions.....	89
4.3.6	32-bit Integer Divide Option.....	90
4.3.6.1	32-bit Integer Divide Option Architectural Additions.....	90
4.3.7	MAC16 Option.....	91
4.3.7.1	MAC16 Option Architectural Additions.....	91
4.3.7.2	Use With CLAMPS Instruction.....	94
4.3.8	Miscellaneous Operations Option.....	94
4.3.8.1	Miscellaneous Operations Option Architectural Additions.....	94
4.3.9	Deposit Bits Option.....	96
4.3.9.1	Deposit Bits Option Architectural Additions.....	96
4.3.10	Boolean Option.....	97

4.3.10.1 Boolean Option Architectural Additions.....	97
4.3.10.2 Booleans.....	98
4.3.11 Floating-Point Coprocessor Option.....	99
4.3.11.1 Floating-Point Coprocessor Option Architectural Additions.....	99
4.3.11.2 Floating-Point Representation.....	107
4.3.11.3 Floating-Point State.....	108
4.3.11.4 Floating-Point Exceptional Conditions.....	110
4.3.11.5 Divide and Square Root Sequences.....	110
4.3.12 Multiprocessor Synchronization Option.....	115
4.3.12.1 Memory Access Ordering.....	115
4.3.12.2 Multiprocessor Synchronization Option Architectural Additions.....	116
4.3.12.3 Inter-Processor Communication with the L32AI and S32RI Instructions.....	117
4.3.13 Conditional Store Option.....	118
4.3.13.1 Conditional Store Option Architectural Additions.....	119
4.3.13.2 Exclusive Access with the S32C1I Instruction.....	119
4.3.13.3 Use Models for the S32C1I Instruction.....	120
4.3.13.4 The Atomic Operation Control Register (ATOMCTL) under the Conditional Store Option.....	121
4.3.13.5 Memory Ordering and the S32C1I Instruction.....	122
4.3.14 Exclusive Access Option.....	123
4.3.14.1 Exclusive Access Option Architectural Additions.....	123
4.3.14.2 Exclusive Access with the Exclusive Instructions.....	124
4.4 Options for Interrupts and Exceptions.....	126
4.4.1 Exception Option 2.....	126
4.4.1.1 Exception Option 2 Architectural Additions.....	127
4.4.1.2 Exception Causes under the Exception Option 2.....	129
4.4.1.3 The Processor Status Register (PS) under the Exception Option 2.....	132
4.4.1.4 Value of Variables under the Exception Option 2.....	134
4.4.1.5 The Exception Cause Register (EXCCAUSE) under the Exception Option 2.....	135
4.4.1.6 The Exception Virtual Address Reg (EXCVADDR) under the Exception Option 2.....	139
4.4.1.7 The Exception Program Counter (EPC) under the Exception Option 2.....	139
4.4.1.8 The Double Exception Program Counter (DEPC) under the Exception Option 2.....	139
4.4.1.9 The Exception Save Register (EXCSAVE) under the Exception Option 2.....	140
4.4.1.10 Handling of Exceptional Conditions under the Exception Option 2...	140
4.4.1.11 Exception Priority under the Exception Option 2.....	145
4.4.2 Relocatable Vector Option.....	147
4.4.2.1 Relocatable Vector Option Architectural Additions.....	147
4.4.3 Unaligned Exception Option.....	148

4.4.3.1 Unaligned Exception Option Architectural Additions.....	149
4.4.4 Coprocessor Context Option.....	149
4.4.4.1 Coprocessor Context Option Architectural Additions.....	149
4.4.4.2 Coprocessor Context Switch.....	150
4.4.5 Interrupt Option.....	151
4.4.5.1 Interrupt Option Architectural Additions.....	151
4.4.5.2 Specifying Interrupts.....	153
4.4.5.3 The Level-1 Interrupt Process.....	156
4.4.5.4 Use of Interrupt Instructions.....	156
4.4.6 High-Priority Interrupt Option.....	157
4.4.6.1 High-Priority Interrupt Option Architectural Additions.....	157
4.4.6.2 Specifying High-Priority Interrupts.....	159
4.4.6.3 The High-Priority Interrupt Process.....	159
4.4.6.4 Checking for Interrupts.....	160
4.4.7 Timer Interrupt Option.....	161
4.4.7.1 Timer Interrupt Option Architectural Additions.....	161
4.4.7.2 Clock Counting and Comparison.....	162
4.5 Options for Local Memory.....	162
4.5.1 General Cache Option Features.....	163
4.5.1.1 Cache Terminology.....	163
4.5.1.2 Cache Tag Format.....	163
4.5.2 Instruction Cache Option.....	164
4.5.3 Data Cache Option.....	165
4.5.4 General RAM/ROM Option Features.....	165
4.5.5 Instruction RAM Option.....	166
4.5.6 Instruction ROM Option.....	166
4.5.7 Instruction Memory Access Option.....	167
4.5.8 Data RAM Option.....	167
4.5.9 Data ROM Option.....	167
4.6 Hardware Alignment Option.....	168
4.7 Memory ECC/Parity Option.....	168
4.7.1 Memory ECC/Parity Option Architectural Additions.....	169
4.7.2 Memory Error Information Registers.....	170
4.7.3 The Exception Registers.....	181
4.7.4 Memory Error Semantics.....	181
5 Options for Memory Protection and Translation.....	183
5.1 Overview of Memory Management Concepts.....	184
5.1.1 Overview of Memory Translation.....	184
5.1.2 Overview of Memory Protection.....	186
5.1.3 Overview of Attributes.....	189
5.2 The Memory Access Process.....	190
5.2.1 Choose the TLB.....	191
5.2.2 Lookup in the TLB.....	192
5.2.3 Check the Access Rights.....	193

5.2.4	Direct the Access to Local Memory.....	193
5.2.5	Direct the Access to PIF.....	196
5.2.6	Direct the Access to Cache.....	196
5.3	Region Protection Option.....	196
5.3.1	Region Protection Option Architectural Additions.....	197
5.3.2	Formats for Accessing Region Protection Option TLB Entries.....	198
5.3.3	Region Protection Option Memory Attributes.....	200
5.4	Region Translation Option.....	202
5.4.1	Region Translation Option Architectural Additions.....	203
5.4.2	Region Translation Option Formats for Accessing TLB Entries.....	203
5.4.3	Region Translation Option Memory Attributes.....	205
5.5	Memory Protection Unit Option.....	205
5.5.1	Memory Protection Unit Option Architectural Additions.....	205
5.5.2	Memory Protection Unit Option Register Formats.....	208
5.5.2.1	MPUCFG.....	208
5.5.2.2	MPUENB.....	208
5.5.2.3	ERACCESS.....	209
5.5.2.4	CACHEADDRDIS.....	209
5.5.3	The Structure of the Memory Protection Unit Option TLB.....	209
5.5.4	Formats for Writing Memory Protection Unit Option TLB Entries.....	211
5.5.5	Formats for Reading Memory Protection Unit Option TLB Entries.....	212
5.5.6	Formats for Probing Memory Protection Unit Option TLB Entries.....	213
5.5.7	Memory Protection Unit Option Access Rights Field.....	214
5.5.8	Memory Protection Unit Option Memory Type Field.....	215
5.6	MMU Option.....	217
5.6.1	MMU Option Architectural Additions.....	218
5.6.2	MMU Option Register Formats.....	222
5.6.2.1	PTEVADDR.....	222
5.6.2.2	RASID.....	223
5.6.2.3	ITLBCFG.....	223
5.6.2.4	DTLBCFG.....	223
5.6.2.5	ERACCESS.....	223
5.6.3	The Structure of the MMU Option TLBs.....	224
5.6.4	The MMU Option Memory Map.....	226
5.6.5	Formats for Writing MMU Option TLB Entries.....	227
5.6.6	Formats for Reading MMU Option TLB Entries.....	228
5.6.7	Formats for Probing MMU Option TLB Entries.....	229
5.6.8	Format for Invalidating MMU Option TLB Entries.....	230
5.6.9	MMU Option Auto-Refill TLB Ways and PTE Format.....	231
5.6.10	MMU Option Memory Attributes when <code>EXTMEMATTRIBUTES=False</code>	233
5.6.11	MMU Option Memory Type when <code>EXTMEMATTRIBUTES=True</code>	236
5.6.12	MMU Option Operation Semantics.....	238
6	Options for Other Purposes.....	239
6.1	Windowed Register Option.....	240

6.1.1	Windowed Register Option Architectural Additions.....	241
6.1.2	Managing Physical Registers.....	244
6.1.3	Window Overflow Check.....	245
6.1.4	Call, Entry, and Return Mechanism.....	247
6.1.5	Windowed Procedure-Call Protocol.....	248
6.1.6	Window Overflow and Underflow to and from the Program Stack.....	252
6.2	Miscellaneous Special Registers Option.....	254
6.2.1	Miscellaneous Special Registers Option Architectural Additions.....	254
6.3	Thread Pointer Option.....	255
6.3.1	Thread Pointer Option Architectural Additions.....	255
6.4	Processor ID Option.....	255
6.4.1	Processor ID Option Architectural Additions.....	256
6.5	Debug Option.....	256
6.5.1	Debug Option Architectural Additions.....	256
6.5.2	Debug Cause Register.....	258
6.5.3	Using Breakpoints.....	259
6.5.4	Debug Exceptions.....	261
6.5.5	Instruction Counting.....	261
6.5.6	Debug Registers.....	262
6.5.7	Debug Interrupts.....	263
6.5.8	The <code>checkIcount</code> Procedure.....	263
7	Processor State.....	265
7.1	Processor State Alphabetical List.....	266
7.2	General Registers.....	271
7.3	Program Counter.....	272
7.4	Special Registers.....	272
7.4.1	Reading and Writing Special Registers.....	277
7.4.2	LOOP Special Registers.....	278
7.4.3	MAC16 Special Registers.....	280
7.4.4	Other Unprivileged Special Registers.....	281
7.4.5	Processor Status Special Register.....	283
7.4.6	Windowed Register Option Special Registers.....	289
7.4.7	Memory Management Special Registers.....	290
7.4.8	Exception Option 2 Support Special Registers.....	294
7.4.9	Exception Option 2 State Special Registers.....	298
7.4.10	Interrupt Option Special Registers.....	303
7.4.11	Timing Special Registers.....	305
7.4.12	Breakpoint Special Registers.....	308
7.4.13	Other Privileged Special Registers.....	310
7.5	User Registers.....	315
7.5.1	Reading and Writing User Registers.....	315
7.5.2	The List of User Registers.....	316
7.6	TLB Entries.....	317
7.7	Additional Register Files.....	318

7.8 Caches and Local Memories.....	318
8 Instruction Descriptions.....	321
8.1 Instruction Word.....	322
8.2 Instruction Exception Groups.....	322
8.3 Instructions.....	324
8.3.1 ABS—Absolute Value.....	324
8.3.2 ABS.D—Absolute Value Double.....	324
8.3.3 ABS.S—Absolute Value Single.....	325
8.3.4 ADD—Add.....	326
8.3.5 ADD.N—Narrow Add.....	326
8.3.6 ADD.D—Add Double.....	327
8.3.7 ADD.S—Add Single.....	328
8.3.8 ADDEXP.D—Add Exponent Double.....	329
8.3.9 ADDEXP.S—Add Exponent Single.....	329
8.3.10 ADDEXP.M.D—Add Exponent from Mantissa Double.....	330
8.3.11 ADDEXP.M.S—Add Exponent from Mantissa Single.....	331
8.3.12 ADDI—Add Immediate.....	332
8.3.13 ADDI.N—Narrow Add Immediate.....	333
8.3.14 ADDMI—Add Immediate with Shift by 8.....	334
8.3.15 ADDX2—Add with Shift by 1.....	335
8.3.16 ADDX4—Add with Shift by 2.....	336
8.3.17 ADDX8—Add with Shift by 3.....	336
8.3.18 ALL4—All 4 Booleans True.....	337
8.3.19 ALL8—All 8 Booleans True.....	338
8.3.20 AND—Bitwise Logical And.....	339
8.3.21 ANDB—Boolean And.....	339
8.3.22 ANDBC—Boolean And with Complement.....	340
8.3.23 ANY4—Any 4 Booleans True.....	341
8.3.24 ANY8—Any 8 Booleans True.....	341
8.3.25 BALL—Branch if All Bits Set.....	342
8.3.26 BANY—Branch if Any Bit Set.....	343
8.3.27 BBC—Branch if Bit Clear.....	344
8.3.28 BBCI—Branch if Bit Clear Immediate.....	345
8.3.29 BBCI.L—Branch if Bit Clear Immediate LE.....	346
8.3.30 BBS—Branch if Bit Set.....	347
8.3.31 BBSI—Branch if Bit Set Immediate.....	348
8.3.32 BBSI.L—Branch if Bit Set Immediate LE.....	349
8.3.33 BEQ—Branch if Equal.....	349
8.3.34 BEQI—Branch if Equal Immediate.....	350
8.3.35 BEQZ—Branch if Equal to Zero.....	351
8.3.36 BEQZ.N—Narrow Branch if Equal Zero.....	352
8.3.37 BF—Branch if False.....	353
8.3.38 BGE—Branch if Greater Than or Equal.....	354
8.3.39 BGEI—Branch if Greater Than or Equal Immediate.....	355

8.3.40 BGEU—Branch if Greater Than or Equal Unsigned.....	356
8.3.41 BGEUI—Branch if Greater Than or Eq Unsigned Imm.....	357
8.3.42 BGEZ—Branch if Greater Than or Equal to Zero.....	358
8.3.43 BLT—Branch if Less Than.....	359
8.3.44 BLTI—Branch if Less Than Immediate.....	360
8.3.45 BLTU—Branch if Less Than Unsigned.....	361
8.3.46 BLTUI—Branch if Less Than Unsigned Immediate.....	361
8.3.47 BLTZ—Branch if Less Than Zero.....	362
8.3.48 BNALL—Branch if Not-All Bits Set.....	363
8.3.49 BNE—Branch if Not Equal.....	364
8.3.50 BNEI—Branch if Not Equal Immediate.....	365
8.3.51 BNEZ—Branch if Not-Equal to Zero.....	366
8.3.52 BNEZ.N—Narrow Branch if Not Equal Zero.....	367
8.3.53 BNONE—Branch if No Bit Set.....	368
8.3.54 BREAK—Breakpoint.....	369
8.3.55 BREAK.N—Narrow Breakpoint.....	370
8.3.56 BT—Branch if True.....	371
8.3.57 CALL0—Non-windowed Call.....	372
8.3.58 CALL4—Call PC-relative, Rotate Window by 4.....	373
8.3.59 CALL8—Call PC-relative, Rotate Window by 8.....	375
8.3.60 CALL12—Call PC-relative, Rotate Window by 12.....	376
8.3.61 CALLX0—Non-windowed Call Register.....	377
8.3.62 CALLX4—Call Register, Rotate Window by 4.....	378
8.3.63 CALLX8—Call Register, Rotate Window by 8.....	379
8.3.64 CALLX12—Call Register, Rotate Window by 12.....	380
8.3.65 CEIL.D—Ceiling Double to Fixed.....	382
8.3.66 CEIL.S—Ceiling Single to Fixed.....	383
8.3.67 CLAMPS—Signed Clamp.....	383
8.3.68 CLREX—Clear Exclusive.....	384
8.3.69 CONST.D—Constant Double.....	385
8.3.70 CONST.S—Constant Single.....	386
8.3.71 CONST16—Shift In 16-bit Constant.....	387
8.3.72 CVTD.S—Convert Single to Double.....	388
8.3.73 CVTS.D—Convert Double to Single.....	389
8.3.74 DCI—Data Cache Coherent Hit Invalidate.....	389
8.3.75 DCWB—Data Cache Coherent Hit Writeback.....	391
8.3.76 DCWBI—Data Cache Coherent Hit WB Invalidate.....	392
8.3.77 DEPBITS—Deposit Bits.....	394
8.3.78 DHI—Data Cache Hit Invalidate.....	395
8.3.79 DHI.B—Block Data Cache Hit Invalidate.....	396
8.3.80 DHU—Data Cache Hit Unlock.....	397
8.3.81 DHWB—Data Cache Hit Writeback.....	398
8.3.82 DHWB.B—Block Data Cache Hit Writeback.....	400
8.3.83 DHWBI—Data Cache Hit Writeback Invalidate.....	400
8.3.84 DHWBI.B—Block Data Cache Hit Writeback Inv.....	402

8.3.85 DII—Data Cache Index Invalidate.....	403
8.3.86 DIU—Data Cache Index Unlock.....	404
8.3.87 DIV0.D—Divide Begin Double.....	406
8.3.88 DIV0.S—Divide Begin Single.....	406
8.3.89 DIVN.D—Divide Final Double.....	407
8.3.90 DIVN.S—Divide Final Single.....	408
8.3.91 DIWB—Data Cache Index Write Back.....	409
8.3.92 DIWBI—Data Cache Index Write Back Invalidate.....	410
8.3.93 DIWBUI.P—Data Cache Empty.....	412
8.3.94 DPFL—Data Cache Prefetch and Lock.....	413
8.3.95 DPFM.B—Block Data Cache Prefetch and Modify.....	415
8.3.96 DPFM.BF—Block Data Cache Prefetch/Modify First.....	416
8.3.97 DPFR—Data Cache Prefetch for Read.....	416
8.3.98 DPFR.B—Block Data Cache Prefetch for Read.....	418
8.3.99 DPFR.BF—Block Data Cache Prefetch for Read First.....	419
8.3.100 DPFRO—Data Cache Prefetch for Read Once.....	419
8.3.101 DPFW—Data Cache Prefetch for Write.....	421
8.3.102 DPFW.B—Block Data Cache Prefetch for Write.....	422
8.3.103 DPFW.BF—Block Data Cache Prefetch for Write First.....	423
8.3.104 DPFWO—Data Cache Prefetch for Write Once.....	424
8.3.105 DSYNC—Load/Store Synchronize.....	425
8.3.106 ENTRY—Subroutine Entry.....	426
8.3.107 ESYNC—Execute Synchronize.....	427
8.3.108 EXCW—Exception Wait.....	428
8.3.109 EXTUI—Extract Unsigned Immediate.....	429
8.3.110 EXTW—External Wait.....	430
8.3.111 FLOAT.D—Convert Fixed to Double.....	431
8.3.112 FLOAT.S—Convert Fixed to Single.....	431
8.3.113 FLOOR.D—Floor Double to Fixed.....	432
8.3.114 FLOOR.S—Floor Single to Fixed.....	433
8.3.115 FSYNC—Fetch Synchronize.....	434
8.3.116 GETEX—Get Exclusive Result.....	434
8.3.117 IDTLB—Invalidate Data TLB Entry.....	435
8.3.118 IHI—Instruction Cache Hit Invalidate.....	436
8.3.119 IHU—Instruction Cache Hit Unlock.....	438
8.3.120 III—Instruction Cache Index Invalidate.....	439
8.3.121 IITLB—Invalidate Instruction TLB Entry.....	441
8.3.122 IIU—Instruction Cache Index Unlock.....	442
8.3.123 ILL—Illegal Instruction.....	443
8.3.124 ILL.N—Narrow Illegal Instruction.....	444
8.3.125 IPF—Instruction Cache Prefetch.....	444
8.3.126 IPFL—Instruction Cache Prefetch and Lock.....	446
8.3.127 ISYNC—Instruction Fetch Synchronize.....	447
8.3.128 J—Unconditional Jump.....	449
8.3.129 J.L—Unconditional Jump Long.....	449

8.3.130 JX—Unconditional Jump Register.....	450
8.3.131 L8UI—Load 8-bit Unsigned.....	450
8.3.132 L16SI—Load 16-bit Signed.....	451
8.3.133 L16UI—Load 16-bit Unsigned.....	453
8.3.134 L32AI—Load 32-bit Acquire.....	454
8.3.135 L32E—Load 32-bit for Window Exceptions.....	455
8.3.136 L32EX—Load 32-bit Exclusive.....	457
8.3.137 L32I—Load 32-bit.....	458
8.3.138 L32I.N—Narrow Load 32-bit.....	459
8.3.139 L32R—Load 32-bit PC-Relative.....	461
8.3.140 LDCT—Load Data Cache Tag.....	463
8.3.141 LDCW—Load Data Cache Word.....	464
8.3.142 LDDEC—Load with Autodecrement.....	465
8.3.143 LDDR32.P—Load to DDR Register.....	467
8.3.144 LDI—Load Double Immediate.....	467
8.3.145 LDINC—Load with Autoincrement.....	468
8.3.146 LDIP—Load Double Immediate Post-Increment.....	469
8.3.147 LDX—Load Double Indexed.....	471
8.3.148 LDXP—Load Double Indexed Post-Increment.....	472
8.3.149 LOOP—Loop.....	473
8.3.150 LOOPGTZ—Loop if Greater Than Zero.....	475
8.3.151 LOOPNEZ—Loop if Not-Equal Zero.....	476
8.3.152 LSI—Load Single Immediate.....	478
8.3.153 LSIP—Load Single Immediate Post-Increment.....	480
8.3.154 LSIU—Load Single Immediate Update.....	481
8.3.155 LSX—Load Single Indexed.....	482
8.3.156 LSXP—Load Single Indexed Post-Increment.....	483
8.3.157 LSXU—Load Single Indexed Update.....	484
8.3.158 MADD.D—Multiply and Add Double.....	486
8.3.159 MADD.S—Multiply and Add Single.....	486
8.3.160 MADDN.D—Multiply and Add Double Round Nearest.....	487
8.3.161 MADDN.S—Multiply and Add Single Round Nearest.....	488
8.3.162 MAX—Maximum Value.....	489
8.3.163 MAXU—Maximum Value Unsigned.....	489
8.3.164 MEMW—Memory Wait.....	490
8.3.165 MIN—Minimum Value.....	491
8.3.166 MINU—Minimum Value Unsigned.....	491
8.3.167 MKDADJ.D—Make Divide Adjust Double.....	492
8.3.168 MKDADJ.S—Make Divide Adjust Single.....	493
8.3.169 MKSADJ.D—Make Square Root Adjust Double.....	494
8.3.170 MKSADJ.S—Make Square Root Adjust Single.....	494
8.3.171 MOV—Move.....	495
8.3.172 MOV.D—Move Double.....	496
8.3.173 MOV.N—Narrow Move.....	497
8.3.174 MOV.S—Move Single.....	498

8.3.175 MOVEQZ—Move if Equal to Zero.....	499
8.3.176 MOVEQZ.D—Move Double if Equal to Zero.....	499
8.3.177 MOVEQZ.S—Move Single if Equal to Zero.....	500
8.3.178 MOVF—Move if False.....	501
8.3.179 MOVF.D—Move Double if False.....	502
8.3.180 MOVF.S—Move Single if False.....	503
8.3.181 MOVGEZ—Move if Greater Than or Equal to Zero.....	504
8.3.182 MOVGEZ.D—Move Double if Greater Than or Eq Zero.....	504
8.3.183 MOVGEZ.S—Move Single if Greater Than or Eq Zero.....	505
8.3.184 MOVI—Move Immediate.....	506
8.3.185 MOVI.N—Narrow Move Immediate.....	507
8.3.186 MOVLTZ—Move if Less Than Zero.....	508
8.3.187 MOVLTZ.D—Move Double if Less Than Zero.....	509
8.3.188 MOVLTZ.S—Move Single if Less Than Zero.....	510
8.3.189 MOVNEZ—Move if Not-Equal to Zero.....	510
8.3.190 MOVNEZ.D—Move Double if Not Equal to Zero.....	511
8.3.191 MOVNEZ.S—Move Single if Not Equal to Zero.....	512
8.3.192 MOVSP—Move to Stack Pointer.....	513
8.3.193 MOVT—Move if True.....	514
8.3.194 MOVT.D—Move Double if True.....	515
8.3.195 MOVT.S—Move Single if True.....	516
8.3.196 MSUB.D—Multiply and Subtract Double.....	517
8.3.197 MSUB.S—Multiply and Subtract Single.....	517
8.3.198 MUL.AA.*—Signed Multiply.....	518
8.3.199 MUL.AD.*—Signed Multiply.....	519
8.3.200 MUL.DA.*—Signed Multiply.....	520
8.3.201 MUL.DD.*—Signed Multiply.....	521
8.3.202 MUL.D—Multiply Double.....	522
8.3.203 MUL.S—Multiply Single.....	522
8.3.204 MUL16S—Multiply 16-bit Signed.....	523
8.3.205 MUL16U—Multiply 16-bit Unsigned.....	524
8.3.206 MULA.AA.*—Signed Multiply/Accumulate.....	524
8.3.207 MULA.AD.*—Signed Multiply/Accumulate.....	525
8.3.208 MULA.DA.*—Signed Multiply/Accumulate.....	526
8.3.209 MULA.DA*.LDDEC—Signed Mult/Accum, Ld/Autodec.....	527
8.3.210 MULA.DA*.LDINC—Signed Mult/Accum, Ld/Autoinc.....	528
8.3.211 MULA.DD.*—Signed Multiply/Accumulate.....	530
8.3.212 MULA.DD*.LDDEC—Signed Mult/Accum, Ld/Autodec.....	531
8.3.213 MULA.DD*.LDINC—Signed Mult/Accum, Ld/Autoinc.....	532
8.3.214 MULL—Multiply Low.....	534
8.3.215 MULS.AA.*—Signed Multiply/Subtract.....	535
8.3.216 MULS.AD.*—Signed Multiply/Subtract.....	535
8.3.217 MULS.DA.*—Signed Multiply/Subtract.....	536
8.3.218 MULS.DD.*—Signed Multiply/Subtract.....	537
8.3.219 MULSH—Multiply Signed High.....	538

8.3.220 MULUH—Multiply Unsigned High.....	539
8.3.221 NEG—Negate.....	540
8.3.222 NEG.D—Negate Double.....	540
8.3.223 NEG.S—Negate Single.....	541
8.3.224 NEXP01.D—Narrow Exponent Range Double.....	541
8.3.225 NEXP01.S—Narrow Exponent Range Single.....	542
8.3.226 NOP—No-Operation.....	543
8.3.227 NOP.N—Narrow No-Operation.....	544
8.3.228 NSA—Normalization Shift Amount.....	545
8.3.229 NSAU—Normalization Shift Amount Unsigned.....	546
8.3.230 OEQ.D—Compare Double Equal.....	547
8.3.231 OEQ.S—Compare Single Equal.....	547
8.3.232 OLE.D—Compare Double Ord & Less Than or Equal.....	548
8.3.233 OLE.S—Compare Single Ord & Less Than or Equal.....	549
8.3.234 OLT.D—Compare Double Ordered and Less Than.....	550
8.3.235 OLT.S—Compare Single Ordered and Less Than.....	551
8.3.236 OR—Bitwise Logical Or.....	551
8.3.237 ORB—Boolean Or.....	552
8.3.238 ORBC—Boolean Or with Complement.....	553
8.3.239 TLB—PDTLB Probe Data.....	553
8.3.240 PITLB—Probe Instruction TLB.....	554
8.3.241 PPTLB—Probe Protection TLB.....	555
8.3.242 QUOS—Quotient Signed.....	556
8.3.243 QUOU—Quotient Unsigned.....	557
8.3.244 RDTLB0—Read Data TLB Entry Virtual.....	558
8.3.245 RDTLB1—Read Data TLB Entry Translation.....	559
8.3.246 RECIP0.D—Reciprocal Begin Double.....	560
8.3.247 RECIP0.S—Reciprocal Begin Single.....	560
8.3.248 REMS—Remainder Signed.....	561
8.3.249 REMU—Remainder Unsigned.....	562
8.3.250 RER—Read External Register.....	563
8.3.251 RET—Non-Windowed Return.....	564
8.3.252 RET.N—Narrow Non-Windowed Return.....	564
8.3.253 RETW—Windowed Return.....	565
8.3.254 RETW.N—Narrow Windowed Return.....	567
8.3.255 RFDD—Return from Debug and Dispatch.....	568
8.3.256 RFDE—Return from Double Exception.....	569
8.3.257 RFDO—Return from Debug Operation.....	570
8.3.258 RFE—Return from Exception.....	570
8.3.259 RFI—Return from High-Priority Interrupt.....	571
8.3.260 RFME—Return from Memory Error.....	572
8.3.261 RFR—Move FR to AR.....	573
8.3.262 RFRD—Move FR to AR Upper.....	573
8.3.263 RFUE—Return from User-Mode Exception.....	574
8.3.264 RFWO—Return from Window Overflow.....	575

8.3.265 RFWU—Return From Window Underflow.....	576
8.3.266 RITLB0—Read Instruction TLB Entry Virtual.....	576
8.3.267 RITLB1—Read Instruction TLB Entry Translation.....	577
8.3.268 ROTW—Rotate Window.....	578
8.3.269 ROUND.D—Round Double to Fixed.....	579
8.3.270 ROUND.S—Round Single to Fixed.....	580
8.3.271 RPTLB0—Read Protection TLB Entry Address.....	580
8.3.272 RPTLB1—Read Protection TLB Entry Info.....	581
8.3.273 RSIL—Read and Set Interrupt Level.....	582
8.3.274 RSQRT0.D—Reciprocal Sqrt Begin Double.....	583
8.3.275 RSQRT0.S—Reciprocal Sqrt Begin Single.....	584
8.3.276 RSR.*—Read Special Register.....	585
8.3.277 RSYNC—Register Read Synchronize.....	586
8.3.278 RUR.*—Read User Register.....	586
8.3.279 S8I—Store 8-bit.....	587
8.3.280 S16I—Store 16-bit.....	588
8.3.281 S32C1I—Store 32-bit Compare Conditional.....	589
8.3.282 S32E—Store 32-bit for Window Exceptions.....	591
8.3.283 S32EX—Store 32-bit Exclusive.....	592
8.3.284 S32I—Store 32-bit.....	594
8.3.285 S32I.N—Narrow Store 32-bit.....	595
8.3.286 S32NB—Store 32-bit Non-Buffered.....	596
8.3.287 S32RI—Store 32-bit Release.....	597
8.3.288 SALT—Set AR if Less Than.....	599
8.3.289 SALTU—Set AR if Less Than Unsigned.....	600
8.3.290 SDDR32.P—Store from DDR Register.....	600
8.3.291 SDI—Store Double Immediate.....	601
8.3.292 SDIP—Store Double Immediate Post-Increment.....	602
8.3.293 SDX—Store Double Indexed.....	603
8.3.294 SDXP—Store Double Indexed Post-Increment.....	604
8.3.295 SEXT—Sign Extend.....	605
8.3.296 SICT—Store Instruction Cache Tag.....	606
8.3.297 SICW—Store Instruction Cache Word.....	607
8.3.298 SIMCALL—Simulator Call.....	609
8.3.299 SLL—Shift Left Logical.....	609
8.3.300 SLLI—Shift Left Logical Immediate.....	610
8.3.301 SQRT0.D—Square Root Begin Double.....	611
8.3.302 SQRT0.S—Square Root Begin Single.....	612
8.3.303 SRA—Shift Right Arithmetic.....	613
8.3.304 SRAI—Shift Right Arithmetic Immediate.....	613
8.3.305 SRC—Shift Right Combined.....	614
8.3.306 SRL—Shift Right Logical.....	615
8.3.307 SRLI—Shift Right Logical Immediate.....	616
8.3.308 SSA8B—Set Shift Amount for BE Byte Shift.....	616
8.3.309 SSA8L—Set Shift Amount for LE Byte Shift.....	617

8.3.310 SSAI—Set Shift Amount Immediate.....	618
8.3.311 SSI—Store Single Immediate.....	619
8.3.312 SSIP—Store Single Immediate Post-Increment.....	620
8.3.313 SSIU—Store Single Immediate Update.....	621
8.3.314 SSL—Set Shift Amount for Left Shift.....	622
8.3.315 SSR—Set Shift Amount for Right Shift.....	623
8.3.316 SSX—Store Single Indexed.....	624
8.3.317 SSXP—Store Single Indexed Post-Increment.....	625
8.3.318 SSXU—Store Single Indexed Update.....	626
8.3.319 SUB—Subtract.....	627
8.3.320 SUB.D—Subtract Double.....	627
8.3.321 SUB.S—Subtract Single.....	628
8.3.322 SUBX2—Subtract with Shift by 1.....	629
8.3.323 SUBX4—Subtract with Shift by 2.....	629
8.3.324 SUBX8—Subtract with Shift by 3.....	630
8.3.325 SYSCALL—System Call.....	631
8.3.326 TRUNC.D—Truncate Double to Fixed.....	632
8.3.327 TRUNC.S—Truncate Single to Fixed.....	632
8.3.328 UEQ.D—Compare Double Unordered or Equal.....	633
8.3.329 UEQ.S—Compare Single Unordered or Equal.....	634
8.3.330 UFLOAT.D—Convert Unsigned Fixed to Double.....	635
8.3.331 UFLOAT.S—Convert Unsigned Fixed to Single.....	636
8.3.332 ULE.D—Compare Double Unord or Less Than or Equal.....	636
8.3.333 ULE.S—Compare Single Unord or Less Than or Equal.....	637
8.3.334 ULT.D—Compare Double Unordered or Less Than.....	638
8.3.335 ULT.S—Compare Single Unordered or Less Than.....	639
8.3.336 UMUL.AA.*—Unsigned Multiply.....	639
8.3.337 UN.D—Compare Double Unordered.....	640
8.3.338 UN.S—Compare Single Unordered.....	641
8.3.339 UTRUNC.D—Truncate Double to Fixed Unsigned.....	642
8.3.340 UTRUNC.S—Truncate Single to Fixed Unsigned.....	642
8.3.341 WAITI—Wait for Interrupt.....	643
8.3.342 WDTLB—Write Data TLB Entry.....	644
8.3.343 WER—Write External Register.....	645
8.3.344 WFR—Move AR to FR.....	646
8.3.345 WFRD—Move AR to FR Double.....	647
8.3.346 WITLB—Write Instruction TLB Entry.....	648
8.3.347 WPTLB—Write Protection TLB Entry.....	649
8.3.348 WSR.*—Write Special Register.....	650
8.3.349 WUR.*—Write User Register.....	651
8.3.350 XOR—Bitwise Logical Exclusive Or.....	652
8.3.351 XORB—Boolean Exclusive Or.....	652
8.3.352 XSR.*—Exchange Special Register.....	653

9 Instruction Formats and Opcodes.....	655
--	-----

9.1	Formats.....	656
9.1.1	RRR.....	656
9.1.2	RRI4.....	656
9.1.3	RRI8.....	656
9.1.4	RI16.....	657
9.1.5	RSR.....	657
9.1.6	CALL.....	657
9.1.7	CALLX.....	657
9.1.8	BRI8.....	658
9.1.9	BRI12.....	658
9.1.10	RRRN.....	658
9.1.11	RI7.....	658
9.1.12	RI6.....	659
9.2	Instruction Fields.....	659
9.3	Opcode Encodings.....	660
9.3.1	Opcode Maps.....	661
9.3.2	CUST0 and CUST1 Opcode Encodings.....	680
9.3.3	Cache-Option Opcode Encodings (Implementation-Specific).....	680
10	Using the Xtensa Architecture.....	683
10.1	The Windowed Register and CALL0 ABIs.....	684
10.1.1	Windowed Register Usage and Stack Layout.....	684
10.1.2	CALL0 AR Register Usage and Stack Layout.....	686
10.1.3	Data Types and Alignment.....	687
10.1.4	Argument Passing in AR Registers.....	688
10.1.5	Return Values in AR Registers.....	689
10.1.6	Variable Arguments.....	690
10.2	Floating Point Type Arguments and Return Values.....	690
10.3	Boolean (Xtbool) Types Arguments and Return Values.....	691
10.4	State Register Conventions.....	692
10.5	Stack Frame with Wide Alignment.....	692
10.6	Stack Initialization.....	694
10.7	Other Conventions.....	695
10.7.1	Break Instruction Operands.....	695
10.7.2	System Calls.....	698
10.8	Assembly Code.....	698
10.8.1	Assembler Replacements and the Underscore Form.....	699
10.8.2	Instruction Idioms.....	699

List of Tables

Table 1: Modular Components.....	32
Table 2: Instruction-Description Expressions.....	40
Table 3: Instruction-Description Statements.....	42
Table 4: Uses Of Instruction Fields.....	43
Table 5: Core Processor-Configuration Parameters.....	46
Table 6: Core-Architecture Set.....	47
Table 7: Reading and Writing Special Registers.....	49
Table 8: Operand Formats and Alignment.....	49
Table 9: Core Instruction Summary.....	55
Table 10: Load Instructions.....	56
Table 11: Store Instructions.....	58
Table 12: Memory Order Instructions.....	61
Table 13: Jump and Call Instructions.....	62
Table 14: Conditional Branch Instructions.....	62
Table 15: Branch Immediate (b4const) Encodings.....	64
Table 16: Branch Unsigned Immediate (b4constu) Encodings.....	65
Table 17: Move Instructions.....	66
Table 18: Arithmetic Instructions.....	66
Table 19: Bitwise Logical Instructions.....	68
Table 20: Shift Instructions.....	68
Table 21: Processor Control Instructions.....	70
Table 22: Core Architecture Processor-Configurations.....	77
Table 23: Core Architecture Processor-State.....	77
Table 24: Core Architecture Instructions.....	78
Table 25: Code Density Option Instruction Additions.....	83
Table 26: Loop Option Processor-Configuration Additions.....	84
Table 27: Loop Option Processor-State Additions.....	84
Table 28: Loop Option Instruction Additions.....	85
Table 29: Extended L32R Option Processor-State Additions.....	87
Table 30: 16-bit Integer Multiply Option Instruction Additions.....	88
Table 31: 32-bit Integer Multiply Option Processor-Configuration Additions.....	89
Table 32: 32-bit Integer Multiply Instruction Additions.....	89
Table 33: 32-bit Integer Divide Option Processor-Configuration Additions.....	90
Table 34: 32-bit Integer Divide Option Exception Additions.....	90
Table 35: 32-bit Integer Divide Option Instruction Additions.....	90
Table 36: MAC16 Option Processor-State Additions.....	91
Table 37: MAC16 Option Instruction Additions.....	92
Table 38: Miscellaneous Operations Option Processor-Configuration Additions.....	94

Table 39: Miscellaneous Operations Instruction Additions.....	94
Table 40: Deposit Bits Option Instruction Additions.....	97
Table 41: Boolean Option Processor-State Additions.....	97
Table 42: Boolean Option Instruction Additions.....	97
Table 43: Floating-Point Coprocessor Option Processor-Configuration Additions.....	99
Table 44: Floating-Point Coprocessor Option Processor-State Additions.....	99
Table 45: Floating-Point Coprocessor Option Instruction Additions.....	100
Table 46: Floating-Point Coprocessor Option DoublePrecision1 Instruction Additions.....	103
Table 47: FCR fields.....	108
Table 48: FSR fields.....	109
Table 49: Multiprocessor Synchronization Option Instruction Additions.....	117
Table 50: Conditional Store Option Processor-State Additions.....	119
Table 51: Conditional Store Option Instruction Additions.....	119
Table 52: ATOMCTL Register Fields.....	121
Table 53: Exclusive Access Option Processor-State Additions.....	123
Table 54: Exclusive Access Option Constant Additions (Exception Causes).....	124
Table 55: Exclusive Access Option Instruction Additions.....	124
Table 56: Exception Option 2 Constant Additions (Exception Causes).....	127
Table 57: Exception Option 2 Processor-Configuration Additions.....	127
Table 58: Exception Option 2 Processor-State Additions.....	128
Table 59: Exception Option 2 Instruction Additions.....	128
Table 60: Instruction Exceptions under the Exception Option 2.....	129
Table 61: Interrupts under the Exception Option.....	131
Table 62: Machine Checks under the Exception Option 2.....	132
Table 63: Debug Conditions under the Exception Option 2.....	132
Table 64: PS Register Fields.....	133
Table 65: Exception Causes.....	135
Table 66: Exception and Interrupt Information Registers by Vector.....	141
Table 67: Exception and Interrupt Exception Registers by Vector.....	143
Table 68: Relocatable Vector Option Processor-Configuration Additions.....	148
Table 69: Relocatable Vector Option Processor-State Addition.....	148
Table 70: Unaligned Exception Option Constant Additions (Exception Causes).....	149
Table 71: Coprocessor Context Option Exception Additions.....	150
Table 72: Coprocessor Context Option Processor-State Additions.....	150
Table 73: Interrupt Option Constant Additions (Exception Causes).....	151
Table 74: Interrupt Option Processor-Configuration Additions.....	152
Table 75: Interrupt Option Processor-State Additions.....	152
Table 76: Interrupt Option Instruction Additions.....	153
Table 77: Interrupt Types.....	154
Table 78: High-Priority Interrupt Option Processor-Configuration Additions.....	157
Table 79: High-Priority Interrupt Option Processor-State Additions.....	158

Table 80: High-Priority Interrupt Option Instruction Additions.....	158
Table 81: Timer Interrupt Option Processor-Configuration Additions.....	161
Table 82: Timer Interrupt Option Processor-State Additions.....	162
Table 83: RAM/ROM Access Restrictions.....	166
Table 84: Memory ECC/Parity Option Processor-Configuration Additions.....	169
Table 85: Memory ECC/Parity Option Processor-State Additions.....	170
Table 86: Memory ECC/Parity Option Instruction Additions.....	170
Table 87: MESR Register Fields.....	171
Table 88: MECR Register Fields.....	179
Table 89: MEVADDR Contents.....	180
Table 90: Access Characteristics Encoded in the Attributes.....	189
Table 91: Local Memory Accesses.....	193
Table 92: Region Protection Option Exception Additions.....	197
Table 93: Region Protection Option Processor-State Additions.....	197
Table 94: Region Protection Option Instruction Additions.....	197
Table 95: Region Protection Option Attribute Field Values.....	201
Table 96: Memory Protection Unit Option Processor-Configuration Additions.....	206
Table 97: Memory Protection Unit Option Exception Additions (Exception Option 2).....	206
Table 98: Memory Protection Unit Option Processor-State Additions.....	207
Table 99: Memory Protection Unit Option Instruction Additions.....	207
Table 100: Memory Protection Unit Option Access Rights.....	214
Table 101: Memory Protection Unit Option Memory Type.....	216
Table 102: MMU Option Processor-Configuration Additions.....	218
Table 103: MMU Option Exception Additions.....	219
Table 104: MMU Option Processor-State Additions.....	220
Table 105: MMU Option Instruction Additions.....	222
Table 106: MMU Option Page Sizes and Entry Counts.....	224
Table 107: MMU Option TLB Write Formats.....	228
Table 108: MMU Option TLB Read Formats.....	229
Table 109: MMU Option TLB Invalidate Formats.....	231
Table 110: MMU Option Attribute Field Values.....	234
Table 111: MMU Option Memory Type.....	236
Table 112: Windowed Register Option Constant Additions (Exception Causes).....	241
Table 113: Windowed Register Option Processor-Configuration Additions.....	241
Table 114: Windowed Register Option Processor-State Additions and Changes.....	242
Table 115: Windowed Register Option Instruction Additions.....	242
Table 116: Windowed Register Usage.....	248
Table 117: Miscellaneous Special Registers Option Processor-Configuration Additions.....	254
Table 118: Miscellaneous Special Registers Option Processor-State Additions.....	254
Table 119: Thread Pointer Option Processor-State Additions.....	255
Table 120: Processor ID Option Processor-State Additions.....	256

Table 121: Debug Option Processor-Configuration Additions.....	257
Table 122: Debug Option Processor-State Additions.....	257
Table 123: Debug Option Instruction Additions.....	258
Table 124: DEBUGCAUSE Fields.....	259
Table 125: DBREAK Fields.....	260
Table 126: DBREAKC[i] Register Fields.....	262
Table 127: Alphabetical List of Processor State.....	266
Table 128: Numerical List of Special Registers.....	272
Table 129: LBEG - Special Register #0.....	278
Table 130: LEND - Special Register #1.....	279
Table 131: LCOUNT - Special Register #2.....	279
Table 132: ACCLO - Special Register #16.....	280
Table 133: ACCHI - Special Register #17.....	280
Table 134: M0..3 - Special Register #32-35.....	281
Table 135: SAR - Special Register #3.....	281
Table 136: BR - Special Register #4.....	282
Table 137: LITBASE - Special Register #5.....	282
Table 138: SCOMPARE1 - Special Register #12.....	283
Table 139: PS - Special Register #230.....	284
Table 140: PS.INTLEVEL - Special Register #230 (part).....	284
Table 141: PS.EXCM - Special Register #230 (part).....	285
Table 142: PS.UM - Special Register #230 (part).....	286
Table 143: PS.RING - Special Register #230 (part).....	287
Table 144: PS.OWB - Special Register #230 (part).....	287
Table 145: PS.CALLINC - Special Register #230 (part).....	288
Table 146: PS.WOE - Special Register #230 (part).....	288
Table 147: WindowBase - Special Register #72.....	289
Table 148: WindowStart - Special Register #73.....	290
Table 149: PTEVADDR - Special Register #83.....	291
Table 150: RASID - Special Register #90.....	291
Table 151: MPUENB - Special Register #90.....	292
Table 152: ITLBCFG - Special Register #91.....	292
Table 153: DTLBCFG - Special Register #92.....	293
Table 154: MPUCFG - Special Register #92.....	293
Table 155: CACHEADDRDIS - Special Register #98.....	294
Table 156: EXCCAUSE - Special Register #232.....	295
Table 157: EXCVADDR - Special Register #238.....	295
Table 158: VECBASE - Special Register #231.....	296
Table 159: MESR - Special Register #109.....	296
Table 160: MECR - Special Register #110.....	297
Table 161: MEVADDR - Special Register #111.....	297

Table 162: DEBUGCAUSE - Special Register #233.....	298
Table 163: EPC1 - Special Register #177.....	298
Table 164: EPC2..7 - Special Register #178-183.....	299
Table 165: DEPC - Special Register #192.....	299
Table 166: MEPC - Special Register #106.....	300
Table 167: EPS2..7 - Special Register #194-199.....	300
Table 168: MEPS - Special Register #107.....	301
Table 169: EXCSAVE1 - Special Register #209.....	301
Table 170: EXCSAVE2..7- Special Register #210-215.....	302
Table 171: MESAVE- Special Register #108.....	302
Table 172: INTERRUPT - Special Register #226 (read).....	303
Table 173: INTSET - Special Register #226 (write).....	303
Table 174: INTCLEAR - Special Register #227.....	304
Table 175: INTENABLE - Special Register #228.....	305
Table 176: ICOUNT - Special Register #236.....	305
Table 177: ICOUNTLEVEL - Special Register #237.....	306
Table 178: CCOUNT - Special Register #234.....	307
Table 179: CCOMPARE0..2 - Special Register #240-242.....	307
Table 180: IBREAKENABLE - Special Register #96.....	308
Table 181: IBREAKA0..1 - Special Register #128-129.....	308
Table 182: DBREAKC0..1 - Special Register #160-161.....	309
Table 183: DBREAKA0..1 - Special Register #144-145.....	310
Table 184: PRID - Special Register #235.....	310
Table 185: MMID - Special Register #89.....	311
Table 186: DDR - Special Register #104.....	311
Table 187: CPENABLE - Special Register #224.....	312
Table 188: ERACCESS - Special Register #95.....	312
Table 189: MISC0..3 - Special Register #244-247.....	313
Table 190: ATOMCTL - Special Register #99.....	313
Table 191: MEMCTL - Special Register #97.....	314
Table 192: Numerical List of User Registers.....	315
Table 193: THREADPTR - User Register #231.....	316
Table 194: FCR - User Register #232.....	317
Table 195: FSR - User Register #233.....	317
Table 196: Performance of L32R Instruction.....	462
Table 197: Uses Of Instruction Fields.....	659
Table 198: Whole Opcode Space.....	661
Table 199: QRST (from Table 7–283) Formats RRR, CALLX, and RSR (t, s, r, op2 vary)...	661
Table 200: RST0 (from Table 7–284) Formats RRR and CALLX (t, s, r vary).....	662
Table 201: ST0 (from Table 7–285) Formats RRR and CALLX (t, s vary).....	662
Table 202: SNM0 (from Table 7–286) Format CALLX (n, s vary).....	662

Table 203: JR (from Table 7–287) Format CALLX (s varies).....	662
Table 204: CALLX (from Table 7–287) Format CALLX (s varies).....	663
Table 205: SYNC (from Table 7–286) Format RRR (s varies).....	663
Table 206: SYNC0 (from Table 7–290) Format RRR.....	663
Table 207: RFEI (from Table 7–286) Format RRR (s varies).....	663
Table 208: SYSIM (from Table 7–286) Format RRR (t varies).....	664
Table 209: WTLS (from Table 7–286) Format RRR (s varies).....	664
Table 210: BLKSR (from Table 7–292) Format RRR (no bits vary).....	664
Table 211: RFET (from Table 7–292) Format RRR (no bits vary).....	665
Table 212: RFM (from Table 7–292) Format RRR (nothing varies).....	665
Table 213: ST1 (from Table 7–285) Format RRR (t, s vary).....	665
Table 214: TLB (from Table 7–285) Format RRR (t, s vary).....	666
Table 215: RT0 (from Table 7–285) Format RRR (t, r vary).....	666
Table 216: RST1 (from Table 7–284) Format RRR (t, s, r vary).....	666
Table 217: ACCER (from Table 7–301) Format RRR (t, s vary).....	667
Table 218: IMP (from Table 7–301) Format RRR (t, s vary) ().....	667
Table 219: RFDX (from Table 7–303) Format RRR (s varies).....	667
Table 220: RST2 (from Table 7–284) Format RRR (t, s, r vary).....	668
Table 221: RST3 (from Table 7–284) Formats RRR and RSR (t, s, r vary).....	668
Table 222: LSCX (from Table 7–284) Format RRR (t, s, r vary).....	668
Table 223: LSC4 (from Table 7–284) Format RRI4 (t, s, r vary).....	669
Table 224: BLKPRF (from Table 7–308) Format RRR (t, s vary).....	669
Table 225: DISPL (from Table 7–308) Format RRR (t, s vary).....	669
Table 226: DISPS (from Table 7–308) Format RRR (t, s vary).....	670
Table 227: FP0 (from Table 7–284) Format RRR (t, s, r vary).....	670
Table 228: FP1OP (from Table 7–312) Format RRR (s, r vary).....	671
Table 229: FP1 (from Table 7–284) Format RRR (t, s, r vary).....	671
Table 230: DFP0 (from Table 7–284) Format RRR (t, s, r vary).....	671
Table 231: FP2OP (from Table 7–315) Format RRR (s, r vary).....	672
Table 232: DFP1 (from Table 7–284) Format RRR (t, s, r vary).....	672
Table 233: LSAI (from Table 7–283) Formats RRI8 and RRI4 (t, s, imm8 vary).....	672
Table 234: CACHE (from Table 7–318) Formats RRI8 and RRI4 (s, imm8 vary).....	673
Table 235: DCE (from Table 7–319) Format RRI4 (s, imm4 vary).....	673
Table 236: ICE (from Table 7–319) Format RRI4 (s, imm4 vary).....	673
Table 237: LSCI (from Table 7–283) Format RRI8 (t, s, imm8 vary).....	674
Table 238: MAC16 (from Table 7–283) Format RRR (t, s, r, op1 vary).....	674
Table 239: MACID (from Table 7–323) Format RRR (t, s, r vary).....	674
Table 240: MACIA (from Table 7–323) Format RRR (t, s, r vary).....	675
Table 241: MACDD (from Table 7–323) Format RRR (t, s, r vary).....	675
Table 242: MACAD (from Table 7–323) Format RRR (t, s, r vary).....	675
Table 243: MACCD (from Table 7–323) Format RRR (t, s, r vary).....	676

Table 244: MACCA (from Table 7–323) Format RRR (t, s, r vary).....	676
Table 245: MACDA (from Table 7–323) Format RRR (t, s, r vary).....	676
Table 246: MACAA (from Table 7–323) Format RRR (t, s, r vary).....	677
Table 247: MACI (from Table 7–323) Format RRR (t, s, r vary).....	677
Table 248: MACC (from Table 7–323) Format RRR (t, s, r vary).....	677
Table 249: CALLN (from Table 7–283) Format CALL (offset varies).....	678
Table 250: SI (from Table 7–283) Formats CALL, BRI8 and BRI12(offset varies).....	678
Table 251: BZ (from Table 7–335) Format BRI12 (s, imm12 vary).....	678
Table 252: BI0 (from Table 7–335) Format BRI8 (s, r, imm8 vary).....	678
Table 253: BI1 (from Table 7–335) Formats BRI8 and BRI12 (s, r, imm8 vary).....	678
Table 254: B1 (from Table 7–338) Format BRI8 (s, imm8 vary).....	678
Table 255: B (from Table 7–283) Format RRI8 (t, s, imm8 vary).....	679
Table 256: ST2 (from Table 7–283) Formats RI7 and RI6 (s, r vary).....	679
Table 257: ST3 (from Table 7–283) Format RRRN (t, s vary).....	679
Table 258: S3 (from Table 7–342) Format RRRN (s varies).....	680
Table 259: ILH (from Table 7–342) Format RRRN (no fields vary).....	680
Table 260: Windowed AR Register Usage.....	684
Table 261: CALL0 AR Register Usage.....	687
Table 262: Data Types and Alignment.....	687
Table 263: BR Register Usage.....	691
Table 264: Breakpoint Instruction Operand Conventions.....	696
Table 265: Instruction Idioms.....	699

List of Figures

Figure 1: Big and Little Bit Numbering for BBC/BBS Instructions.....	38
Figure 2: Big and Little Endian Byte Ordering.....	39
Figure 3: Virtual Address Fields.....	50
Figure 4: LITBASE Register Format.....	87
Figure 5: PS Register Format.....	133
Figure 6: EXCCAUSE Register.....	135
Figure 7: EXCVADDR Register Format.....	139
Figure 8: EPC Register Format for Exception Option 2.....	139
Figure 9: DEPC Register Format.....	140
Figure 10: EXCSAVE Register Format.....	140
Figure 11: Instruction and Data Cache Tag Format for Xtensa.....	164
Figure 12: Instruction and Data Cache Tag Address Format for Xtensa.....	164
Figure 13: MESR Register Format.....	171
Figure 14: MECR Register Format.....	179
Figure 15: MEVADDR Register Format.....	180
Figure 16: Virtual-to-Physical Address Translation.....	185
Figure 17: A Single Process' Rings.....	188
Figure 18: Nested Rings of Multiple Processes with Some Sharing.....	188
Figure 19: Region Protection Option Addressing (as) Format for WxTLB, RxTLB1, & PxTLB.....	198
Figure 20: Region Protection Option Data (at) Format for WxTLB.....	199
Figure 21: Region Protection Option Data (at) Format for RxTLB1.....	199
Figure 22: Region Protection Option Data (at) Format for PxTLB.....	199
Figure 23: Region Translation Option Addressing (as) Format for WxTLB, RxTLB1, & PxTLB.....	203
Figure 24: Region Translation Option Data (at) Format for WxTLB.....	204
Figure 25: Region Translation Option Data (at) Format for RxTLB1.....	204
Figure 26: Region Translation Option Data (at) Format for PxTLB.....	204
Figure 27: Memory Protection Unit Option Format for MPUCFG.....	208
Figure 28: Memory Protection Unit Addressing.....	210
Figure 29: Memory Protection Unit Option Addressing (as) Format for WPTLB.....	211
Figure 30: Memory Protection Unit Option Data (at) Format for WPTLB.....	212
Figure 31: Memory Protection Unit Option Addressing (as) Format for RPTLB0 and RPTLB1.....	212
Figure 32: Memory Protection Unit Option Data (at) Format for RPTLB0.....	212
Figure 33: Memory Protection Unit Option Data (at) Format for RPTLB1.....	213
Figure 34: Memory Protection Unit Option Addressing (as) Format for PxTLB.....	213
Figure 35: Memory Protection Unit Option Data (at) Format for PPTLB.....	214

Figure 36: MMU Option PTEVADDR Register Format.....	223
Figure 37: MMU Option RASID Register Format.....	223
Figure 38: MMU Option Address Map with IVARWAY56 and DVARWAY56 Fixed.....	227
Figure 39: MMU Option Addressing (as) Format for PxTLB.....	230
Figure 40: MMU Option Data (at) Format for PITLB.....	230
Figure 41: MMU Option Data (at) Format for PDTLB.....	230
Figure 42: MMU Option Page Table Entry (PTE) Format when EXTMEMATTRIBUTES=False.....	232
Figure 43: MMU Option Page Table Entry (PTE) Format when EXTMEMATTRIBUTES=True.....	232
Figure 44: Conceptual Register Window Read.....	244
Figure 45: Faster Register Window Read.....	245
Figure 46: Fastest Register Window Read.....	245
Figure 47: Register Window Near Overflow.....	246
Figure 48: Register Window Just Before Underflow.....	248
Figure 49: Stack Frame Before alloca().....	250
Figure 50: Stack Frame After First alloca().....	250
Figure 51: Stack Frame Layout.....	251
Figure 52: DEBUGCAUSE Register.....	259
Figure 53: DBREAKC[i] Format.....	262
Figure 54: Stack Frame for the Windowed Register ABI (Variable Window).....	685
Figure 55: Stack Frame for the Windowed Register ABI (Fixed Window).....	686
Figure 56: Dynamic Alignment for 64 byte-aligned Stack Frame with Windowed ABI.....	693

Preface

This manual is written for Cadence customers who are experienced in working with microprocessors or in writing assembly code or compilers. It is NOT a specification for one particular implementation of the Architecture, but rather a reference for the ongoing Instruction Set Architecture. For a detailed specification for specific products, refer to a specific Cadence processor data book.

About this Document

This document is derived from the *Cadence[®] Xtensa[®] Instruction Set Architecture (ISA) Reference Manual*. This summary document describes the ISA available for Xtensa LX processors.

The *Xtensa[®] Instruction Set Architecture (ISA) Reference Manual* is available to licensed users of Cadence Tensilica IP. The *Xtensa[®] Instruction Set Architecture (ISA) Reference Manual* also describes many architecture options in detail.

This Xtensa ISA Summary may refer to additional architecture options, but they are not described in this document. Disregard references to the following options:

- TIE language extensions
- Xtensa NX architecture and configuration options
- Exception Option 3 (not available for Xtensa LX configurations)
- Halt Option
- Data Cache Test Option
- Data Cache Index Lock Option
- Instruction Cache Test Option
- Instruction Cache Index Lock Option
- Prefetch Option
- Block Prefetch Option
- XLMI Option
- Processor Interface Option
- APB Option
- CSR Parity Option
- Secure Mode Bit Option
- Trace Port Option

Notation

- *italic_name* indicates a program or file name, document title, or term being defined.
- \$ represents your shell prompt, in user-session examples.
- **literal_input** indicates literal command-line input.
- `variable` indicates a user parameter.
- `literal_keyword` (in text paragraphs) indicates a literal command keyword.
- `literal_output` indicates literal program output.
- `... output ...` indicates unspecified program output.
- `[optional-variable]` indicates an optional parameter.
- `[variable]` indicates a parameter within literal square-braces.
- `{variable}` indicates a parameter within literal curly-braces.
- `(variable)` indicates a parameter within literal parentheses.
- | means *OR*.
- `(var1 | var2)` indicates a required choice between one of multiple parameters.
- `[var1 | var2]` indicates an optional choice between one of multiple parameters.
- `var1 [, varn]*` indicates a list of 1 or more parameters (0 or more repetitions).
- `4'b0010` is a 4-bit value specified in binary.
- `12'o7016` is a 12-bit value specified in octal.
- `10'd4839` is a 10-bit value specified in decimal.
- `32'hff2a` or `32'HFF2A` is a 32-bit value specified in hexadecimal.

Terms

- *0x* at the beginning of a value indicates a hexadecimal value.
- *b* means bit.
- *B* means byte.
- *flush* is deprecated due to potential ambiguity (it may mean *write-back* or *discard*).
- *Mb* means megabit.
- *MB* means megabyte.
- *PC* means program counter.
- *word* means 4 bytes.

1. Introduction

Topics:

- *The Xtensa Instruction Set Architecture*

This chapter provides an overview of the Xtensa Instruction Set Architecture (ISA), and the Xtensa Processor Generator.

1.1 The Xtensa Instruction Set Architecture

The Xtensa Instruction Set Architecture (ISA) is a new post-RISC ISA targeted at embedded, communication, and consumer products. The ISA is designed to provide:

- A high degree of extensibility
- Industry-leading code density
- Optimized low-power implementation
- High performance
- Low-cost implementation

This manual describes the Xtensa ISA—both the core architecture and the architectural options. This manual does not describe the memory map, extensions in the TIE language, or peripherals that can be implemented in particular configurations of the Xtensa ISA. For information on these subjects, refer to the *Tensilica Instruction Extension (TIE) Language Reference Manual* and the *Xtensa Microprocessor Data Book*.

1.1.1 Configurability

The Xtensa ISA goes further than incorporating post-RISC features: it is modular, consisting of a core architecture and architectural options. [Modular Components](#) lists the initial set of modular components.

Table 1: Modular Components

Component	Reference
Core Architecture	Core Architecture on page 45
Core Architecture options	Core Architecture on page 77
<i>Options for Additional Instructions</i>	
Code Density Option	Code Density Option on page 82
Loop Option	Loop Option on page 84
Extended L32R Option	Extended L32R Option on page 86
16-bit Integer Multiply Option	16-bit Integer Multiply Option on page 87
32-bit Integer Multiply Option	32-bit Integer Multiply Option on page 88
MAC16 Option	MAC16 Option on page 91

Component	Reference
Miscellaneous Operations Option	Miscellaneous Operations Option on page 94
Coprocessor Context Option	Coprocessor Context Option on page 149
Boolean Option	Boolean Option on page 97
Floating-Point Coprocessor Option	Floating-Point Coprocessor Option on page 99
Multiprocessor Synchronization Option	Multiprocessor Synchronization Option on page 115
Conditional Store Option	Conditional Store Option on page 118
<i>Options for Interrupts and Exceptions</i>	
Exception Option 2	Exception Option 2 on page 126
Unaligned Exception Option	Unaligned Exception Option on page 148
Interrupt Option	Interrupt Option on page 151
High-Priority Interrupt Option	High-Priority Interrupt Option on page 157
Timer Interrupt Option	Timer Interrupt Option on page 161
<i>Options for Memory</i>	
Hardware Alignment Option	Hardware Alignment Option on page 168
Memory ECC/Parity Option	Memory ECC/Parity Option on page 168
<i>Options for Memory Protection</i>	
Region Protection Option	Region Protection Option on page 196
Region Translation Option	Region Translation Option on page 202
MMU Option	MMU Option on page 217
MPU Option	Memory Protection Unit Option on page 205
<i>Options for Other Purposes</i>	
Windowed Register Option	Windowed Register Option on page 240

Component	Reference
Miscellaneous Special Registers Option	Miscellaneous Special Registers Option on page 254
Thread Pointer Option	Thread Pointer Option on page 255
Processor ID Option	Processor ID Option on page 255
Debug Option	Debug Option on page 256

1.1.2 Extensibility

In addition to the Xtensa components shown in [Modular Components](#), designers can extend the Xtensa architecture by adding States, Register Files, and instructions that operate both on the AR Register File and on the additional states the designer has added. These instructions can be single cycle or multiple cycles, and share or re-use logic.

1.1.2.1 State Extensions

The designer can add State Registers. These State Registers can be the source or destination of various instructions and are saved and restored by the operating system.

1.1.2.2 Register File Extensions

The designer can add Register Files of widely varying size. These Register Files can be the source or destination of various instructions and are saved and restored by the operating system. The registers within them are allocated by the compiler, which can spill and re-fill them if necessary.

1.1.2.3 Instruction Extensions

The designer can define new instructions that contain simple functions consisting of combinatorial logic that takes one or two source operands from registers and produces a result to be written to a register:

```
AR[r] ← f(AR[s], AR[t])
```

Instructions can also be much more complex with register file values and State appearing as both inputs and outputs. These Instructions are described using the Tensilica Instruction Extension (TIE) language (see [System-Specific Instructions—The TIE Language](#)).

1.1.2.4 Coprocessor Extensions

Another mechanism to extend the Xtensa ISA is to use the Coprocessor Context Option. A coprocessor is defined as a combination of registers, other state, and logic that operates on

that state, including loads, stores and setting of Booleans for branch true/false operations. A particular coprocessor can be enabled or disabled to control with one bit whether or not instructions accessing that combination of registers and other state may or may not execute.

1.1.3 Time-to-Market

The Xtensa Software Development Toolkit includes automatically generated software that matches the designer's processor configuration and eliminates tool headaches. The ISA's rich set of features (for example, interrupt and debug facilities) makes the system designer's job easier. The ability to create custom instructions with the TIE language allows the designer to reach performance goals with less code-tuning or hard-to-interface-to external logic.

1.1.4 Code Density

The Xtensa core ISA is implemented as 24-bit instructions. This instruction width provides a direct 25% reduction in code size compared with 32-bit ISAs. The instructions provide access to the entire processor hardware and support special functions, such as single-instruction compare-and-branch, which reduce the number of instructions required to implement various applications. These special functions result in further code-size reductions.

The Xtensa ISA also includes a Code Density Option that further reduces code size. This option adds 16-bit instructions that are distinguished by opcode, and that can be freely intermixed with 24-bit instructions to achieve higher code density without giving up the performance of a 32-bit ISA. The 16-bit instructions add no new functionality but provide compact encoding of the most frequently used 24-bit instructions. In typical code, roughly half of all instructions can be encoded in 16 bits.

The core ISA omits the branch delay slots required by some RISC ISAs. This increases code density by eliminating NOPs the compiler uses to fill the slot after a branch when it cannot find a real instruction to put there (only 50% of the branch delay slots are filled on some RISC architectures).

The Xtensa ISA provides a Windowed Registers Option. Xtensa windowed registers reduce code size by:

- Eliminating register saves and restores at procedure entry and exit
- Reducing argument shuffling
- Allowing more local variables to live permanently in registers

1.1.5 Low Implementation Cost

The Xtensa architecture is designed to facilitate efficient implementation. It can be implemented with simple instruction pipelines and direct hardware execution without micro code. Operations that are too complex to easily implement with single instructions are synthesized into appropriate instruction sequences by the compiler. The base architecture avoids instructions that would need extra register file read or write ports. This keeps the minimal configuration low-cost and low-power.

The Xtensa ISA's improvements in code size help reduce system cost (for example, by reducing the amount of ROM, Flash, or RAM required). Making features like the number of debug registers configurable allows the system designer, instead of the processor designer, to decide the cost/benefit trade-off.

1.1.6 Low-Power

The Xtensa ISA has several energy-efficient attributes that enhance battery-operated systems.

The core ISA uses a register file with only two read ports and one write port, a configuration that requires fewer transistors and less power than architectures with more ports.

The Xtensa Windowed Registers Option saves power by reducing the number of dynamic data-memory references and increasing the opportunities for variables to reside in registers, where accesses require less power than memory accesses.

The `WAITI` (Wait for Interrupt) instruction, which is a part of the Interrupt Option, saves power by setting the current interrupt level, powering down the processor's logic, and waiting for an interrupt.

1.1.7 Performance

The Xtensa ISA achieves its extensibility, code density, and low-power advantages without sacrificing performance. The Xtensa 24-bit instructions can access 16 virtual registers with 3 register operands, and 16-bit instructions can access all 16 registers with 1 to 3 register operands. The mapping of the 16 virtual registers to the physical register file can eliminate register saves and restores at procedure entry and exit, also increasing performance.

The Xtensa ISA also enhances performance by providing:

- A complete set of compare-and-branch instructions, eliminating the need for separate comparison instructions
- `LOOP`, `LOOPNEZ`, and `LOOPGTZ` instructions that provide zero-overhead looping

These features are described in [Instruction Summary](#) on page 55 of this manual. Other features of the architecture minimize critical paths, allow better compiler scheduling, and require fewer executed instructions to implement a given program.

2. Notation

Topics:

- [Bit and Byte Order](#)
- [Expressions](#)
- [Unsigned Semantics](#)
- [Case](#)
- [Statements](#)
- [Instruction Fields](#)

This manual uses the following notation for instruction descriptions. Additional notation specific to opcode encodings is provided in [Opcode Encodings](#) on page 660.

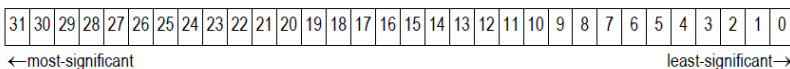
2.1 Bit and Byte Order

This manual consistently uses little-endian bit ordering for describing instructions and registers. Bits in little-endian notation are numbered starting from 0 for the least-significant bit of a field. However, this notation convention is independent of how an Xtensa processor actually numbers bits, because a given processor can be configured for either little- or big-endian byte and bit ordering. For most Xtensa instructions, bit numbering is irrelevant; only the `BBC` and `BBS` instructions assign bit numbers to values on which the processor operates. The `BBC/BBS` instructions use big-endian bit ordering (0 is the most-significant bit) on a big-endian processor configuration. Bit numbering by the `BBC/BBS` instructions is illustrated in [Big and Little Endian Byte Ordering](#).

In specifying little- or big-endian ordering during actual processor configuration, you are specifying both the bit and the byte order; the two orderings have the same most-significant and least-significant ends.

[Big and Little Endian Byte Ordering](#) illustrates big- and little-endian byte order, as implemented by Xtensa `load ()` and `store ()` instructions. Xtensa processors transfer data to and from the system using interfaces that are configurable in width (32, 64, 128, 256, or 512 bits in current implementations). These interfaces arrange their n bits according to their significance representing an n -bit unsigned integer value (that is, 0 to 2^n-1). Load and store instructions that reference quantities less than n bits access different bits of this integer in little-endian and big-endian byte orderings (for example, by changing the selection algorithm for loads). Xtensa processors *do not* rearrange bits of a word to implement endianness (for example, swapping bytes for big-endian operation).

Little-Endian bit numbering for `BBC/BBS` instructions:



Big-Endian bit numbering for `BBC/BBS` instructions:

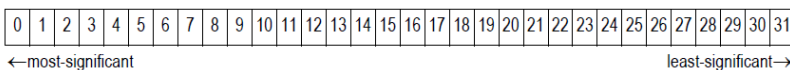


Figure 1: Big and Little Bit Numbering for BBC/BBS Instructions

Little-Endian byte addresses, 128-bit processor interface:

	127 (←most-significant)	(least-significant→) 0														
word 0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
word 1	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
word 2														...		32

Big-Endian byte addresses, 128-bit processor interface:

	127 (←most-significant)	(least-significant→) 0														
word 0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
word 1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
word 2	32	...														

Little-Endian byte addresses, 64-bit processor interface:

	63 (←most-significant)	(least-significant→) 0						
word 0	7	6	5	4	3	2	1	0
word 1	15	14	13	12	11	10	9	8
word 2							...	16

Big-Endian byte addresses, 64-bit processor interface:

	63 (←most-significant)	(least-significant→) 0						
word 0	0	1	2	3	4	5	6	7
word 1	8	9	10	11	12	13	14	15
word 2	16	...						

Little-Endian byte addresses, 32-bit processor interface:

	31	0		
word 0	3	2	1	0
word 1	7	6	5	4
word 2			...	8

Big-Endian byte addresses, 32-bit processor interface:

	31	0		
word 0	0	1	2	3
word 1	4	5	6	7
word 2	8	...		

Figure 2: Big and Little Endian Byte Ordering

2.2 Expressions

Instruction-Description Expressions defines notational forms used in expressions that describe the operation of instructions. In the table, v is an n -bit quantity, u is an m -bit quantity, and t is a 1-bit quantity.

Table 2: Instruction-Description Expressions

Expression Notation ¹	Definition
<code>v[x]</code>	Bit <i>x</i> of <i>v</i> . The result is 1 bit.
<code>v[x..y]</code>	Bits from position <i>x</i> to <i>y</i> of <i>v</i> . The result is <i>x-y+1</i> bits.
<code>v[y]</code>	The value <i>v</i> replicated <i>y</i> times. The result is <i>n×y</i> bits.
<code>array[i]</code>	Reference to element <i>i</i> of <i>array</i> .
<code>u v</code>	The catenation of bit strings <i>u</i> and <i>v</i> . The result is <i>m+n</i> bits.
<code>not v</code>	Bitwise logical complement of <i>v</i> . The result is <i>n</i> bits.
<code>u and v</code>	Bitwise logical and of <i>u</i> and <i>v</i> . <i>u</i> and <i>v</i> must be the same width. The result is <i>n</i> bits.
<code>u or v</code>	Bitwise logical or of <i>u</i> and <i>v</i> . <i>u</i> and <i>v</i> must be the same width. The result is <i>n</i> bits.
<code>u xor v</code>	Bitwise logical exclusive or of <i>u</i> and <i>v</i> . <i>u</i> and <i>v</i> must be the same width. The result is <i>n</i> bits.
<code>u = v</code>	Test for exact equality of <i>u</i> and <i>v</i> . <i>u</i> and <i>v</i> must be the same width. The result is 1 bit.
<code>u ≠ v</code>	Test for inequality of <i>u</i> and <i>v</i> . <i>u</i> and <i>v</i> must be the same width. The result is 1 bit.
<code>u < v</code>	Two's complement less-than test on <i>u</i> and <i>v</i> . <i>u</i> and <i>v</i> must be the same width. The result is 1 bit.
<code>u ≤ v</code>	Two's complement less-than or equal-to test on <i>u</i> and <i>v</i> . <i>u</i> and <i>v</i> must be the same width. The result is 1 bit.
<code>u > v</code>	Two's complement greater-than test on <i>u</i> and <i>v</i> . <i>u</i> and <i>v</i> must be the same width. The result is 1 bit.
<code>u ≥ v</code>	Two's complement greater-than or equal-to test on <i>u</i> and <i>v</i> . <i>u</i> and <i>v</i> must be the same width. The result is 1 bit.

Expression Notation ¹	Definition
$u + v$	Two's complement addition of u and v . u and v must be the same width. The result is n bits.
$u - v$	Two's complement subtraction of u and v . u and v must be the same width. The result is n bits.
$u \times v$	Low-order product of two's complement multiplication of u and v . u and v must be the same width. The result is n bits.
$u \text{ quo } v$	Quotient of two's complement division of u by v . u and v must be the same width. The result is n bits.
$u \text{ rem } v$	Remainder of two's complement division of u by v . u and v must be the same width. The result is n bits.
$\text{if } t \text{ then } u \text{ else } v$	Conditional expression. The value is u if $t = 1$. The value is v if $t = 0$.
$u +_s v$	IEEE754 single-precision floating-point addition of u and v . u and v must be 32 bits. The result is 32 bits.
$u -_s v$	IEEE754 single-precision floating-point subtraction of u and v . u and v must be 32 bits. The result is 32 bits.
$u \times_s v$	IEEE754 single-precision floating-point multiplication of u and v . u and v must be 32 bits. The result is 32 bits.
$u \div_s v$	IEEE754 single-precision floating-point division of u by v . u and v must be 32 bits. The result is 32 bits.
$\text{sqrt}_s(u)$	IEEE754 single-precision floating-point square root of u . u must be 32 bits. The result is 32 bits.
$\text{pows}(u, v)$	IEEE754 single-precision floating-point power function where u is raised to the v power. u must be 32 bits. The result is 32 bits.
<p>1. t is a 1-bit quantity, u is a m-bit quantity, v is an n-bit quantity. Constants are written either as decimal numbers, in which case the width is determined from context, or in binary.</p>	

2.3 Unsigned Semantics

In this notation, prepending a zero bit is often used for unsigned semantics. For example, the following notation indicates an unsigned less-than test:

```
(0 | u) (0 | v)
```

2.4 Case

Processor-state variables (for example, registers) are shown in UPPER CASE.

Temporary variables are shown in lower case. If a particular variable is in italics (*variable*), it is local in the sense that it has no meaning outside the local instruction flow. If it is plain (*variable*), it comes from or is used outside of the local instruction flow such as an instruction field or the next PC.

2.5 Statements

Instruction-Description Statements defines notational forms used in statements used to describe the operation of instructions.

Table 3: Instruction-Description Statements

Statement Notation	Definition
<code>v ← expr</code>	Assignment of <code>expr</code> to <code>v</code> .
<pre>if t1 then s1 [elseif t2 then s2] . . . [else sn] endif</pre>	Conditional statement. If <code>t1 = 1</code> then execute statements <code>s1</code> . Otherwise, if <code>t2 = 1</code> then execute statements <code>s2</code> , etc. Finally if none of the previous tests are true, execute statements <code>sn</code> .
<code>label:</code>	Define <code>label</code> for use as a <code>goto</code> target.
<code>goto label</code>	Transfer control to <code>label</code> .

2.6 Instruction Fields

The fields in *Uses Of Instruction Fields* are used in the descriptions of the instructions. Instruction formats and opcodes are described in *Instruction Formats and Opcodes* on page 655.

Table 4: Uses Of Instruction Fields

Field	Definition
op0	Major opcode
op1	4-bit sub-opcode for 24-bit instructions
op2	4-bit sub-opcode for 24-bit instructions
r	AR target (result), BR target (result), 4-bit immediate, 4-bit sub-opcode
s	AR source, BR source, AR target
t	AR target, BR target, AR source, BR source, 4-bit sub-opcode
n	Register window increment, 2-bit sub-opcode, n 2'b00 is used as a AR target on CALLn/CALLXn
m	2-bit sub-opcode
i	1-bit sub-opcode
z	1-bit sub-opcode
imm4	4-bit immediate
imm6	6-bit immediate (PC-relative offset)

Field	Definition
imm7	7-bit immediate (for <code>MOVI .N</code>)
imm8	8-bit immediate
imm12	12-bit immediate
imm16	16-bit immediate
offset	18-bit PC-relative offset
ai4const	4-bit immediate, if 0 interpreted as -1, else sign-extended
b4const	4-bit encoded constant value
bbi	5-bit selector for Booleans in registers
sa	4- or 5-bit shift amount
sr	8-bit special register selector
x	1-bit MAC16 data register selector (m0 or m1 only)
y	1-bit MAC16 data register selector (m2 or m3 only)
w	2-bit MAC16 data register selector (m0, m1, m2, or m3)

3. Core Architecture

Topics:

- *Overview of the Core Architecture*
- *Processor-Configuration Parameters*
- *Registers*
- *Data Formats and Alignment*
- *Memory*
- *Reset*
- *Exceptions and Interrupts*
- *Instruction Summary*

The Xtensa Core Architecture provides a baseline set of instructions available in every Xtensa implementation. Having such a baseline eases the implementation of core software such as operating system ports and a compiler. This chapter describes that Core Architecture.

3.1 Overview of the Core Architecture

The Xtensa Instruction Set is the product of extensive research into the right balance of features to best address the needs of the embedded processor market. It borrows the best features of other architectures as well as bringing new ISA innovations of its own. While the Xtensa ISA derives most of its features from RISC, it has targeted areas in which older CISC architectures have been strongest, such as compact code.

The Xtensa core ISA is implemented as a set of 24-bit instructions that perform 32-bit operations. The instruction width was chosen primarily with code-size economy in mind. The instructions themselves were selected for their utility in a wide range of embedded applications. The core ISA has many powerful features, such as compound operation instructions, that enhance its fit to embedded applications, but it avoids features that would benefit some applications at the expense of cost or power on others (for example, features that require extra register-file ports). Such features can be implemented in the Xtensa architecture using options and coprocessors specifically targeted at a particular application area.

The Xtensa ISA is organized as a core set of instructions with various optional packages that extend the functionality for specific application areas. This allows the designer to include only the required functionality in the processor core, maximizing the efficiency of the solution. The core ISA provides the functionality required for general control applications, and excels at decision-making and bit and byte manipulation. The core also provides a target for third-party software, and for this reason deletions from the core are not supported. Conversely, numeric computing applications such as digital signal processing are best done with optional ISA packages appropriate for specific application areas, such as the MAC16 Option for integer filters, or the Floating-Point Coprocessor Option for high-end audio processing.

3.2 Processor-Configuration Parameters

[Core Processor-Configuration Parameters](#) lists the processor-configuration parameters that are required in the core architecture. Additional processor-configuration parameters are listed with each option described in [Architectural Options](#) on page 73.

Table 5: Core Processor-Configuration Parameters

Parameter	Description	Valid Values
<code>msbFirst</code>	Byte order	0 or 1 0 → Little-endian (least significant bit first) 1 → Big-endian (most significant bit first)

3.3 Registers

Core-Architecture Set lists the core-architecture registers. Each register is described in the sections that follow. Additional registers are added with many of the options described in *Architectural Options* on page 73. The complete set of registers that are predefined in the architecture, including all registers used by the architectural options, is listed in *Alphabetical List of Processor State*.

Table 6: Core-Architecture Set

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
AR	16 ²	32	Address registers (general registers)	R/W	—
PC	1	32	Program counter	R/W	—
SAR	1	6	Shift-amount register	R/W	3

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` (*Processor Control Instructions*. Processor state is listed in *Alphabetical List of Processor State*. A dash (—) means that the register is not a Special Register.

2. See *Windowed Register Option* on page 240.

3.3.1 General (*AR*) Registers

Each instruction contains up to three 4-bit general-register specifiers, each of which can select one of 16 32-bit registers. These general registers are named address registers (*AR*) to distinguish them from coprocessor registers, which in many systems might serve as “data” registers. However, the *AR* registers are not restricted to holding addresses; they can also hold data.

If the Windowed Register Option is configured, the address register file is extended and a mapping from virtual to physical registers is used.

The contents of the address register file are undefined after reset.

3.3.2 Shifts and the Shift Amount Register (*SAR*)

The ISA provides conventional immediate shifts (logical left, logical right, and arithmetic right), but it does not provide single-instruction shifts in which the shift amount is a register operand. Taking the shift amount from a general register can create a critical timing path.

Also, simple shifts do not extend efficiently to larger widths. Funnel shifts (where two data values are catenated on input to the shifter) solve this problem, but require too many operands. The ISA solves both problems by providing a funnel shift in which the shift amount is taken from the `SAR` register. Variable shifts are synthesized by the compiler using an instruction to compute `SAR` from the shift amount in a general register, followed by a funnel shift.

Another advantage is that a unidirectional funnel shifter can be manipulated to provide either right or left shifts based on the order of the source operands and transformation of the shift amount. The ISA facilitates implementations that exploit this to reduce the logic required by the shifter.

Funnel shifts are also useful for working with the 40-bit accumulator values created by the MAC16 Option.

To facilitate unsigned bit-field extraction, the `EXTUI` instructions take a 4-bit mask field that specifies the number of bits to mask the result of the shift. The 4-bit field specifies masks of one to 16 ones. The `SRLI` instruction provides shifting without a mask.

The legal range of values for `SAR` is zero to 32, not zero to 31, so `SAR` is defined as six bits. The use of `SRC`, `SRA`, `SLL`, or `SRL` when `SAR > 32` is undefined.

`SAR` is undefined after processor reset.

The funnel shifter can also be used efficiently for byte alignment of unaligned memory data. To load four bytes from an arbitrary byte boundary (in a processor that does *not* have the Unaligned Exception Option), use the following code:

```
l32i      a4, a3, 0
l32i      a5, a3, 4
ssa81     a3
src       a4, a5, a4
```

An unaligned block copy can be done (in a processor that does *not* have the Unaligned Exception Option) with the following code for little-endian and small changes for big-endian:

```
l32i      a6, a3, 0
ssa81     a3
loopnez   a4, endloop
loop:
l32i      a7, a3, 4
src       a8, a7, a6
s32i     a8, a2, 0
l32i      a6, a3, 8
src       a8, a6, a7
s32i     a8, a2, 4
addi     a2, a2, 8
addi     a3, a3, 8
endloop:
```


The overhead, compared to an aligned copy, is only one SRC per L32I.

3.3.3 Reading and Writing the Special Registers

The SAR register is part of the Non-Privileged Special Register set in the Xtensa ISA (the other registers in this set are associated with the architectural options). The contents of the special register in the Core Architecture can be read to an AR register with the read special register (`RSR.SAR`) instruction or written from an AR register with the write special register (`WSR.SAR`) instruction as shown in [Reading and Writing Special Registers](#). The exchange special register (`XSR.SAR`) instruction accomplishes the combined action of the read and write instructions.

Table 7: Reading and Writing Special Registers

Register Name	Special Register Number	RSR.SAR Instruction	WSR.SAR Instruction
SAR	3	$AR[t] \leftarrow 0^{26} \parallel SAR$	$SAR \leftarrow AR[t]_{5..0}$

3.4 Data Formats and Alignment

The Core Architecture supports byte, 2-byte, and 4-byte data formats. Two additional data formats are used in architectural options — a 32-bit single-precision format for the Floating-Point Coprocessor Option, and a 40-bit accumulator value for the [MAC16 Option](#) on page 91. The MAC16 format is not a memory-operand format, but rather a temporary format held in a special 40-bit accumulator register during MAC16 execution; the result can be moved to two 32-bit registers for further operation or storage.

[Operand Formats and Alignment](#) summarizes the width and alignment of each data type. The processor uses byte addressing for all data types stored in memory (that is, all except the MAC16 accumulator). Byte order can be specified as either big-endian or little-endian. In big-endian byte order, byte 0 is the most-significant (left-most) byte. In little-endian byte order, byte 0 is the least-significant (right-most) byte. When specifying a byte order, both the *byte order* and the *bit order* are specified: the two orderings always have the same most-significant and least-significant ends.

Table 8: Operand Formats and Alignment

Operand	Length	Alignment Address in Memory
Byte	8 bit	xxxx
2-byte	16 bits	xxx0
4-byte (word)	32 bits	xx00

Operand	Length	Alignment Address in Memory
IEEE-754 single-precision (Floating-Point Coprocessor Option)	32 bits	xx00
MAC16 accumulator (<i>MAC16 Option</i> on page 91)	40 bits	register image only (not in memory)

3.5 Memory

The Xtensa ISA is based on 32-bit virtual and physical memory addresses, which provides a 2^{32} or 4 GB address space for instructions and data.

3.5.1 Memory Addressing

Virtual Address Fields shows an example of the processor's interpretation of addresses when configured with caches. The widths of all fields are configurable, and in some cases the width may be zero (in particular, there are always zero ignored bits today). The cache index and cache tag will overlap if the page size is smaller than the size of a single way of the cache and if physical tags are used. See for more detail on memory addressing.

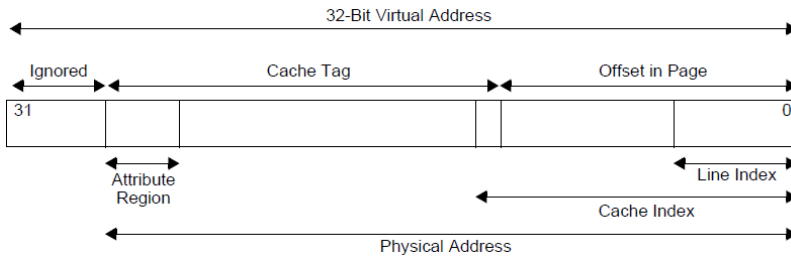


Figure 3: Virtual Address Fields

Without the Region Protection Option or the MMU Option, virtual and physical addresses are identical; if physical addresses are configured to be smaller than virtual addresses, virtual addresses are mapped to physical addresses only by truncation (high-order bits are ignored). With the Region Protection Option or the MMU Option, virtual page numbers are translated to physical page numbers.

Without the Region Protection Option or the MMU Option, the formal definition of virtual to physical translation is as follows (note that the `ring` parameter is ignored):

```
function ftranslate(vAddr, ring) -- fetch translate
    b ← vAddr(VABITS-1)..(VABITS-3)
    cacheattr ← CACHEATTR(b12'b11)..(b12'b00)
```

```

    attributes ← fcodecode(cacheattr)
    cause ← invalid(attributes) then InstructionFetchErrorCause else 0
    ftranslate ← (vAddrPABITS-1..0, attributes, cause)
endfunction ftranslate

function ltranslate (vAddr, ring) -- load translate
    b ← vAddr(VABITS-1)..(VABITS-3)
    cacheattr ← CACHEATTR(b12'b11)..(b12'b00)

    attributes ← lcodecode(cacheattr)
    cause ← invalid(attributes) then LoadStoreErrorCause else 0
    ltranslate ← (vAddrPABITS-1..0, attributes, cause)
endfunction ltranslate

function stranslate(vAddr, ring) -- store translate
    b ← vAddr(VABITS-1)..(VABITS-3)
    cacheattr ← CACHEATTR(b12'b11)..(b12'b00)

    attributes ← scodecode(cacheattr)
    cause ← invalid(attributes) then LoadStoreErrorCause else 0
    stranslate ← (vAddrPABITS-1..0, attributes, cause)
endfunction stranslate

```

Translation with the MMU Option is described in [MMU Option](#) on page 217.

The core ISA supports both little-endian (PC compatible) and big-endian (Internet compatible) address models as a configuration parameter. In this manual:

- `msbFirst = 1` is big-endian.
- `msbFirst = 0` is little-endian.

3.5.2 Addressing Modes

The core instruction set implements the register + immediate addressing mode. The core ISA does not implement auto-incrementing stores or indexed loads. However, such addressing modes are possible for coprocessors. For example, the Floating-Point Coprocessor Option implements indexed as well as immediate addressing modes.

3.5.3 Program Counter

The 32-bit program counter (PC) holds a byte address and can address 4 GB of virtual memory for instructions. However, when the Windowed Register Option is configured, the register-window call instructions only store the low 30 bits of the return address. Register-window return instructions leave the two most-significant bits of the PC unchanged. Therefore, subroutines called using register window instructions must be placed in the same 1 GB address region as the call.

3.5.4 Instruction Fetch

This section describes the execution loop of the processor using the notation of [Notation](#) on page 37. The individual instruction actions are represented by the `Inst()` statement, and are detailed in subsequent sections. Two versions of this code are supported; one for little-endian (`msbFirst = 0`) and one for big-endian (`msbFirst = 1`). This definition is in terms of a

hypothetical aligned 64-bit fetch, and should not be confused with the fetch algorithms used by specific Xtensa ISA implementations. Aligned 32-bit fetch and unaligned fetch are other possible implementations, which would produce logically equivalent results, but with different timings. Also, actual implementations would be expected to access memory only once for each fetch unit, not once per instruction as in the definition in [Little-Endian Fetch Semantics](#) on page 52 and [Big-Endian Fetch Semantics](#) on page 53.

The processor may speculatively fetch instructions following the address in the program counter. To facilitate this and to allow flexibility in the implementation, software must not position instructions within the last 64 bytes before a boundary where protection or cache attributes change. This exclusion does not apply if one of the two protections or attributes is invalid. Instructions may be placed within 64 bytes before a transition from valid to invalid or from invalid to valid — but not before any other transition. In addition, if the Windowed Register Option is implemented, software must not position instructions within the last 16 bytes of a 2^{30} (1 GB) boundary, to allow flexibility in the implementation of the register-window call and return instructions. The operation of the processor in these exclusion regions is not defined.

3.5.4.1 Little-Endian Fetch Semantics

Little-endian instruction fetch is defined as follows for a 64-bit fetch width (other fetch sizes are similar):

```

checkInterrupts() -- see "Checking for Interrupts on page 160"
vAddr0 ← PC31..3!3'b000 -- this example is 64-bit fetch
(pAddr0, attributes, cause) ← ftranslate(vAddr0, CRING)
if invalid(attributes) then
    EXCVADDR ← vAddr0
    Exception (cause)
    goto abortInstruction
endif
(mem0, error) ← ReadInstMemory(pAddr0, attributes, 8'b11111111)
                                     -- get start of instruction

if error then
    EXCVADDR ← vAddr0
    Exception (InstructionFetchErrorCause)
    goto abortInstruction
endif
b ← 0!PC2..0
if b2 = 0 or b1 = 0 or (b0 = 0 and mem0(b13'b011) = 1) then
    -- instruction contained within a single fetch (64 bits in this example)
    inst ← (undefined64!mem0)((b+2)13'b111)..(b13'b000)
else
    -- instruction crosses a fetch boundary (64 bits in this example)
    vAddr1 ← vaddr0 + 32'd8
    (pAddr1, attributes, cause) ← ftranslate(vAddr1, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr1
        Exception (cause)
        goto abortInstruction
    endif
    (mem1, error) ← ReadInstMemory(pAddr1,
                                   attributes, 8'b11111111)

if error then

```

```

        EXCVADDR ← vAddr1
        Exception (InstructionFetchErrorCause)
        goto abortInstruction
    endif
    inst ← (mem0|mem0) ((b+2)|3'b111)..(b|3'b000)
endif
-- now have a 24-bit instruction (8 bits undefined if 16-bit), break it into fields
op0 ← inst3..0
t ← inst7..4
s ← inst11..8
r ← inst15..12
op1 ← inst19..16
op2 ← inst23..20
imm8 ← inst23..16
imm12 ← inst23..12
imm16 ← inst23..8
offset ← inst23..6
n ← inst5..4
m ← inst7..6
-- compute nextPC (may be overridden by branches, etc.)
nextPC ← PC + (030 | (if op03 then 2'b10 else 2'b11))
if LCOUNT ≠ 032 and CLOOPENABLE and nextPC = LEND then
    LCOUNT ← LCOUNT - 1
    nextPC ← LBEG
endif
-- execute instruction
Inst()
checkIcount()
abortInstruction:
    PC ← nextPC

```

3.5.4.2 Big-Endian Fetch Semantics

Big-endian instruction fetch is defined as follows for a 64-bit fetch width (other fetch sizes are similar):

```

checkInterrupts()          -- see "Checking for Interrupts on page 160"
vAddr0 ← PC31..3|3'b000    -- this example is 64-bit fetch
(pAddr0, attributes, cause) ← ftranslate(vAddr0, CRING)
if invalid(attributes) then
    EXCVADDR ← vAddr0
    Exception (cause)
    goto abortInstruction
endif
(mem0, error) ← ReadInstMemory(pAddr0, attributes, 8'b11111111)
    -- get start of instruction
if error then
    EXCVADDR ← vAddr0
    Exception (InstructionFetchErrorCause)
    goto abortInstruction
endif
b ← 0|PC2..0
p0 ← b xor 14
p2 ← (b + 2) xor 14
if b2 = 0 or b1 = 0 or (b0 = 0 and (mem0|undefined64)(p0|3'b111) = 1)
then
    -- instruction contained within a single fetch (64 bits in this example)
    inst ← (mem0|undefined64)(p0|3'b111)..(p2|3'b000)
else
    -- instruction crosses a fetch boundary (64 bits in this example)

```

```

vAddr1 ← vaddr0 + 32'd8
(pAddr1, attributes, cause) ← ftranslate(vAddr1, CRING)
if invalid(attributes) then
    EXCVADDR ← vAddr1
    Exception (cause)
    goto abortInstruction
endif
(mem1, error) ← ReadInstMemory(pAddr1,
                                attributes, 8'b11111111)
if error then
    EXCVADDR ← vAddr1
    Exception (InstructionFetchErrorCause)
    goto abortInstruction
endif
inst ← (mem0|mem1) (p0|3'b111)..(p2|3'b000)
endif
-- now have a 24-bit instruction (8 bits undefined if 16-bit), break it into fields
op0 ← inst23..20
t ← inst19..16
s ← inst15..12
r ← inst11..8
op1 ← inst7..4
op2 ← inst3..0
imm8 ← inst7..0
imm12 ← inst11..0
imm16 ← inst15..0
offset ← inst17..0
n ← inst19..18
m ← inst17..16
-- compute nextPC (may be overridden by branches, etc.)
nextPC ← PC + (030 | (if op03 then 2'b10 else 3'b11))
if LCOUNT ≠ 032 and CLOOPENABLE and nextPC = LEND then
    LCOUNT ← LCOUNT - 1
    nextPC ← LBEG
endif
-- execute instruction
Inst()
checkIcount ()
abortInstruction:
PC ← nextPC

```

3.6 Reset

When the processor emerges from the reset state, it initializes many registers. The ISA guarantees the values of some states after reset but leaves many others undefined. Actual Xtensa processor implementations will often define the values of state left undefined by the ISA. [Processor State](#) on page 265 contains information about each state value, including the value to which it is reset.

3.7 Exceptions and Interrupts

The core ISA does not include support for exceptions or interrupts. These are architectural options are described in [Options for Interrupts and Exceptions](#) on page 126. Software

running on a processor that is configured without an Exception Option should be well tested, as such a processor will do something unexpected if it encounters a software error.

3.8 Instruction Summary

[Core Instruction Summary](#) summarizes the core instructions included in all versions of the Xtensa architecture. The remainder of this section gives an overview of the core instructions.

Table 9: Core Instruction Summary

Instruction Category	Instructions ¹	Reference
Load	L8UI, L16SI, L16UI, L32I, L32R	Load Instructions on page 56
Store	S8I, S16I, S32I	Store Instructions on page 58
Memory ordering	MEMW, EXTW	Memory Access Ordering on page 60
Jump, Call	CALL0, CALLX0, RET J, JX	Jump and Call Instructions on page 61
Conditional branch	BALL, BNALL, BANY, BNONE BEC, BBCI, BBS, BBSI BEQ, BEQI, BEQZ BNE, BNEI, BNEZ BGE, BGEI, BGEU, BGEUI, BGEZ BLT, BLTI, BLTU, BLTUI, BLTZ	Conditional Branch Instructions on page 62
Move	MOVI, MOVEQZ, MOVGEZ, MOVLtz, MOVNEZ	Move Instructions on page 65
Arithmetic	ADDI, ADDMI, ADD, ADDX2, ADDX4, ADDX8, SUB, SUBX2, SUBX4, SUBX8, NEG, ABS, SALT, SALTU	Arithmetic Instructions on page 66
Bitwise logical	AND, OR, XOR	Bitwise Logical Instructions on page 68

Instruction Category	Instructions ¹	Reference
Shift	EXTUI, SRLI, SRAI, SLLI SRC, SLL, SRL, SRA SSL, SSR, SSAI, SSA8B, SSA8L	Shift Instructions on page 68
Processor control	RSR, WSR, XSR, RUR, WUR, ISYNC, RSYNC, ESYNC, DSYNC, FSYNC, NOP	Processor Control Instructions on page 70
1. These instructions are fully described in Instruction Descriptions on page 321.		

3.8.1 Load Instructions

Load instructions form a virtual address by adding a base register and an 8-bit unsigned offset. This virtual address is translated to a physical address if necessary. The physical address is then used to access the memory system (often through a cache). The memory system returns a data item (either 32, 64, 128, 256, or 512 bits, depending on the configuration). The load instructions then extract the referenced data from that memory item and either zero-extend or sign-extend the result to be written into a register. Unless the Unaligned Exception Option is enabled, the processor does not handle misaligned data or trap when a misaligned address is used; instead it simply loads the aligned data item containing the computed virtual address. This allows the funnel shifter to be used with a pair of loads to reference data on any byte address.

Only the loads `L32I`, `L32I.N`, and `L32R` can access InstRAM and InstROM locations. [Load Instructions](#) shows the loads in the Core Architecture.

Table 10: Load Instructions

Instruction	Format	Definition
<code>L8UI</code>	RR18 on page 656	8-bit unsigned load (8-bit offset)
<code>L16SI</code>	RR18 on page 656	16-bit signed load (8-bit shifted offset)
<code>L16UI</code>	RR18 on page 656	16-bit unsigned load (8-bit shifted offset)
<code>L32I</code>	RR18 on page 656	32-bit load (8-bit shifted offset)
<code>L32R</code>	RI16 on page 657	32-bit load PC-relative (16-bit negative word offset)

Because the operation of caches is implementation-specific, this manual does not provide a formal specification of cache access.

The following routines define the load instructions:

```
function ReadMemory (pAddr, attributes, bytemask)
    ReadMemory ← (Memory[pAddr], 0) -- for now, no cache
endfunction ReadMemory

function Load8 (vAddr)
    (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
        goto abortInstruction
    endif
    p ← pAddr2..0 xor msbFirst3
    (mem64, error) ← ReadMemory(pAddr31..3, attributes, 07-P1110P)
    mem8 ← mem64(p13'b111)..(p13'b000)
    Load8 ← (mem8, error)
endfunction Load8

function Load16 (vAddr)
    if UnalignedExceptionOption & Vaddr0 ≠ 1'b0 then
        EXCVADDR ← vAddr
        Exception (LoadStoreAlignmentCause)
        goto abortInstruction
    endif
    (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
    endif
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
        goto abortInstruction
    endif
    p ← pAddr2..1 xor msbFirst2
    (mem64, error) ← ReadMemory(pAddr31..3, attributes,
        (2'b00)3-P12'b111(2'b00)p)
    mem16 ← mem64(p14'b1111)..(p14'b0000)
    Load16 ← (mem16, error)
endfunction Load16

function Load32 (vAddr)
    if UnalignedExceptionOption & Vaddr1..0 ≠ 2'b00 then
        EXCVADDR ← vAddr
        Exception (LoadStoreAlignmentCause)
        goto abortInstruction
    endif
    (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
        goto abortInstruction
    endif
    p ← pAddr2 xor msbFirst
    (mem64, error) ← ReadMemory(pAddr31..3, attributes,
        (4'b0000)1-P14'b11111(4'b0000)P)
    mem32 ← mem64(p15'b11111)..(p15'b00000)
    Load32 ← (mem32, error)
endfunction Load32
```

```

function Load32Ring (vAddr, ring)
  if UnalignedExceptionOption & Vaddr1..0 ≠ 2'b00 then
    EXCVADDR ← vAddr
    Exception (LoadStoreAlignmentCause)
    goto abortInstruction
  endif
  (pAddr, attributes, cause) ← ltranslate(vAddr, ring)
  if invalid(attributes) then
    EXCVADDR ← vAddr
    Exception (cause)
    goto abortInstruction
  endif
  p ← pAddr2 xor msbFirst
  (mem64, error) ← ReadMemory(pAddr31..3, attributes,
    (4'b0000)1-P∥4'b1111∥(4'b0000)P)
  mem32 ← mem64(p15'b1111)..(p15'b0000)
  Load32 ← (mem32, error)
endfunction Load32Ring

function Load64 (vAddr)
  if UnalignedExceptionOption & Vaddr2..0 ≠ 3'b000 then
    EXCVADDR ← vAddr
    Exception (LoadStoreAlignmentCause)
    goto abortInstruction
  endif
  (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
  if invalid(attributes) then
    EXCVADDR ← vAddr
    Exception (cause)
    goto abortInstruction
  endif
  Load64 ← ReadMemory(pAddr31..3, attributes, 8'b11111111)
endfunction Load64

```

3.8.2 Store Instructions

Store instructions are similar to load instructions in address formation. Store memory errors are not synchronous exceptions; it is expected that the memory system will use an interrupt to indicate an error on a store.

Only the stores $S32I$ and $S32I.N$ can access InstRAM.

[Store Instructions](#) shows the loads in the Core Architecture.

Table 11: Store Instructions

Instruction	Format	Definition
$S8I$	RR18 on page 656	8-bit store (8-bit offset)
$S16I$	RR18 on page 656	16-bit store (8-bit shifted offset)
$S32I$	RR18 on page 656	32-bit store (8-bit shifted offset)

The following routines define the store instructions:

```

procedure WriteMemory (pAddr, attributes, bytemask, data64)
  -- for now, no cache if bytemask0 then
    Memory[pAddr]7..0 ← data647..0
  endif
  if bytemask1 then
    Memory[pAddr]15..8 ← data6415..8
  endif
  if bytemask2 then
    Memory[pAddr]23..16 ← data6423..16
  endif
  if bytemask3 then
    Memory[pAddr]31..24 ← data6431..24
  endif
  if bytemask4 then
    Memory[pAddr]39..32 ← data6439..32
  endif
  if bytemask5 then
    Memory[pAddr]47..40 ← data6447..40
  endif
  if bytemask6 then
    Memory[pAddr]55..48 ← data6455..48
  endif
  if bytemask7 then
    Memory[pAddr]63..56 ← data6463..56
  endif
endprocedure WriteMemory

procedure Store8 (vAddr, data8)
  (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
  if invalid(attributes) then
    EXCVADDR ← vAddr
    Exception (cause)
    goto abortInstruction
  endif
  p ← pAddr2..0 xor msbFirst3
  WriteMemory(pAddr31..3, attributes, 07-P1110P,
    undefined(7-P)13b000data8undefinedP13b000)
endprocedure Store8

procedure Store16 (vAddr, data16)
  if UnalignedExceptionOption & Vaddr0 ≠ 1'b0 then
    EXCVADDR ← vAddr
    Exception (LoadStoreAlignmentCause)
    goto abortInstruction
  endif
  (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
  if invalid(attributes) then
    EXCVADDR ← vAddr
    Exception (cause)
    goto abortInstruction
  endif
  p ← pAddr2..1 xor msbFirst2
  WriteMemory(pAddr31..3, attributes, (2'b00)3-P12'b11(2'b00)P,
    undefined(3-P)14b0000data16undefinedP14b0000)
endprocedure Store16

procedure Store32 (vAddr, data32)

  if UnalignedExceptionOption & Vaddr1..0 ≠ 2'b00 then

```

```

        EXCVADDR ← vAddr
        Exception (LoadStoreAlignmentCause)
        goto abortInstruction
    endif
    (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
        goto abortInstruction
    endif
    p ← pAddr2 xor msbFirst
    WriteMemory(pAddr31..3, attributes, (4'b0000)1-
P|4'b1111|(4'b0000)P,
                undefined(1-P)|5'b00000|data32|undefinedP|5'b00000)
endprocedure Store32

procedure Store32Ring (vAddr, data32, ring)
    if UnalignedExceptionOption & Vaddr1..0 ≠ 2'b00 then
        EXCVADDR ← vAddr
        Exception (LoadStoreAlignmentCause)
        goto abortInstruction
    endif
    (pAddr, attributes, cause) ← stranslate(vAddr, ring)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
        goto abortInstruction
    endif
    p ← pAddr2 xor msbFirst
    WriteMemory(pAddr31..3, attributes, (4'b0000)1-
P|4'b1111|(4'b0000)P,
                undefined(1-P)|5'b00000|data32|undefinedP|5'b00000)
endprocedure Store32Ring

procedure Store64 (vAddr, data64)
    if UnalignedExceptionOption & Vaddr2..0 ≠ 3'b000 then
        EXCVADDR ← vAddr
        Exception (LoadStoreAlignmentCause)
        goto abortInstruction
    endif
    (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
        goto abortInstruction
    endif
    WriteMemory(pAddr31..3, attributes, 8'b11111111, data64)
endprocedure Store64

```

3.8.3 Memory Access Ordering

Xtensa implementations can perform ordinary load and store operations in any order, as long as loads return the last (as defined by program execution order) values stored to each byte of the load address for a single processor and a simple memory. This flexibility is appropriate because most memory accesses require only these semantics and some implementations may be able to execute programs significantly faster by exploiting non-program order memory access. The Xtensa ISA only requires that implementations follow a simplified version of the Release Consistency model¹ of memory access ordering, although many

implement stricter orderings for simplicity. For more on the Xtensa memory order semantics, see [Multiprocessor Synchronization Option](#) on page 115.

However, some load and store instructions are executed not just to read and write storage, but to cause some side effects on some other part of the system (for example, another processor or an I/O device). In C and C++, such variables must be declared `volatile`. Loads and stores to such locations must be executed in program order. The Xtensa ISA therefore provides an instruction that can be used to give program ordering of load and store memory accesses.

The `MEMW` instruction causes all memory and cache accesses (loads, stores, acquires, releases, prefetches, and cache operations, but *not* instruction fetches) before itself in program order to access memory before all memory and cache accesses (but *not* instruction fetches) after. At least one `MEMW` should be executed in between every load or store to a `volatile` variable. The Multiprocessor Synchronization Option provides some additional instructions that also affect memory ordering in a more focused fashion. `MEMW` has broader applications than these other instructions (for example, when reading and writing device registers), but it also may affect performance more than the synchronization instructions.

The `EXTW` instruction is similar to `MEMW`, but it separates all external effects of instructions before the `EXTW` in program order from all external effects of instructions after the `EXTW` in program order. `EXTW` is a superset of `MEMW`, and includes memory accesses in what it orders.

[Memory Order Instructions](#) shows the memory ordering instructions in the Core Architecture.

Table 12: Memory Order Instructions

Instruction	Format	Definition
<code>MEMW</code>		Order memory accesses before with memory access after
<code>EXTW</code>		Order all external effects before with all external effects after

3.8.4 Jump and Call Instructions

The unconditional branch instruction, `J`, has a longer range (PC-relative) than conditional branches. Calls have a slightly longer range because they target 32-bit aligned addresses. In addition, jump and call indirect instructions provide support for case dispatch, function variables, and dynamic linking.

¹ Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy, “*Memory consistency and event ordering in scalable shared-memory multiprocessors*,” Proceedings of the 17th Annual International Symposium on Computer Architecture, pages 15-26, May 1990.

[Jump and Call Instructions](#) shows the jump and call instructions.

Table 13: Jump and Call Instructions

Instruction	Format	Definition
CALL0	CALL on page 657	Call subroutine, PC-relative
CALLX0	CALLX on page 657	Call subroutine, address in register
J	CALLX on page 657	Unconditional jump, PC-relative
JX	CALLX on page 657	Unconditional jump, address in register
RET	CALLX on page 657	Subroutine return—jump to return address. Used to return from a routine called by CALL0 / CALLX0 .

3.8.5 Conditional Branch Instructions

The branch instructions in [Conditional Branch Instructions](#) compare a register operand against zero, an immediate, or a second register value and conditional branch based on the result of the comparison. Compound compare and branch instructions improve code density and performance compared to other ISAs. All branches are PC-relative; the immediate field contains the difference between the target PC and the current PC plus four. The use of a PC-relative offset of minus three to zero is illegal and reserved for future use.

Table 14: Conditional Branch Instructions

Instruction	Format	Definition
BEQZ	BRI12 on page 658	Branch if equal to zero
BNEZ	BRI12 on page 658	Branch if not equal to zero
BGEZ	BRI12 on page 658	Branch if greater than or equal to zero
BLTZ	BRI12 on page 658	Branch if less than zero
BEQI	BRI8 on page 658	Branch if equal immediate ¹
BNEI	BRI8 on page 658	Branch if not equal immediate ¹

Instruction	Format	Definition
BGEI	BRI8 on page 658	Branch if greater than or equal immediate ¹
BLTI	BRI8 on page 658	Branch if less than immediate ¹
BGEUI	BRI8 on page 658	Branch if greater than or equal unsigned immediate ²
BLTUI	BRI8 on page 658	Branch if less than unsigned immediate ²
BBCI	RRI8 on page 656	Branch if bit clear immediate
BBSI	RRI8 on page 656	Branch if bit set immediate
BEQ	RRI8 on page 656	Branch if equal
BNE	RRI8 on page 656	Branch if not equal
BGE	RRI8 on page 656	Branch if greater than or equal
BLT	RRI8 on page 656	Branch if less than
BGEU	RRI8 on page 656	Branch if greater than or equal unsigned
BLTU	RRI8 on page 656	Branch if less than Unsigned
BANY	RRI8 on page 656	Branch if any of masked bits set
BNONE	RRI8 on page 656	Branch if none of masked bits set (All Clear)
BALL	RRI8 on page 656	Branch if all of masked bits set
BNALL	RRI8 on page 656	Branch if not all of masked bits set
BBC	RRI8 on page 656	Branch if bit clear
BBS	RRI8 on page 656	Branch if bit set
<p>1. See Branch Immediate (<i>b4const</i>) Encodings for encoding of signed immediate constants.</p>		

Instruction	Format	Definition
2. See Branch Unsigned Immediate ($b4_{constu}$) Encodings for encoding of unsigned immediate constants.		

The encodings for the branch immediate constant ($b4_{const}$) field and the branch unsigned immediate constant ($b4_{constu}$) fields, shown in [Branch Immediate \(\$b4_{const}\$ \) Encodings](#) and [Branch Unsigned Immediate \(\$b4_{constu}\$ \) Encodings](#), specify one of the sixteen most frequent compare immediates for each type of constant.

Table 15: Branch Immediate ($b4_{const}$) Encodings

Encoding	Decimal Value of Immediate	Hex Value of Immediate
0	-1	32'hFFFFFFF
1	1	32'h00000001
2	2	32'h00000002
3	3	32'h00000003
4	4	32'h00000004
5	5	32'h00000005
6	6	32'h00000006
7	7	32'h00000007
8	8	32'h00000008
9	10	32'h0000000A
10	12	32'h0000000C
11	16	32'h00000010
12	32	32'h00000020
13	64	32'h00000040
14	128	32'h00000080
15	256	32'h00000100

Table 16: Branch Unsigned Immediate (b4constu) Encodings

Encoding	Decimal Value of Immediate	Hex Value of Immediate
0	32768	32'h00008000
1	65536	32'h00010000
2	2	32'h00000002
3	3	32'h00000003
4	4	32'h00000004
5	5	32'h00000005
6	6	32'h00000006
7	7	32'h00000007
8	8	32'h00000008
9	10	32'h0000000A
10	12	32'h0000000C
11	16	32'h00000010
12	32	32'h00000020
13	64	32'h00000040
14	128	32'h00000080
15	256	32'h00000100

3.8.6 Move Instructions

`MOVI` sets a register to a constant encoded in the instruction. The conditional move instructions shown in *Move Instructions* are used for branch avoidance.

Table 17: Move Instructions

Instruction	Format	Definition
MOVI	<i>RRR</i> on page 656	Load register with 12-bit signed constant
MOVEQZ	<i>RRR</i> on page 656	Conditional move if zero
MOVNEZ	<i>RRR</i> on page 656	Conditional move if non-zero
MOVLTZ	<i>RRR</i> on page 656	Conditional move if less than zero
MOVGEZ	<i>RRR</i> on page 656	Conditional move if greater than or equal to zero

3.8.7 Arithmetic Instructions

The arithmetic instructions that *Arithmetic Instructions* lists include add and subtract with a small shift for address calculations and for synthesizing constant multiplies. The `ADDMI` instruction is included for extending the range of load and store instructions.

Table 18: Arithmetic Instructions

Instruction	Format	Definition
ADD	<i>RRR</i> on page 656	Add two registers $AR[r] \leftarrow AR[s] + AR[t]$
ADDX2	<i>RRR</i> on page 656	Add register to register shifted by 1 $AR[r] \leftarrow (AR[s]_{30..0} \parallel 0) + AR[t]$
ADDX4	<i>RRR</i> on page 656	Add register to register shifted by 2 $AR[r] \leftarrow (AR[s]_{29..0} \parallel 0^2) + AR[t]$
ADDX8	<i>RRR</i> on page 656	Add register to register shifted by 3 $AR[r] \leftarrow (AR[s]_{28..0} \parallel 0^3) + AR[t]$
SUB	<i>RRR</i> on page 656	Subtract two registers $AR[r] \leftarrow AR[s] - AR[t]$

Instruction	Format	Definition
SUBX2	<i>RRR</i> on page 656	Subtract register from register shifted by 1 $AR[r] \leftarrow (AR[s]_{30..0} \parallel 0) - AR[t]$
SUBX4	<i>RRR</i> on page 656	Subtract register from register shifted by 2 $AR[r] \leftarrow (AR[s]_{29..0} \parallel 0^2) - AR[t]$
SUBX8	<i>RRR</i> on page 656	Subtract register from register shifted by 3 $AR[r] \leftarrow (AR[s]_{28..0} \parallel 0^3) - AR[t]$
NEG	<i>RRR</i> on page 656	Negate $AR[r] \leftarrow 0 - AR[t]$
ABS	<i>RRR</i> on page 656	Absolute value $AR[r] \leftarrow \text{if } AR[s]_{31} \text{ then } 0 - AR[s] \text{ else } AR[s]$
ADDI	<i>RR18</i> on page 656	Add signed constant to register $AR[t] \leftarrow AR[s] + (imm8_7^{24} \parallel imm8)$
ADDMI	<i>RR18</i> on page 656	Add signed constant shifted by 8 to register $AR[t] \leftarrow AR[s] + (imm8_7^{16} \parallel imm8 \parallel 0^8)$
SALT	<i>RRR</i> on page 656	Set AR if Less Than $AR[r] \leftarrow \text{if } AR[s] < AR[t] \text{ then } 0^{31} \parallel 1 \text{ else } 0^{32}$
SALTU	<i>RRR</i> on page 656	Set AR if Less Than Unsigned $AR[r] \leftarrow \text{if } AR[s] <_u AR[t] \text{ then } 0^{31} \parallel 1 \text{ else } 0^{32}$

3.8.8 Bitwise Logical Instructions

The bitwise logical instructions in [Bitwise Logical Instructions](#) provide a core set from which other logicals can be synthesized. Immediate forms of these instructions are not provided because the immediate would be only four bits.

Table 19: Bitwise Logical Instructions

Instruction	Format	Definition
AND	RRR on page 656	Bitwise logical AND $AR[r] \leftarrow AR[s] \text{ and } AR[t]$
OR	RRR on page 656	Bitwise logical OR $AR[r] \leftarrow AR[s] \text{ or } AR[t]$
XOR	RRR on page 656	Bitwise logical exclusive OR $AR[r] \leftarrow AR[s] \text{ xor } AR[t]$

3.8.9 Shift Instructions

The shift instructions in [Shift Instructions](#) provide a rich set of operations while avoiding critical timing paths. See [Shifts and the Shift Amount Register \(SAR\)](#) on page 47 for more information.

Table 20: Shift Instructions

Instruction	Format	Definition
EXTUI	RRR on page 656	Extract unsigned field immediate Shifts right by 0 . . 31 and ANDs with a mask of 1 . . 16 ones The operation of this instruction when the number of mask bits exceeds the number of significant bits remaining after the shift is undefined and reserved for future use.
SLLI	RRR on page 656	Shift left logical immediate by 1 . . 31 bit positions (see for encoding of the immediate value).
SRLI	RRR on page 656	Shift right logical immediate by 0 . . 15 bit positions

Instruction	Format	Definition
		There is no SRLI for shifts ≥ 16 ; use EXTUI instead.
SRAI	<i>RRR</i> on page 656	Shift right arithmetic immediate by 0 . . 31 bit positions
SRC	<i>RRR</i> on page 656	Shift right combined (a funnel shift with shift amount from SAR) The two source registers are catenated, shifted, and the least significant 32 bits returned.
SRA	<i>RRR</i> on page 656	Shift right arithmetic (shift amount from SAR)
SLL	<i>RRR</i> on page 656	Shift left logical (Funnel shift AR[s] and 0 by shift amount from SAR)
SRL	<i>RRR</i> on page 656	Shift right logical (Funnel shift 0 and AR[s] by shift amount from SAR)
SSA8B	<i>RRR</i> on page 656	Set shift amount register (SAR) for big-endian byte align The τ field must be zero.
SSA8L	<i>RRR</i> on page 656	Set shift amount register (SAR) for little-endian byte align
SSR	<i>RRR</i> on page 656	Set shift amount register (SAR) for shift right logical This instruction differs from WSR to SAR in that only the five least significant bits of the register are used.
SSL	<i>RRR</i> on page 656	Set shift amount register (SAR) for shift left logical
SSAI	<i>RRR</i> on page 656	Set shift amount register (SAR) immediate

3.8.10 Processor Control Instructions

Processor Control Instructions contains processor control instructions. The `RSR.*`, `WSR.*`, and `XSR.*` instructions read, write, and exchange Special Registers for both the Core Architecture and the architectural options, as detailed in *Numerical List of Special Registers*. They save and restore context, process interrupts and exceptions, and control address translation and attributes. The `XSR.*` instruction reads and writes both the Special Register, and `AR[t]`. It combines the `RSR.*` and `WSR.*` operations to exchange the Special Register with `AR[t]`. The `XSR.*` instruction is not present in T1030 and earlier processors.

The `xSYNC` instructions synchronize Special Register writes and their uses. See *Processor State* on page 265 for more information on how `xSYNC` instructions are used. These synchronization instructions are separate from the synchronization instructions used for multiprocessors, which are described in *Multiprocessor Synchronization Option* on page 115.

On some Xtensa implementations the latency of `RSR` is greater than one cycle, and so it is advantageous to schedule uses of the `RSR` result away from the `RSR` to avoid an interlock.

The point at which `WSR.*` or `XSR.*` to most Special Registers affects subsequent instructions is not defined (`SAR` and `ACC` are exceptions). In these cases, *Numerical List of Special Registers* explains how to ensure the effects are seen by a particular point in the instruction stream (typically involving the use of one of the `ISYNC`, `RSYNC`, `ESYNC`, or `DSYNC` instructions). A `WSR.*` or `XSR.*` followed by a `RSR.*` of the same register must be separated by an `ESYNC` instruction to guarantee the value written is read back. A `WSR.PS` or `XSR.PS` followed by a `RSIL` also requires an `ESYNC` instruction.

Table 21: Processor Control Instructions

Instruction	Format	Definition
<code>RSR</code>	RSR on page 657	Read Special Register
<code>WSR</code>	RSR on page 657	Write Special Register
<code>XSR</code>	RSR on page 657	Exchange Special Register (combined <code>RSR</code> and <code>WSR</code>) Not present in T1030 and earlier processors
<code>ISYNC</code>	RRR on page 656	Instruction fetch synchronize: Waits for all previously fetched load, store, cache, and special register write instructions that affect instruction fetch to be performed before fetching the next instruction.

Instruction	Format	Definition
RSYNC	RRR on page 656	Instruction register synchronize: Waits for all previously fetched WSR and XSR instructions to be performed before interpreting the register fields of the next instruction. This operation is also performed as part of ISYNC.
ESYNC	RRR on page 656	Register value synchronize: Waits for all previously fetched WSR and XSR instructions to be performed before the next instruction uses any register values. This operation is also performed as part of ISYNC and RSYNC.
DSYNC	RRR on page 656	Load/store synchronize: Waits for all previously fetched WSR and XSR instructions to be performed before interpreting the virtual address of the next load or store instruction. This operation is also performed as part of ISYNC, RSYNC, and ESYNC.
FSYNC	RRR on page 656	Fetch synchronize: Implementation defined.
NOF	RRR on page 656	No operation

4. Architectural Options

Topics:

- [Option Introduction](#)
- [Core Architecture](#)
- [Options for Additional Instructions](#)
- [Options for Interrupts and Exceptions](#)
- [Options for Local Memory](#)
- [Hardware Alignment Option](#)
- [Memory ECC/Parity Option](#)

This chapter defines the Xtensa ISA options. Each option adds some associated configuration resources and capabilities. Some options are dependent on the implementation of other options. These interdependencies, if any, are listed as *Prerequisites* at the beginning of the description of each option. The additional parameters required to define the option, the new state and instructions added by the option, and any other new features (such as exceptions) added by the option are listed and the operation of the option is described.

4.1 Option Introduction

This section discusses the purpose of options, and provides an overview.

4.1.1 Purpose of Options

The Xtensa architecture gains part of its flexibility by defining options. This chapter describes those options and the state and instructions that exist when each option is configured. Additional state and instructions may be added to any configuration by use of the TIE language as described in [System-Specific Instructions—The TIE Language](#).

Some options cannot be configured unless certain other options are also configured. These other options are listed in a bullet labeled "Prerequisites" near the beginning of the option description. The architecture allows many combinations of options, but some options cannot be configured along with certain other options. These options are listed in a bullet labeled "Incompatible Options" near the beginning of the option description. A third bullet is sometimes included in the set to contain various "Compatibility Notes."

Incompatible Options and Compatibility Notes refer only to compatibilities or incompatibilities which are architectural in nature and are expected to apply to all possible configurations. However, not all compatible combinations of options can actually be configured. The Data Book for a product should be consulted to determine which combinations of options can be configured.

4.1.2 Overview of Options

[Core Architecture](#) on page 77 provides a synopsis of the Core Architecture (covered in more detail in [Core Architecture](#) on page 45) in a format similar to the format used for the options. The Instruction Set options available with an Xtensa processor are listed in five groups below.

[Options for Additional Instructions](#) on page 82 lists options whose primary function is to add new instructions to the processor's instruction set, including:

- The [Code Density Option](#) on page 82 adds 16-bit encodings of the most frequently used 24-bit instructions for higher code density.
- The [Loop Option](#) on page 84 adds a "zero overhead loop," which requires neither the extra instruction for a branch at the end of a loop nor the additional delay slots that would result from the taken branch. A few fixed cycles of overhead mean that each iteration of the loop pays no cost for the loop branch.
- The [Extended L32R Option](#) on page 86 allows an additional choice in the addressing mode of the `L32R` instruction.
- The [16-bit Integer Multiply Option](#) on page 87 adds signed and unsigned 16x16 multiplication instructions that produce 32-bit results.
- The [32-bit Integer Multiply Option](#) on page 88 adds signed and unsigned 32x32 multiplication instructions that produce high and low parts of a 64-bit result.

- The *32-bit Integer Divide Option* on page 90 implements signed and unsigned 32-bit division and remainder instructions.
- The *MAC16 Option* on page 91 adds multiply-accumulate functions that are useful in digital signal processing (DSP).
- The *Miscellaneous Operations Option* on page 94 provides a series of instructions useful for some applications, but which are not necessary for others. By making these optional, the Xtensa architecture allows the designer to choose only those additional instructions that benefit the application.
- The *Boolean Option* on page 97 adds a set of Boolean registers, which can be set and cleared by user instructions and that can be used as branch conditions.
- The *Floating-Point Coprocessor Option* on page 99 adds a floating-point unit for single-precision floating-point and, optionally, for double-precision floating-point.
- The *Multiprocessor Synchronization Option* on page 115 adds acquire and release instructions with specific memory ordering relationships to the other Xtensa memory access instructions.
- The *Conditional Store Option* on page 118 adds a compare and swap type atomic operation to the instruction set.
- The *Exclusive Access Option* on page 123 adds a load exclusive and store exclusive method for atomic operations to the instruction set.

Options for Interrupts and Exceptions on page 126 lists options whose primary function is to add and control exceptions and interrupts, including:

- The *Exception Option 2* on page 126 adds the basic functions needed for the processor to take, and return from, exceptions.
- The *Relocatable Vector Option* on page 147 adds the ability for the exception vectors to be relocated at run time.
- The *Unaligned Exception Option* on page 148 adds an exception for memory accesses that are not aligned by their own size. They may then be emulated in software.
- The *Coprocessor Context Option* on page 149 allows the grouping of certain states in the processor and adds an enable bit, which allows for lazy context switching by taking an exception when state in that group is accessed.
- The *Interrupt Option* on page 151 builds upon the Exception Option 2 to add a flexible software prioritized interrupt system.
- The *High-Priority Interrupt Option* on page 157 adds a hardware prioritized interrupt system for higher performance.
- The *Timer Interrupt Option* on page 161 adds timers and interrupts, which are caused when the timer expires.

Options for Local Memory on page 162 lists options whose primary function is to add different kinds of memory, such as RAMs, ROMs, or caches to the processor, including:

- The *Hardware Alignment Option* on page 168 adds the ability for the hardware to handle unaligned accesses to data memory.

- The *Memory ECC/Parity Option* on page 168 provides the ability to add parity or ECC to cache and local memories.

Options for Memory Protection and Translation on page 183 lists options whose primary function is to control access to and manage memory, including:

- The *Region Protection Option* on page 196 adds protection on memory in eight segments.
- The *Region Translation Option* on page 202 adds protection on memory in eight segments and allows translations from one segment to another.
- The *Memory Protection Unit Option* on page 205 adds intermediate size Memory Protection Unit (MPU) hardware.
- The *MMU Option* on page 217 adds full paging virtual memory management hardware.

Options for Other Purposes on page 239 lists options that do not fall conveniently into one of the other groups, including:

- The *Windowed Register Option* on page 240 adds additional physical `AR` registers and a mapping mechanism, which together lead to smaller code size and higher performance.
- The *Processor Interface Option* adds a bus interface used by memory accesses, which are to locations other than local memories. It is used for cache misses for cacheable addresses as well as for cache bypass memory accesses.
- The *Miscellaneous Special Registers Option* on page 254 provides one to four scratch registers within the processor readable and writable by `RSR`, `WSR`, and `XSR`, which may be used for application-specific exceptions and interrupt processing tasks.
- The *Thread Pointer Option* on page 255 provides a Special Register that may be used for a thread pointer.
- The *Processor ID Option* on page 255 adds a register that software can use to distinguish which of several processors it is running on.
- The *CSR Parity Option* controls for checking parity on control and status registers.
- The *Secure Mode Bit Option* provides a simple secure mode, which allows implementations to have secure memory regions.
- The *Debug Option* on page 256 adds instructions-counting and breakpoint exceptions for debugging by software or external hardware.

The functionality of a fairly complete micro-controller is provided by enabling the Code Density Option, the Exception Option 2, the Interrupt Option, the High-Priority Interrupt Option, the Timer Interrupt Option, the Debug Option, and the Windowed Register Option.

The primary reason to disable the Code Density Option (16-bit instructions) is to provide maximum opcode space for extensions. The primary reason to disable the other options listed above is reduce the processor core area.

The choice of Cache, RAM, or ROM Options for instruction and data depends on the characteristics of the application. RAM is not as flexible as Cache, but it requires slightly less area because tags are not required. RAM may also be desirable when performance

predictability is required. ROM is even less flexible than RAM, but avoids the need to load the memory and offers some protection from program errors and tampering.

4.2 Core Architecture

The Core Architecture is not an option, but rather a minimum base of processor state and instructions, which allows system software and compiled code to run on all Xtensa implementations. There are no prerequisites or incompatible options, but the tables normally used to show option additions are used here to give the base set. [Core Architecture Processor-Configurations](#) through [Core Architecture Instructions](#) show Core Architecture processor configurations, processor state, and instructions.

Table 22: Core Architecture Processor-Configurations

Parameter	Description	Valid Values
msbFirst	Byte order for memory accesses	0 or 1 0 → Little-endian (least significant bit first) 1 → Big-endian (most significant bit first)

Table 23: Core Architecture Processor-State

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
AR	16	32	Address register file	R/W	—
PC	1	32	Program counter	—	—
SAR	1	6	Shift amount register	R/W	3
MEMCTL	1	22 ²	L1 Memory Control	R/W	97

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` ([Processor Control Instructions](#)). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.
2. Configurations without cache, branch prediction or L0 Instruction Buffer do not have any bits in `MEMCTL`.

Table 24: Core Architecture Instructions

Instruction ¹	Format	Definition
ABS	RRR on page 656	Absolute value
ADD	RRR on page 656	Add two registers
ADDI	RRI8 on page 656	Add a register and an 8-bit immediate
ADDMI	RRI8 on page 656	Add a register and a shifted 8-bit immediate
ADDX2/4/8	RRR on page 656	Add two registers with one of them shifted left by one/two/three
AND	RRR on page 656	Bitwise AND of two registers
BALL/BANY	RRI8 on page 656	Branch if all/any bits specified by a mask in one register are set in another register
BBC/BBS	RRI8 on page 656	Branch if the bit specified by another register is clear/set
BBCI/BBSI	RRI8 on page 656	Branch if the bit specified by an immediate is clear/set
BEQ	RRI8 on page 656	Branch if a register equals another register
BEQI	RRI8 on page 656	Branch if a register equals an encoded constant
BEQZ	BRI12 on page 658	Branch if a register equals zero
BGE	RRI8 on page 656	Branch if one register is greater than or equal to a register
BGEI	RRI8 on page 656	Branch if one register is greater than or equal to an encoded constant
BGEU	RRI8 on page 656	Branch if one register is greater or equal to a register as unsigned

Instruction ¹	Format	Definition
BGEUI	BRI8 on page 658	Branch if one register is greater or equal to an encoded constant as unsigned
BGEZ	BRI12 on page 658	Branch if a register is greater than or equal to zero
BLT	RRI8 on page 656	Branch if one register is less than a register
BLTI	BRI8 on page 658	Branch if one register is less than an encoded constant
BLTU	RRI8 on page 656	Branch if one register is less than a register as unsigned
BLTUI	RRI8 on page 656	Branch if one register is less than an encoded constant as unsigned
BLTZ	BRI12 on page 658	Branch if a register is less than zero
BNALL/BNONE	RRI8 on page 656	Branch if some/all bits specified by a mask in a register are clear in another register
BNE	RRI8 on page 656	Branch if a register does not equal a register
BNEI	RRI8 on page 656	Branch if a register does not equal an encoded constant
BNEZ	BRI12 on page 658	Branch if a register does not equal zero
CALL0	CALL on page 657	Call subroutine at PC plus offset, place return address in A0
CALLX0	CALLX on page 657	Call subroutine register specified location, place return address in A0
DSYNC/ESYNC	RRR on page 656	Wait for data memory/execution related changes to resolve
EXTUI	RRR on page 656	Extract field specified by immediates from a register

Instruction ¹	Format	Definition
EXTW	<i>RRR</i> on page 656	Wait for any possible external ordering requirement (added in RA-2004.1)
FSYNC	<i>RRR</i> on page 656	Fetch Synchronize (implementation defined)
ILL	<i>RRR</i> on page 656	Illegal instruction executed
ISYNC	<i>RRR</i> on page 656	Wait for instruction fetch related changes to resolve
J	<i>CALL</i> on page 657	Jump to PC plus offset
JX	<i>CALLX</i> on page 657	Jump to register specified location
L8UI	<i>RR18</i> on page 656	Load zero extended byte
L16SI/L16UI	<i>RR18</i> on page 656	Load sign/zero extended 16-bit quantity
L32I	<i>RR18</i> on page 656	Load 32-bit quantity
L32R	<i>RI16</i> on page 657	Load literal at offset from PC (or from LITBASE with the Extended L32R Option)
MEMW	<i>RRR</i> on page 656	Wait for any possible memory ordering requirement
MOVEQZ	<i>RRR</i> on page 656	Move register if the contents of a register is zero
MOVGEZ	<i>RRR</i> on page 656	Move register if the contents of a register is greater than or equal to zero
MOVI	<i>RR18</i> on page 656	Move a 12-bit immediate to a register
MOVLTZ	<i>RRR</i> on page 656	Move register if the contents of a register is less than zero
MOVNEZ	<i>RRR</i> on page 656	Move register if the contents of a register is not zero

Instruction ¹	Format	Definition
NEG	<i>RRR</i> on page 656	Negate a register
NOP	<i>RRR</i> on page 656	No operation (added as a full instruction in RA-2004.1)
OR	<i>RRR</i> on page 656	Bitwise OR two registers
RER	<i>RRR</i> on page 656	Read External Register
RET	<i>CALLX</i> on page 657	Subroutine return through A0
RSR.*	<i>RSR</i> on page 657	Read a Special Register
RSYNC	<i>RRR</i> on page 656	Wait for dispatch related changes to resolve
S8I/S16I/S32I	<i>RRR8</i> on page 656	Store byte/16-bit quantity/32-bit quantity
SALT/SALTU	<i>RRR</i> on page 656	Set AR if Less Than Signed/Unsigned
SLL/SLLI	<i>RRR</i> on page 656	Shift left logical by SAR/immediate
SRA/SRAI	<i>RRR</i> on page 656	Shift right arithmetic by SAR/immediate
SRC	<i>RRR</i> on page 656	Shift right combined by SAR with two registers as input and one as output
SRL/SRLI	<i>RRR</i> on page 656	Shift right logical by SAR/immediate
SSA8B/SSA8L	<i>RRR</i> on page 656	Use low 2-bits of address register to prepare SAR for SRC assuming big/little endian
SSAI	<i>RRR</i> on page 656	Set SAR to immediate value
SSL/SSR	<i>RRR</i> on page 656	Set SAR from register for left/right shift
SUB	<i>RRR</i> on page 656	Subtract two registers

Instruction ¹	Format	Definition
SUBX2/4/8	RRR on page 656	Subtract two registers with the un-negated one shifted left by one/two/three
WER	RRR on page 656	Write External Register
WSR.*	RSR on page 657	Write a special register
XOR	RRR on page 656	Bitwise XOR two registers
XSR.*	RRR on page 656	Read and write a special register in an exchange (added in T1040)
1. These instructions are fully described in Instruction Descriptions on page 321.		

4.3 Options for Additional Instructions

The options in this section have the primary function of adding new instructions to the processor's instruction set. The new instructions cover a variety of purposes including new architectural capabilities, higher performance on existing capabilities, and smaller code.

4.3.1 Code Density Option

This option adds 16-bit encodings of the most frequently used 24-bit instructions. When a 24-bit instruction can be encoded into a 16-bit form, the code-size savings is significant.

- Prerequisites: None
- Incompatible options: None
- Compatibility note: The additions made by this option were once considered part of the core architecture, thus compatibility with binaries for previous hardware might require the use of this option. Many available third-party software packages including some currently supported operating systems require the Code Density Option.
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.1.1 Code Density Option Architectural Additions

[Code Density Option Instruction Additions](#) shows this option's architectural additions.

Table 25: Code Density Option Instruction Additions

Instruction ¹	Format	Definition
ADD.N	RRRN on page 658	Add two registers (same as ADD instruction but with a 16-bit encoding).
ADDI.N	RRRN on page 658	Add register and immediate (-1 and 1..15).
BEQZ.N	RI16 on page 657	Branch if register is zero with a 6-bit unsigned offset (forward only).
BNEZ.N	RI16 on page 657	Branch if register is non-zero with a 6-bit unsigned offset (forward only).
BREAK.N ²	RRRN on page 658	This instruction is the same as BREAK but with a 16-bit encoding.
ILL.N	RRRN on page 658	Illegal instruction executed
L32I.N	RRRN on page 658	Load 32 bits, 4-bit offset
MOV.N	RRRN on page 658	Narrow move
MOVI.N	RI7 on page 658	Load register with immediate (-32..95).
NOP.N	RRRN on page 658	This instruction performs no operation. It is typically used for instruction alignment.
RET.N	RRRN on page 658	The same as RET but with a 16-bit encoding.
RETW.N ⁴	RRRN on page 658	The same as RETW but with a 16-bit encoding.
S32I.N	RRRN on page 658	Store 32 bits, 4-bit offset
<p>1. These instructions are fully described in Instruction Descriptions on page 321.</p> <p>2. Exists only if the Debug Option described in Debug Option on page 256 is configured.</p> <p>3. Exists only if the Windowed Register Option described in Windowed Register Option on page 240 is configured</p>		

4.3.1.2 Branches

For some implementations, branches to an instruction that crosses a 32-bit memory boundary may suffer a small performance penalty. The compiler (or assembler) is expected to align performance-critical branch targets such that their byte address is 0 mod 4, 1 mod 4, or for 16-bit instructions, 2 mod 4. This can be accomplished either by converting some previous 16-bit-encoded instructions back to their 24-bit form, or by inserting a 16-bit `NOB.N`.

4.3.2 Loop Option

The Loop Option adds the ability for the processor to execute a zero-overhead loop where the number of iterations (not counting an early exit) can be determined prior to entering the loop. This capability is useful in digital signal processing applications where the overhead of a branch in a heavily used loop is unacceptable. A single loop instruction defines both the beginning and end of a loop, as well as a count of how many times the loop will execute.

A Loop Buffer can be added to the Loop Option. It reduces power by holding part or all of the loop while it is executing, so that fewer accesses to L1 Instruction Memory are required.

- Prerequisites: None
- Incompatible options: None
- Compatibility note: The additions made by this option were once considered part of the core architecture, thus compatibility with binaries for previous hardware might require the use of this option. Many available third-party software packages including some currently supported operating systems require the Loop Option.
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.2.1 Loop Option Architectural Additions

[Loop Option Processor-Configuration Additions](#) through [Loop Option Instruction Additions](#) show this option's architectural additions.

Table 26: Loop Option Processor-Configuration Additions

Parameter	Description	Valid Values
LoopBufferSize	Size of buffer used to avoid I-memory accesses	Implementation-dependent

Table 27: Loop Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
LBEG	1	32	Loop begin	R/W	0

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
LEND	1	32	Loop end	R/W	1
LCOUNT	1	32	Loop count	R/W	2
MEMCTL [0]			L1 Memory Controls	R/W	97

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` (*Processor Control Instructions*). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.

Bit[0] of `MEMCTL` is added with this option if `LoopBufferSize` is greater than zero and controls whether or not the Loop Buffer is used to reduce Instruction Memory access during loops. The Loop Buffer is flushed in sequences which properly update Instruction Memory (such as that given with `ISYNC`) and when `MEMCTL [0]` is cleared.

`LBEG` and `LEND` are undefined after processor reset. `LCOUNT` is initialized to zero after processor reset. If `MEMCTL [0]` is added, it is initialized to one after processor reset.

Table 28: Loop Option Instruction Additions

Instruction ¹	Format	Definition
LOOP	BR18 on page 658	Set up a zero-overhead loop by setting <code>LBEG</code> , <code>LEND</code> , and <code>LCOUNT</code> special registers.
LOOPGTZ	BR18 on page 658	Set up a zero-overhead loop by setting <code>LBEG</code> , <code>LEND</code> , and <code>LCOUNT</code> special registers. Skip loop if <code>LCOUNT</code> is not positive.
LOOPNEZ	BR18 on page 658	Set up a zero-overhead loop by setting <code>LBEG</code> , <code>LEND</code> , and <code>LCOUNT</code> special registers. Skip loop if <code>LCOUNT</code> is zero.

1. These instructions are fully described in [Instruction Descriptions](#) on page 321.

4.3.2.2 Restrictions on Loops

Newer implementations may have a loop instruction at any alignment. For many older implementations, there is a restriction on instruction alignment for zero-overhead loops. In the restricted implementations, the first instruction after the `LOOP` instruction, which begins at the

address written to `LBEG` by the `LOOP` instruction, must be entirely contained within a single, naturally-aligned fetch width for the current configuration. This condition can always be met by meeting the somewhat more restrictive condition that the first instruction after the `LOOP` instruction must be entirely contained within a naturally aligned, power of two sized unit of a particular size. That size is the next larger power of two equal to or greater than the instruction length, but not less than 4 bytes. Some older implementations require the latter, more restrictive, condition.

The last instruction of the loop must not be a call, `ISYNC`, `WAITI`, or `RSR.LCOUNT`. If the last instruction of the loop is a taken branch, then the value of `LCOUNT` is undefined in some implementations.

4.3.2.3 Loops Disabled During Exceptions

Loops are disabled when `PS.EXCM` is set under the Exception Option 2 . This prevents program code from maliciously or accidentally setting `LEND` to an address in an exception handler and then causing the exception, thereby transitioning to Ring 0 while retaining control of the processor.

4.3.2.4 Loopback Semantics

The processor includes the following to compute the `PC` of the next instruction:

```
if LCOUNT ≠ 0 and CLOOPENABLE and nextPC = LEND then
    LCOUNT ← LCOUNT - 1
    nextPC ← LBEG
endif
```

The semantics above have some non-obvious consequences. A taken branch to the address in `LEND` does not cause a transfer to `LBEG`. Thus a taken branch to the `LEND` instruction can be used to exit the loop prematurely. This is why a call instruction as the last instruction of a loop will not do the obvious thing (the return will branch to the `LEND` address and exit the loop). To conditionally begin the next loop iteration, a branch to a `NOP` before `LEND` may be used.

4.3.3 Extended `L32R` Option

The Extended `L32R` Option adds functionality to the standard `L32R` instruction. The standard `L32R` instruction has an offset that can reach as far as 256kB below the current `PC`. In the case where an instruction RAM approaches or exceeds 256kB in size, accessing literal data becomes more difficult. This option is intended to ease the access to literal data by providing an optional separate literal base register.

- Prerequisites: None
- Incompatible options: [MMU Option](#) on page 217

- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

Performance improvements in newer implementations largely remove the need for this option. See [Performance of L32R Instruction](#) for more detail.

4.3.3.1 Extended L32R Option Architectural Additions

[Extended L32R Option Processor-State Additions](#) shows this option’s architectural additions.

Table 29: Extended L32R Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
LITBASE	1	21	Literal base ²	R/W	5

- Special Registers are accessed with `RSR`, `WSR`, and `XSR` ([Processor Control Instructions](#)). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.
- See [LITBASE Register Format](#) for the format of this register.

4.3.3.2 The Literal Base Register

The literal base (`LITBASE`) register contains 20 upper bits, which define the location of the literal base and one enable bit (`En`). When the enable bit is clear, the `L32R` instruction loads a literal at a negative offset from the PC. When the enable bit is set, the `L32R` instruction loads a literal at a negative offset from the address formed by the 20 upper bits of literal base and 12 lower bits of `12'h000`. See the `L32R` instruction description in [Instruction Descriptions](#) on page 321. [LITBASE Register Format](#) shows the `LITBASE` register format.



Figure 4: LITBASE Register Format

The enable bit of the literal base register is cleared after reset. The remaining bits are undefined after reset.

4.3.4 16-bit Integer Multiply Option

This option provides two instructions that perform 16×16 multiplication, producing a 32-bit result. It is typically useful for digital signal processing (DSP) algorithms that require 16 bits or less of input precision (32 bits of input precision is provided by the 32-bit Integer Multiply Option) and do not require more than 32-bit accumulation (as provided by the [MAC16 Option](#) on page 91). Because a 16×16 multiplier is one-fourth the area of a 32×32 multiplier, this

option is less costly than the 32-bit Integer Multiply Option. Because it lacks an accumulator and data registers, it is less costly than the [MAC16 Option](#) on page 91.

- Prerequisites: None
- Incompatible options: None
- See Also [MAC16 Option](#) on page 91 and [32-bit Integer Multiply Option](#) on page 88
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.4.1 16-bit Integer Multiply Option Architectural Additions

[16-bit Integer Multiply Option Instruction Additions](#) shows this option’s architectural additions. There are no configuration parameters associated with the MUL16 Option and no additional processor state.

Table 30: 16-bit Integer Multiply Option Instruction Additions

Instruction ¹	Format	Definition
MUL16S	RRR on page 656	Signed 16×16 multiplication of the least-significant 16 bits of AR[<i>s</i>] and AR[<i>t</i>], with the 32-bit product written to AR[<i>r</i>]
MUL16U	RRR on page 656	Unsigned 16×16 multiplication of the least-significant 16 bits of AR[<i>s</i>] and AR[<i>t</i>], with the 32-bit product written to AR[<i>r</i>]
1. These instructions are fully described in Instruction Descriptions on page 321.		

4.3.5 32-bit Integer Multiply Option

This option provides instructions that implement 32-bit integer multiplication as instructions. This provides single instruction targets for the multiplication operators of programming languages such as C. When this option is not enabled, the Xtensa compiler uses subroutine calls to implement 32-bit integer multiplication. Note that various algorithms may be used to implement multiplication, and some hardware implementations may be slower than the software implementations for some operand values. Implementations may allow a choice of algorithms through configuration parameters to optimize among area, speed, and other characteristics.

There is one sub-option within this option: Mul32High. It controls whether the MULSH and MULUH instructions are included or not. For some implementations, generating the high 32 bits of the product requires additional hardware, and so disabling this sub-option may reduce cost.

- Prerequisites: None
- Incompatible options: None
- See Also: [MAC16 Option](#) on page 91 and [16-bit Integer Multiply Option](#) on page 87
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.5.1 32-bit Integer Multiply Option Architectural Additions

[32-bit Integer Multiply Option Processor-Configuration Additions](#) and [32-Bit Integer Multiply Instruction Additions](#) show this option’s architectural additions. This option adds no new processor state.

Table 31: 32-bit Integer Multiply Option Processor-Configuration Additions

Parameter	Description	Valid Values
Mul32High	Determines whether the MULSH and MULUH instructions are included	0 or 1
MulAlgorithm	Determines the multiplication algorithm employed	Implementation-dependent

Table 32: 32-Bit Integer Multiply Instruction Additions

Instruction ¹	Format	Definition
MULL	RRR on page 656	Multiply low (return least-significant 32 bits of product)
MULUH ²	RRR on page 656	Multiply unsigned high (return most-significant 32 bits of product)
MULSH ²	RRR on page 656	Multiply signed high (return most-significant 32 bits of product)

1. These instructions are fully described in [Instruction Descriptions](#) on page 321.
2. These instructions are part of the Mul32High sub-option of 32-bit Integer Multiply Option.

4.3.6 32-bit Integer Divide Option

This option provides instructions that implement 32-bit integer division and remainder operations. When this option is not enabled, the Xtensa compiler uses subroutine calls to implement division and remainder. Note that various algorithms may be used to implement these instructions, and some hardware implementations may be slower than the software implementations for some operand values.

- Prerequisites: None
- Incompatible Options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.6.1 32-bit Integer Divide Option Architectural Additions

[32-bit Integer Divide Option Processor-Configuration Additions](#) through [32-bit Integer Divide Option Instruction Additions](#) show this option's architectural additions. This option adds no new processor state. This option does add a new exception, Integer Divide by Zero, which is raised when the divisor operand of a QUOS, QUOU, REMS, or REMU instruction contains zero.

Table 33: 32-bit Integer Divide Option Processor-Configuration Additions

Parameter	Description	Valid Values
DivAlgorithm	Determines the division algorithm employed	Implementation-dependent

Table 34: 32-bit Integer Divide Option Exception Additions

Exception	Description	EXCCAUSE value
IntegerDivideByZero	Exception raised when divisor is zero	6

Table 35: 32-bit Integer Divide Option Instruction Additions

Instruction ⁷	Format	Definition
QUOS	RRR on page 656	Quotient Signed (divide giving 32-bit quotient)
QUOU	RRR on page 656	Quotient Unsigned (divide giving 32-bit quotient)

Instruction ¹	Format	Definition
REMS	RRR on page 656	Remainder Signed (divide giving 32-bit remainder)
REMU	RRR on page 656	Remainder Unsigned (divide giving 32-bit remainder)
1. These instructions are fully described in Instruction Descriptions on page 321		

4.3.7 MAC16 Option

The MAC16 Option adds multiply-accumulate functions that are useful in DSP and other media-processing operations. The option adds a 40-bit accumulator (ACC), four 32-bit data registers (MR_[n]), and 72 instructions.

The multiplier operates on two 16-bit operands from either the address registers (AR) or MAC16 registers (MR). Each operand may be taken from either the low or high half of a register. The result of the operation is placed in the 40-bit accumulator. The MR registers and the low 32 bits and high 8 bits of the accumulator are readable and writable with the RSR, WSR, and XSR instructions. MR[0] and MR[1] can be used as the first multiplier input, and MR[2] and MR[3] can be used as the second multiplier input. Four of the 72 added instructions can load the MR registers with 32-bit values from memory in parallel with multiply-accumulate operations.

The accumulator (ACC) and data registers (MR) are undefined after reset.

- Prerequisites: None
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.7.1 MAC16 Option Architectural Additions

[MAC16 Option Processor-State Additions](#) and [MAC16 Option Instruction Additions](#) show this option's architectural additions.

Table 36: MAC16 Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
ACCLO	1	32	Accumulator low	R/W	16
ACCHI	1	8	Accumulator high	R/W	17

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
MR[0] ²	1	32	MAC16 register 0 (m0 in assembler)	R/W	32
MR[1] ²	1	32	MAC16 register 1 (m1 in assembler)	R/W	33
MR[2] ²	1	32	MAC16 register 2 (m2 in assembler)	R/W	34
MR[3] ²	1	32	MAC16 register 3 (m3 in assembler)	R/W	35

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` (*Processor Control Instructions*). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.
2. These registers are known as MR[0..3] in hardware and as m0..3 in the software.

Table 37: MAC16 Option Instruction Additions

Instruction ^{1,2}	Definition ³
LDDEC	Load MAC16 data register (MR) with auto decrement
LDINC	Load MAC16 data register (MR) with auto increment
MUL.AA.qq	Signed multiply of two address registers
MUL.AD.qq	Signed multiply of an address register and a MAC16 data register
MUL.DA.qq	Signed multiply of a MAC16 data register and an address register
MUL.DD.qq	Signed multiply of two MAC16 data registers
MULA.AA.qq	Signed multiply-accumulate of two address registers
MULA.AD.qq	Signed multiply-accumulate of an address register and a MAC16 data register

Instruction ^{1,2}	Definition ³
MULA.DA.qq	Signed multiply-accumulate of a MAC16 data register and an address register
MULA.DD.qq	Signed multiply-accumulate of two MAC16 data registers
MULS.AA.qq	Signed multiply/subtract of two address registers
MULS.AD.qq	Signed multiply/subtract of an address register and a MAC16 data register
MULS.DA.qq	Signed multiply/subtract of a MAC16 data register and an address register
MULS.DD.qq	Signed multiply/subtract of two MAC16 data registers
MULA.DA.qq.LDDEC	Signed multiply-accumulate of a MAC16 data register and an address register, and load a MAC16 data register with auto decrement
MULA.DA.qq.LDINC	Signed multiply-accumulate of a MAC16 data register and an address register, and load a MAC16 data register with auto increment
MULA.DD.qq.LDDEC	Signed multiply-accumulate of two MAC16 data registers, and load a MAC16 data register with auto decrement
MULA.DD.qq.LDINC	Signed multiply-accumulate of two MAC16 data registers, and load a MAC16 data register with auto increment
UMUL.AA.qq	Unsigned multiply of two address registers
<ol style="list-style-type: none"> 1. These instructions are fully described in Instruction Descriptions on page 321. 2. The qq opcode parameter indicates (by HH, HL, LH, or LL) whether the operands are taken from the Low or High 16-bit half of the AR or MR registers. The first q represents the location of the first operand; the second q represents the location of the second operand. 3. The destination for all product and accumulate results is the MAC16 accumulator 	

4.3.7.2 Use With CLAMPS Instruction

The CLAMPS instruction, implemented with the Miscellaneous Operations Option, is useful in conjunction with the MAC16 Option. It allows clamping results to 16 bits before storing to memory.

4.3.8 Miscellaneous Operations Option

These instructions can be individually enabled in groups to provide computational capability required by a few applications.

- Prerequisites: None
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.8.1 Miscellaneous Operations Option Architectural Additions

[Miscellaneous Operations Option Processor-Configuration Additions](#) and [Miscellaneous Operations Instruction Additions](#) show this option's architectural additions.

Table 38: Miscellaneous Operations Option Processor-Configuration Additions

Parameter	Description	Valid Values
InstructionCLAMPS	Enable the signed clamp instruction: CLAMPS	0 or 1
InstructionMINMAX	Enable the minimum and maximum value instructions: MIN, MAX, MINU, MAXU	0 or 1
InstructionNSA	Enabled the normalization shift amount instructions: NSA, NSAU	0 or 1
InstructionSEXT	Enable the sign extend instruction: SEXT	0 or 1

Table 39: Miscellaneous Operations Instruction Additions

Instruction [†]	Format	Definition
CLAMPS	RRR on page 656	Clamp to signed power of two range <pre> sign ← AR[s]₃₁ AR[r] ← if AR[s]_{30..(t+7)} = </pre>

Instruction ¹	Format	Definition
		<pre> sign^{24_t} then AR[s] else sign^(25_t) (not sign)^{t+7} </pre>
MAX	<i>RRR</i> on page 656	<p>Maximum value signed</p> <pre> AR[r] ← if AR[s] < AR[t] then AR[t] else AR[s] </pre>
MAXU	<i>RRR</i> on page 656	<p>Maximum value unsigned</p> <pre> AR[r] ← if (0 AR[s]) < (0 AR[t]) then AR[t] else AR[s] </pre>
MIN	<i>RRR</i> on page 656	<p>Minimum value signed</p> <pre> AR[r] ← if AR[s] < AR[t] then AR[s] else AR[t] </pre>
MINU	<i>RRR</i> on page 656	<p>Minimum value unsigned</p> <pre> AR[r] ← if (0 AR[s]) < (0 AR[t]) then AR[s] else AR[t] </pre>
NSA	<i>RRR</i> on page 656	<p>Normalization shift amount signed</p> <pre> AR[r] ← nsa¹(AR[s]₃₁, AR[s]) </pre> <p>NSA returns the number of contiguous bits in the most significant end of AR[s] that are equal to the sign bit (not counting the sign bit itself), or 31 if AR[s] = 0 or AR[s] = -1. The result may be</p>

Instruction ¹	Format	Definition
		used as a left shift amount such that the result of SLL on AR[s] will have bit31 ≠ bit30 (if AR[s] ≠ 0).
NSAU	RRR on page 656	Normalization shift amount unsigned $AR[r] \leftarrow nsa^1(0, AR[s])$ NSAU returns the number of contiguous zero bits in the most significant end of AR[s], or 32 if AR[s] = 0. The result may be used as a left shift amount such that the result of SLL on AR[s] will have bit31 ≠ 0 (if AR[s] ≠ 0).
SEXT	RRR on page 656	Sign extend <div style="background-color: #f0f0f0; padding: 5px; margin-top: 10px;"> $sign \leftarrow AR[s]_{t+7}$ $AR[r] \leftarrow sign^{(2^4-t)} \mid AR[s]_t$ $+7..0$ </div>
<p>1. These instructions are fully described in Instruction Descriptions on page 321.</p>		

4.3.9 Deposit Bits Option

This option provides an instruction for depositing a bit field from the least significant position of one register to an arbitrary position in another register.

- Prerequisites: None
- Incompatible options: The binary encoding must be changed if the [Floating-Point Coprocessor Option](#) on page 99 is configured.
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.9.1 Deposit Bits Option Architectural Additions

The following table shows this option's architectural additions. There are no configuration parameters associated with the Deposit Bits Option and no additional processor state.

Table 40: Deposit Bits Option Instruction Additions

Instruction ¹	Format	Definition
DEPBITS	<i>RRR</i> on page 656	Deposits a bit field from the least significant portion of $AR[s]$ to an arbitrary position in $AR[t]$.
1. These instructions are fully described in <i>Instruction Descriptions</i> on page 321.		

4.3.10 Boolean Option

This option makes a set of Boolean registers available, along with branches and other operations that refer to them. Multiple coprocessors and other TIE language extensions can use this set.

- Prerequisites: None
- Incompatible options: None
- For more information on option compatibility, see *Purpose of Options* on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.10.1 Boolean Option Architectural Additions

The following table show this option's architectural additions.

Table 41: Boolean Option Processor-State Additions

Register Mnemonic	Quantity	Width (Bits)	Register Name	R/W	Special Register Number ¹
BR ²	16	1	Boolean registers	R/W	4
<p>1. Special Registers are accessed with RSR, WSR, and XSR (<i>Processor Control Instructions</i>). Processor state is listed in <i>Table 127: Alphabetical List of Processor State</i> on page 266.</p> <p>2. This register is known as Special Register BR or as individual Boolean bits b0..15.</p>					

Table 42: Boolean Option Instruction Additions

Instruction ¹	Format	Definition
ALL4	<i>RRR</i> on page 656	4-Boolean and reduction (result is 1 if all of the four Booleans are true)

Instruction ¹	Format	Definition
ALL8	<i>RRR</i> on page 656	8-Boolean and reduction (result is 1 if all of the eight Booleans are true)
ANDB	<i>RRR</i> on page 656	Boolean and
ANDBC	<i>RRR</i> on page 656	Boolean and with complement
ANY4	<i>RRR</i> on page 656	4-Boolean or reduction (result is 1 if any of the four Booleans is true)
ANY8	<i>RRR</i> on page 656	8-Boolean or reduction (result is 1 if any of the eight Booleans is true)
BF	<i>RR18</i> on page 656	Branch if Boolean false
BT	<i>RR18</i> on page 656	Branch if Boolean true
MOVF	<i>RRR</i> on page 656	Conditional move if false
MOVT	<i>RRR</i> on page 656	Conditional move if true
ORB	<i>RRR</i> on page 656	Boolean or
ORBC	<i>RRR</i> on page 656	Boolean or with complement
XORB	<i>RRR</i> on page 656	Boolean exclusive or

1. These instructions are fully described in [Instruction Descriptions](#) on page 321

4.3.10.2 Booleans

A coprocessor test or comparison produces a Boolean result. The Boolean Option provides 16 single-bit Boolean registers for storing the results of coprocessor comparisons for testing in conditional move and branch instructions. Boolean logic may replace branches in some situations. Compared to condition codes used by other ISAs, these Booleans eliminate the bottleneck of having only a single place to store comparison results. It is possible, for example, to do multiple comparisons before the comparison results are used. For Single-Instruction Multiple-Data (SIMD) operations, Booleans provide up to 16 simultaneous compare results and conditionals.

Boolean-producing instructions generate only one sense of the condition (for example, = but not ≠); all Boolean uses allow for complementing of the Boolean. Multiple Booleans may be combined into a single Boolean using the `ANY4`, `ALL4`, and so forth instructions. For example, this is useful after a SIMD comparison to test if any or all of the elements satisfy the test, such as testing if any byte of a word is zero. `ANY2` and `ALL2` instructions are not provided; `ANDB` and `ORB` provide this functionality given `bs+0` and `bs+1` as arguments.

The Boolean registers are undefined after reset.

The Boolean registers are accessible from C using the `xtbool`, `xtbool2`, `xtbool4`, `xtbool8`, and `xtbool16` data types.

4.3.11 Floating-Point Coprocessor Option

The Floating-Point Coprocessor Option adds an IEEE754 compatible single-precision floating-point unit and, optionally, an IEEE754 compatible double-precision floating-point unit. Floating-point operations are useful for DSP that requires >16 bits of precision, such as audio compression and decompression. Also, DSP algorithms for less precise data are more easily coded using floating-point, and good performance is obtainable when programming in languages such as C.

- Prerequisites: [Boolean Option](#) on page 97

If the [Coprocessor Context Option](#) on page 149 is also present, it may be used to protect the state of this option from access.

4.3.11.1 Floating-Point Coprocessor Option Architectural Additions

The tables in this section show this option's architectural additions.

Table 43: Floating-Point Coprocessor Option Processor-Configuration Additions

Parameter	Description	Valid Values
<code>DoublePrecision</code>	Add double-precision operations as well	True, False

Table 44: Floating-Point Coprocessor Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Register Number ¹
FR	16	32 or 64 ²	Floating-point register	R/W	-
FCCR	1	32	Floating-point control register	R/W	User 232

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Register Number ¹
FSR	1	32	Floating-point status register	R/W	User 233

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` (*Processor Control Instructions*). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.

2. 32-bits if DoublePrecision is False, 64-bits if it is True

Table 45: Floating-Point Coprocessor Option Instruction Additions

Instruction ¹	Format	Definition
ABS.S	RRR on page 656	Single-precision absolute value
ADD.S	RRR on page 656	Single-precision add
ADDEXP.S	RRR on page 656	Single-precision add exponent
ADDEXPM.S	RRR on page 656	Single-precision add exponent from mantissa segment
CEIL.S	RRR on page 656	Single-precision floating-point to signed integer conversion with round to $+\infty$
CONST.S	RRR on page 656	Single-precision create floating-point constant
DIV0.S	RRR on page 656	Single-precision IEEE Divide initial step
DIVN.S	RRR on page 656	Single-precision IEEE Divide final step
FLOAT.S	RRR on page 656	Signed integer to single-precision floating-point conversion (current rounding mode)
FLOOR.S	RRR on page 656	Single-precision floating-point to signed integer conversion with round to $-\infty$
LSI	RRR18 on page 656	Load single-precision immediate

Instruction ¹	Format	Definition
LSIP	<i>RRR</i> on page 656	Load single-precision immediate with base post-increment
LSX	<i>RRR</i> on page 656	Load single-precision indexed
LSXP	<i>RRR</i> on page 656	Load single-precision indexed with base post-increment
MADD.S	<i>RRR</i> on page 656	Single-precision multiply-add
MADDN.S	<i>RRR</i> on page 656	Single-precision multiply-add with round mode override to round-to-nearest
MKDADJ.S	<i>RRR</i> on page 656	Single-precision make divide adjust amounts
MKSADJ.S	<i>RRR</i> on page 656	Single-precision make square-root adjust amounts
MOV.S	<i>RRR</i> on page 656	Single-precision move
MOVEQZ.S	<i>RRR</i> on page 656	Single-precision move if equal to zero
MOVF.S	<i>RRR</i> on page 656	Single-precision move if Boolean condition false
MOVGEZ.S	<i>RRR</i> on page 656	Single-precision move if greater than or equal to zero
MOVLTZ.S	<i>RRR</i> on page 656	Single-precision move if less than zero
MOVNEZ.S	<i>RRR</i> on page 656	Single-precision move if not equal to zero
MOVT.S	<i>RRR</i> on page 656	Single-precision move if Boolean condition true
MSUB.S	<i>RRR</i> on page 656	Single-precision multiply-subtract
MUL.S	<i>RRR</i> on page 656	Single-precision multiply
NEG.S	<i>RRR</i> on page 656	Single-precision negate

Instruction ¹	Format	Definition
NEXP01.S	RRR on page 656	Single-precision narrow exponent
OEQ.S	RRR on page 656	Single-precision compare equal
OLE.S	RRR on page 656	Single-precision compare less than or equal
OLT.S	RRR on page 656	Single-precision compare less than
RECIP0.S	RRR on page 656	Single-precision reciprocal initial step
RFR	RRR on page 656	Read floating-point register (FR to AR)
ROUND.S	RRR on page 656	Single-precision floating-point to signed integer conversion with round to nearest
RSQRT0.S	RRR on page 656	Single-precision reciprocal square root initial step
RUR.FCR	RRR on page 656	Read floating-point control register (to AR)
RUR.FSR	RRR on page 656	Read floating-point status register (to AR)
SQRT0.S	RRR on page 656	Single-precision IEEE square root initial step
SSI	RRR18 on page 656	Store single-precision immediate
SSIP	RRR18 on page 656	Store single-precision immediate with base post-increment
SSX	RRR on page 656	Store single-precision indexed
SSXP	RRR on page 656	Store single-precision indexed with base post-increment
SUB.S	RRR on page 656	Single-precision subtract
TRUNC.S	RRR on page 656	Single-precision floating-point to signed integer conversion with round to 0

Instruction ¹	Format	Definition
UEQ.S	RRR on page 656	Single-precision compare unordered or equal
UFLOAT.S	RRR on page 656	Unsigned integer to single-precision floating-point conversion (current rounding mode)
ULE.S	RRR on page 656	Single-precision compare unordered or less than or equal
ULT.S	RRR on page 656	Single-precision compare unordered or less than
UN.S	RRR on page 656	Single-precision compare unordered
UTRUNC.S	RRR on page 656	Single-precision floating-point to unsigned integer conversion with round to 0
WFR	RRR on page 656	Write floating-point register (AR to FR)
WUR.FCR	RRR on page 656	Write floating-point control register (from AR)
WUR.FSR	RRR on page 656	Write floating-point status register (from AR)
<p>1. These instructions are fully described in Instruction Descriptions on page 321.</p>		

Table 46: Floating-Point Coprocessor Option DoublePrecision¹ Instruction Additions

Instruction ²	Format	Definition
ABS.D	RRR on page 656	Double-precision absolute value
ADD.D	RRR on page 656	Double-precision add
ADDEXP.D	RRR on page 656	Double-precision add exponent
ADDEXPM.D	RRR on page 656	Double-precision add exponent from mantissa segment

Instruction ²	Format	Definition
CEIL.D	<i>RRR</i> on page 656	Double-precision floating-point to signed integer conversion with round to $+\infty$
CONST.D	<i>RRR</i> on page 656	Double-precision create floating-point constant
CVTD.S	<i>RRR</i> on page 656	Convert single-precision to double-precision
CVTS.D	<i>RRR</i> on page 656	Convert double-precision to single-precision
DIV0.D	<i>RRR</i> on page 656	Double-precision IEEE Divide initial step
DIVN.D	<i>RRR</i> on page 656	Double-precision IEEE Divide final step
FLOAT.D	<i>RRR</i> on page 656	Signed integer to double-precision floating-point conversion (current rounding mode)
FLOOR.D	<i>RRR</i> on page 656	Double-precision floating-point to signed integer conversion with round to $-\infty$
LDI	<i>RR18</i> on page 656	Load double-precision immediate
LDIP	<i>RR18</i> on page 656	Load double-precision immediate with base post-increment
LDX	<i>RRR</i> on page 656	Load double-precision indexed
LDXP	<i>RRR</i> on page 656	Load double-precision indexed with base post-increment
MADD.D	<i>RRR</i> on page 656	Double-precision multiply-add
MADDN.D	<i>RRR</i> on page 656	Double-precision multiply-add with round mode override to round-to-nearest
MKDADJ.D	<i>RRR</i> on page 656	Double-precision make divide adjust amounts

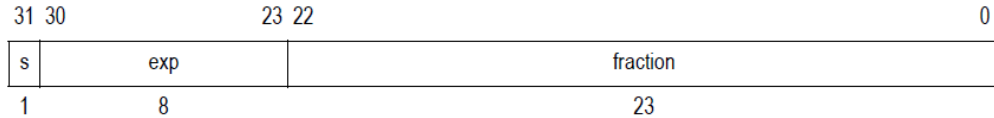
Instruction ²	Format	Definition
MKSADJ . D	<i>RRR</i> on page 656	Double-precision make square-root adjust amounts
MOV . D	<i>RRR</i> on page 656	Double-precision move
MOVEQZ . D	<i>RRR</i> on page 656	Double-precision move if equal to zero
MOVF . D	<i>RRR</i> on page 656	Double-precision move if Boolean condition false
MOVGEZ . D	<i>RRR</i> on page 656	Double-precision move if greater than or equal to zero
MOVLTZ . D	<i>RRR</i> on page 656	Double-precision move if less than zero
MOVNEZ . D	<i>RRR</i> on page 656	Double-precision move if not equal to zero
MOVT . D	<i>RRR</i> on page 656	Double-precision move if Boolean condition true
MSUB . D	<i>RRR</i> on page 656	Double-precision multiply-subtract
MUL . D	<i>RRR</i> on page 656	Double-precision multiply
NEG . D	<i>RRR</i> on page 656	Double-precision negate
NEXP01 . D	<i>RRR</i> on page 656	Double-precision narrow exponent
OEQ . D	<i>RRR</i> on page 656	Double-precision compare equal
OLE . D	<i>RRR</i> on page 656	Double-precision compare less than or equal
OLT . D	<i>RRR</i> on page 656	Double-precision compare less than
RECIP0 . D	<i>RRR</i> on page 656	Double-precision reciprocal initial step
RFRD	<i>RRR</i> on page 656	Read floating-point register upper (FR to AR)

Instruction ²	Format	Definition
ROUND . D	<i>RRR</i> on page 656	Double-precision floating-point to signed integer conversion with round to nearest
RSQRT0 . D	<i>RRR</i> on page 656	Double-precision reciprocal square root initial step
SQRT0 . D	<i>RRR</i> on page 656	Double-precision IEEE square root initial step
SDI	<i>RRR8</i> on page 656	Store double-precision immediate
SDIP	<i>RRR8</i> on page 656	Store double-precision immediate with base post-increment
SDX	<i>RRR</i> on page 656	Store double-precision indexed
SDXP	<i>RRR</i> on page 656	Store double-precision indexed with base post-increment
SUB . D	<i>RRR</i> on page 656	Double-precision subtract
TRUNC . D	<i>RRR</i> on page 656	Double-precision floating-point to signed integer conversion with round to 0
UEQ . D	<i>RRR</i> on page 656	Double-precision compare unordered or equal
UFLOAT . D	<i>RRR</i> on page 656	Unsigned integer to double-precision floating-point conversion
ULE . D	<i>RRR</i> on page 656	Double-precision compare unordered or less than or equal
ULT . D	<i>RRR</i> on page 656	Double-precision compare unordered or less than
UN . D	<i>RRR</i> on page 656	Double-precision compare unordered
UTRUNC . D	<i>RRR</i> on page 656	Double-precision floating-point to unsigned integer conversion with round to 0
WFRD	<i>RRR</i> on page 656	Write floating-point register double (AR to FR)

Instruction ²	Format	Definition
<ol style="list-style-type: none"> 1. This table contains instructions which are present only if the DoublePrecision configuration parameter is True 2. These instructions are fully described in Instruction Descriptions on page 321. 		

4.3.11.2 Floating-Point Representation

For single-precision the primary floating-point data type is IEEE754 single-precision:

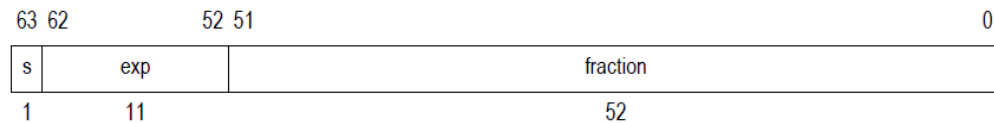


IEEE754 single-precision uses a sign-magnitude format, with a 1-bit sign, an 8-bit exponent with bias 127, and a 24-bit significand formed from 23 fraction bits representing the binary digits to the right the binary point, as shown in the diagram above, and an implicit bit to the left of the binary point (0 if exponent is zero, 1 if exponent is non-zero). Thus, the value of a normal number is:

$$(-1)^s \times 2^{\text{exp}-127} \times \text{implicit.fraction}$$

And the representation for 1.0 is 0x3F800000, with a sign of 0, exp of 127, a zero fraction, and an implicit 1 to the left of the binary point.

When the DoublePrecision parameter is True, IEEE754 double-precision is supported as well:



IEEE754 double-precision uses a sign-magnitude format, with a 1-bit sign, an 11-bit exponent with bias 1023, and a 53-bit significand formed from 52 fraction bits representing the binary digits to the right the binary point, as shown in the diagram above, and an implicit bit to the left of the binary point (0 if exponent is zero, 1 if exponent is non-zero). Thus, the value of a normal number is:

$$(-1)^s \times 2^{\text{exp}-1023} \times \text{implicit.fraction}$$

And the representation for 1.0 is 0x3FF00000_00000000, with a sign of 0, exp of 1023, a zero fraction, and an implicit 1 to the left of the binary point.

When the `DoublePrecision` parameter is `True`, it is not architecturally defined how the single-precision number is represented in the 64-bit wide data registers. Some implementations use the low 32-bits of the 64-bit register but other representations may be used. For example, another possibility might be representation as a double-precision number with proper rounding and range.

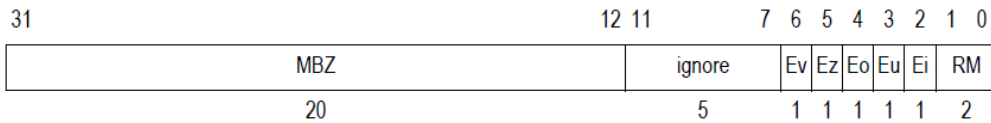
The other major data format is a signed, 32-bit integer used by the `FLOAT.D`, `FLOAT.S`, `TRUNC.D`, `TRUNC.S`, `ROUND.D`, `ROUND.S`, `FLOOR.D`, `FLOOR.S`, `CEIL.D`, and `CEIL.S` instructions. In addition, there is an unsigned, 32-bit integer format used by the `UFLOAT.D`, `UFLOAT.S`, `UTRUNC.D` and `UTRUNC.S` instructions.

The Xtensa ISA includes IEEE754 signed-zero, infinity, NaN, and denormalized numbers and processing rules implemented in hardware. Integer \leftrightarrow floating-point conversions include a binary scale factor to make conversion into and out of fixed-point formats faster.

4.3.11.3 Floating-Point State

[Table 44: Floating-Point Coprocessor Option Processor-State Additions](#) on page 99 summarizes the processor state added by the floating-point coprocessor. The FR register file consists of 16 registers and is used for all data computation. If the `DoublePrecision` parameter is `False`, each register is 32 bits wide. If it is `True`, each register is 64-bits wide. Load and store instructions transfer data between the FR's and memory.

[Table 47: FCR fields](#) on page 108 lists FCR fields and their associated meanings. Following is the format of FCR:



The RM field is used by many of the floating point operations to determine the type of rounding done. Some implementations do not have the Trap Control bits in [Table 47: FCR fields](#) on page 108. In these implementations, the fields must be written with zeros.

Table 47: FCR fields

FCR Field	Meaning
RM	Rounding mode 0 \leftarrow round to nearest 1 \leftarrow round toward 0 (TRUNC) 2 \leftarrow round toward $+\infty$ (CEIL) 3 \leftarrow round toward $-\infty$ (FLOOR)
Ev	Enable for trap on Invalid

FCR Field	Meaning
Ez	Enable for trap on Divide-by-zero
Eo	Enable for trap on Overflow
Eu	Enable for trap on Underflow
Ei	Enable for trap on Inexact
ignore	Reads as 0, ignored on write. Allows a value also containing FSR bits to be written.

The `FSR` register file provides the status flags required by IEEE754. These flags are set by any operation that raises an exceptional condition:

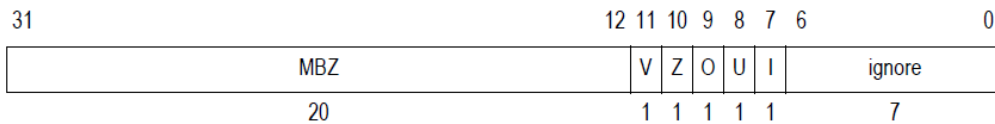


Table 48: FSR fields

FSR Field	Meaning
I	Inexact exception flag
U	Underflow exception flag ¹
O	Overflow exception flag
Z	Divide-by-zero flag
V	Invalid exception flag
MBZ	Reads as 0, Must be written with zeros
ignore	Reads as 0, ignored on write. Allows a value also containing FCR bits to be written.
<p>1. For setting the Underflow flag, Xtensa defines the IEEE-754 specification concept of "tininess" after rounding rather than before, and the IEEE-754 specification concept of "loss of accuracy" as an inexact result rather than a denormalization loss.</p>	

Xtensa's `FSR` may be read and written without waiting for the results of pending floating-point operations. Writes to `FSR` affect subsequent floating-point operations, but there is usually little

performance cost from this dependency. Only reads of `FSR` need cause a significant pipeline interlock.

`FCR` and `FSR` are organized to allow implementation with a single 32-bit physical register. The separate register numbers affect only the bits read and written of this underlying physical register. It is also possible for software to bitwise logical OR the `RUR`'s of `FCR` and `FSR` to create the appearance of a single register and to write this combined value to `FCR` and `FSR`.

4.3.11.4 Floating-Point Exceptional Conditions

The IEEE754 Specification defines five exceptional conditions¹: Invalid Operation, Divide by Zero, Overflow, Underflow², and Inexact. Implementations produce the special result values for exceptional conditions as well as setting the flags. Some implementations are capable of using the trap enables to cause a trap, though the trap may be imprecise.

4.3.11.5 Divide and Square Root Sequences

The Floating-Point Coprocessor Option provides IEEE754 Divide, IEEE754 Square Root, non-IEEE754 Reciprocal, and non-IEEE754 Reciprocal Square Root using a low gate count, reasonable speed method based on Newton-Raphson approximations. Under this method `DIV.S`, `DIV.D`, `RECIP.S`, `RECIP.D`, `RSQRT.S`, `RSQRT.D`, `SQRT.S`, and `SQRT.D` can be supported using sequences of instructions. The sequences do not include memory operations or branch operations. Higher performance versions of these may be provided by another option, but the Floating-Point Coprocessor Option itself provides a baseline version.

The sequences for `DIV.[SD]` and `SQRT.[SD]` provide IEEE754 compliant results, including status flags, in all four round modes. They begin with `DIV0.[SD]` and `SQRT0.[SD]` instructions, respectively, and continue with Newton-Raphson approximations of the reciprocal and reciprocal square root, respectively. Toward the end are instructions which guarantee that the answer is precisely right instead of being within one or two ulps (unit in the last place).

An additional issue that needs solving in the Xtensa environment, though not in environments which provide extended precision hardware, is that the single- or double-precision instructions in the sequence must not overflow the exponent range. This is accomplished by narrowing the exponent range of the arguments to divide and square root sequences so that they are in the range $1.0 \leq \arg < 4.0$. The divide or square root answer is computed for the narrow exponent range, with fully accurate mantissa, and then the exponent is adjusted at the end to account for the exponents of the original arguments. It is critical to combine the

¹ Note that the IEEE754 Specification does not use the same terminology as this ISA Book generally uses. Where the IEEE-754 Specification uses the terms Exception, Trap, and Flag, this ISA Book generally uses the terms Exceptional Condition, Exception, and Flag. In particular, when this ISA Book refers to whether Exceptions are supported, it is referring to whether the concept called Traps in the IEEE754 Specification is supported.

² Setting the Underflow Flag requires the IEEE-754 specification concepts of both "tininess" and "loss of accuracy".

exponent and final mantissa operations together so that there is not a second round of the mantissa of a denormal result. Therefore, the `DIVN. [SD]` instruction is used at the end of both divide and square root sequences. It adjusts the exponent of the result and does the computation for the final multiply-add of the sequence.

Since the `DIVN. [SD]` instruction already has a full three arguments and the exponent range of its arguments is severely reduced, the adjustment is transmitted in the upper exponent bits of two of its operands. The `MKDADJ. [SD]` and `MKSADJ. [SD]` instructions are used to create the adjustments for the divide and square root sequences, respectively, from the original argument(s) and the `ADDEXP. [SD]` and `ADDEXPM. [SD]` instructions are used in each sequence to put half of the adjustment into the upper exponent bits of the first `DIVN. [SD]` operand and the other half into the upper exponent bits of the third `DIVN. [SD]` operand. The second operand of the `DIVN. [SD]` instruction is on the critical path and does not transmit any adjustment bits.

Below are the instruction sequences for single-precision and double-precision divide:

```

DIV0.S      y0, b      ; Divide: q = a/b with recip approximation
NEXP01.S   bN, b      ; Negative and Narrow fully accurate divisor
CONST.S    e, #1     ; Prepare for next instruction
MADDN.S    e, bN, y0 ; First error computation
MOV.S      y, y0     ; Avoid overwriting
MOV.S      ex, b     ; Copy of divisor needed later
NEXP01.S   aN, a     ; Negate and narrow dividend
MADDN.S    y, e, y0  ; Second reciprocal approximation
CONST.S    e, #1     ; Prepare for first madd instruction below
CONST.S    q, #0     ; Prepare for second madd instruction below
NEG.S      r, aN     ; Positive of reduced range dividend
MADDN.S    e, bN, y  ; Second error computation
MADDN.S    q, r, y0  ; First Quotient Approximation
MKDADJ.S   ex, a     ; Make adjustment bits
MADDN.S    y, e, y   ; Third reciprocal approximation
MADDN.S    r, bN, q  ; First Quotient error
CONST.S    e, #1     ; Prepare for next instruction
MADDN.S    e, bN, y  ; Third reciprocal error
MADDN.S    q, r, y   ; Second quotient approximation
NEG.S      r, aN     ; Positive of reduced range dividend
MADDN.S    y, e, y   ; Fourth reciprocal approximation
MADDN.S    r, bN, q  ; Second Quotient error
ADDEXPM.S  q, ex     ; Include adjustment bits
ADDEXP.S   y, ex     ; Include adjustment bits
DIVN.S     q, r, y   ; Third and final quotient is accurate

DIV0.D      y, b      ; Divide: q = a/b with recip approximation
NEXP01.D   bN, b     ; Negative and Narrow fully accurate divisor
CONST.D    e0, #1    ; Prepare for next instruction
MADDN.D    e0, bN, y  ; First error computation
CONST.D    e, #0     ; Prepare for later MADDN
MOV.D      ex, b     ; Prepare for next instruction
MKDADJ.D   ex, a     ; Make divide adjust amount
MADDN.D    y, e0, y   ; Second recip approximation
MADDN.D    e, e0, e0 ; Second error computation
NEXP01.D   aN, a     ; Negate and narrow dividend
DIV0.D     y0, b     ; Repeat original divide to make y0 again
MADDN.D    y, e, y   ; Third Recip approximation
CONST.D    e, #1     ; Prepare for MADDN below

```

```

CONST.D    q, #0      ; Prepare for MADDN below
NEG.D      r, aN     ; Prepare for MADDN below
MADDN.D    e, bN, y  ; Third error computation
MADDN.D    q, r, y0  ; First quotient computation
MADDN.D    y, e, y   ; Fourth recip approximation
MADDN.D    r, bN, q  ; First Quotient error
CONST.D    e, #1     ; Prepare for next instruction
MADDN.D    e, bN, y  ; Fourth error computation
MADDN.D    q, r, y   ; Second Quotient approximation
NEG.D      r, aN     ; Prepare for MADDN below
MADDN.D    y, e, y   ; Fourth recip approximation
MADDN.D    r, bN, q  ; Second Quotient error
ADDEXPM.D  q, ex     ; Include adjustment bits
ADDEXP.D   y, ex     ; Include adjustment bits
DIVN.D     q, r, y   ; Third and final quotient is accurate

```

Below are instruction sequences for single-precision and double-precision square root:

```

SQRT0.S    y, a      ; Square Root: r = sqrt(a) approximation
CONST.S    t1, #0    ; Prepare for next instruction
MADDN.S    t1, y, y   ; Temp y*y
NEXP01.S   hN, a     ; Negative Reduced Range Argument
CONST.S    t2, #3    ; Prepare next instruction with 0.5
ADDEXP.S   hN, t2    ; Half of negative reduced range argument
MADDN.S    t2, t1, hN ; Error in first recip sqrt approx 0.5-0.5*(a*y*y)
NEXP01.S   dN, a     ; Negative Reduced Range Argument
NEG.S      h2, dN    ; Reduced Range Argument
MADDN.S    y, t2, y  ; Second Recip Square Root Approximation
CONST.S    R, #0     ; Prepare for MADDN.D below
CONST.S    t5, #0    ; Prepare for MADDN.D below
CONST.S    H, #0     ; Prepare for MADDN.D below
MADDN.S    R, h2, y2  ; Rirst Red Range Sqrt Approx
MADDN.S    t5, y2, hN ; Temp
CONST.S    t6, #3    ; Prepare for MADDN.D below
MADDN.S    H, t6, y2 ; Half of recip square root approximation
MADDN.S    dN, R, R  ; Error in first red range Sqrt Approx
MADDN.S    t6, t5, y2 ; Temp
NEG.S      HN, H     ; Neg of Half of recip square root approximation
MADDN.S    R, dN, HN ; Second Red Range Sqrt Approx
MADDN.S    H, t6, H  ;
MKSADJ.S   ex, a     ; Make Adjustment for final step
NEXP01.S   dN, a     ; Recreate Negative Reduced Range Argument
MADDN.S    dN, R, S  ; Error in second red range Sqrt Approx
NEG.S      HN, H     ;
ADDEXPM.S  R, ex     ; Include adjustment bits
ADDEXP.S   HN, ex    ; Include adjustment bits
DIVN.S     R, dN, HN ; Third and final Square Root is accurate

SQRT0.D    y, a      ; Square Root: r = sqrt(a) approximation
CONST.D    t1, #0    ; Prepare for next instruction
MADDN.D    t1, y, y   ; Temp y*y
NEXP01.D   hN, a     ; Negative reduced range argument
CONST.D    t2, #3    ; Prepare for next instruction
ADDEXP.D   hn, t2    ; Half of Negative reduced range argument
MADDN.D    t2, t1, hN ; Error in first recip sqrt approx 0.5-0.5*(a*y*y)
NEXP01.D   aN, a     ; Recreate Negative reduced range argument
MADDN.D    y, t2, y  ; Second Recip Square Root Approximation
CONST.D    t3, #0    ; Prepare for next instruction
MADDN.D    t3, y, hN ; Temp 0.5*(a*y)
CONST.D    t4, #3    ; Prepare for next instruction
MADDN.D    t4, t3, y ; Error second recip sqrt approx 0.5-0.5*(a*y*y)

```



```

NEG.D      h2, aN      ; Reduced Range Argument
MADDN.D   y, t4, y    ; Third Recip Square Root Approximation
CONST.D   R, #0      ; Prepare for MADDN.D below
CONST.D   t5, #0     ; Prepare for MADDN.D below
CONST.D   H, #0      ; Prepare for MADDN.D below
MADDN.D   R, h2, y    ; First Red Range Sqrt Approx
MADDN.D   t5, y, hN   ; Temp
CONST.D   t6, #3     ; Prepare for MADDN.D below
MADDN.D   H, t6, y    ; Half of recip square root approximation
MADDN.D   aN, R, R    ; Error in first red range Sqrt Approx
MADDN.D   t6, t5, y   ; Temp
NEG.D     HN, H       ; Neg of Half of recip square root approximation
MADDN.D   R, aN, HN   ; Second Red Range Sqrt Approx
MADDN.D   H, t6, H    ;
MKSADJ.D  ex, a       ; Make Adjustment for final step
NEXP01.D  d1, a       ; Recreate Negative Reduced Range Argument
MADDN.D   dl, R, R    ; Error in second red range Sqrt Approx
NEG.D     H2, H       ;
ADDEXPM.D R, ex       ; Include adjustment bits
ADDEXP.D  H2, ex      ; Include adjustment bits
DIVN.D    R, dl, H2   ; Third and final Square Root is accurate

```

The sequences for `RECIP.[SD]` and `RSQRT.[SD]` provide non-IEEE results that are within 1-ulp for `RECIP.[SD]` and 2-ulp for `RSQRT.[SD]` of a fully IEEE accurate result. They begin with `RECIP0.[SD]` or `RSQRT0.[SD]` and continue with just the Newton-Raphson steps. They do not have the extra instructions to provide the precisely rounded result and are therefore faster and take up less code space. One additional multiply converts these sequences into a fast divide or square root with somewhat less accuracy.

Below are the instruction sequences for single-precision and double-precision reciprocal:

```

RECIP0.S   r, b       ; Reciprocal: r = 1.0/b with recip approximation
CONST.S   e, #1      ; Prepare for following instruction
MSUB.S    e, b, r     ; Compute error in first approximation
MADD.S    r, r, e     ; Correct to get second approximation of r
CONST.S   e, #1      ; Prepare for following instruction
MSUB.S    e, b, r     ; Compute error in second approximation
MADDN.S   r, r, e     ; Correct to get third approximation of r

RECIP0.D  r, b       ; Reciprocal: r = 1.0/b with recip approximation
CONST.D   ep, #2     ; Prepare for following instruction
MSUB.D    ep, b, r   ; Compute 1.0 minus error in first approximation
MUL.D     em, b, r   ; Prepare for faster second approximation error
CONST.D   e1, #2    ; Prepare for following instruction
MUL.D     r, r, ep   ; Compute second approximation
MSUB.D    e1, em, ep ; Compute 1.0 minus error in second approximation
CONST.D   e, #1     ; Prepare for following instruction
MUL.D     r, r, e1   ; Compute third approximation
MSUB.D    e, b, r    ; Compute error in third approximation
MADDN.D   r, r, e    ; Correct to get fourth approximation of r

```

Below are instruction sequences for single-precision and double-precision reciprocal square root:

```

RSQRT0.S    r, b      ; Rsqrt: r = 1.0/sqrt(b) with rsqrt approximation
MUL.S       0, b, r   ; Temp b*r
CONST.S     h, #3    ; Half or 0.5
MUL.S       h0, h, r  ; Temp 0.5*r
CONST.S     e0, #1   ; Prepare for following instruction
MSUB.S      e0, t0, r ; Error is 1.0-(b*r*r)
MADD.S      r, h0, e0 ; Compute second rsqrt approximation
MUL.S       t1, b, r  ; Temp b*r
MUL.S       h1, h, r  ; Temp 0.5*r
CONST.S     e1, #1   ; Prepare for following instruction
MSUB.S      e1, t1, r ; Error is 1.0-(b*r*r)
MADDN.S     r, h1, e1 ; Compute third rsqrt approximation

RSQRT0.D    r, b      ; Rsqrt: r = 1.0/sqrt(b) with rsqrt approximation
MUL.D       t0, b, r  ; Temp b*r
CONST.D     h, #3    ; Half or 0.5
MUL.D       h0, h, r  ; Temp 0.5*r
CONST.D     e0, #1   ; Prepare for following instruction
MSUB.D      e0, t0, r ; Error is 1.0-(b*r*r)
MADD.D      r, h0, e0 ; Compute second rsqrt approximation
CONST.D     e1, #1   ; Prepare for MSUB.D below
MUL.D       t1, b, r  ; Temp b*r
MUL.D       h1, h, r  ; Temp 0.5*r
MSUB.D      e1, t1, r ; Error is 1.0-(b*r*r)
MADD.D      r, h1, e1 ; Compute third rsqrt approximation
CONST.D     e2, #1   ; Prepare for MSUB.D below
MUL.D       t2, b, r  ; Temp b*r
MUL.D       h2, h, r  ; Temp 0.5*r
MSUB.D      e2, t2, r ; Error is 1.0-(b*r*r)
MADDN.D     r, h2, e2 ; Compute fourth rsqrt approximation

```

All single-precision and double-precision divide and reciprocal sequences start with the following table lookup approximation:

```

255, 253, 251, 249, 247, 245, 244, 242, 240, 238, 237, 235, 233, 232, 230, 228,
227, 225, 224, 222, 221, 219, 218, 216, 215, 213, 212, 211, 209, 208, 207, 205,
204, 203, 202, 200, 199, 198, 197, 196, 194, 193, 192, 191, 190, 189, 188, 187,
186, 185, 184, 183, 182, 181, 180, 179, 178, 177, 176, 175, 174, 173, 172, 171,
170, 169, 168, 168, 167, 166, 165, 164, 163, 163, 162, 161, 160, 159, 159, 158,
157, 156, 156, 155, 154, 153, 153, 152, 151, 151, 150, 149, 149, 148, 147, 147,
146, 145, 145, 144, 143, 143, 142, 142, 141, 140, 140, 139, 139, 138, 137, 137,
136, 136, 135, 135, 134, 133, 133, 132, 132, 131, 131, 130, 130, 129, 129, 129

```

The row in the table is determined by the first three mantissa bits after the hidden bit in the divisor. If the divisor is a denormal, then it is normalized and the row in the table is determined by the first three mantissa bits after the '1' at the beginning. Which entry in the row is determined by the next four mantissa bits. The decimal number in the table is converted to an 8-bit value, which determines the first eight bits of the first reciprocal approximation, including the hidden bit. This process results in a worst case relative error of

2**-7.485. The values in the table cover the range for a single exponent starting at just over a power of two and going up to just under the next power of two.

All single-precision and double-precision square root and reciprocal square root sequences start with the following table lookup approximation:

```
180, 179, 178, 176, 175, 174, 172, 171, 170, 169, 168, 167, 166, 165, 163, 162,
161, 160, 159, 158, 158, 157, 156, 155, 154, 153, 152, 151, 151, 150, 149, 148,
147, 147, 146, 145, 144, 144, 143, 142, 142, 141, 140, 140, 139, 138, 138, 137,
137, 136, 135, 135, 134, 134, 133, 132, 132, 131, 131, 130, 130, 129, 129, 128,
255, 253, 251, 249, 247, 246, 244, 242, 241, 239, 237, 236, 234, 233, 231, 230,
228, 227, 225, 224, 223, 221, 220, 219, 218, 216, 215, 214, 213, 212, 211, 210,
208, 207, 206, 205, 204, 203, 202, 201, 200, 199, 198, 198, 197, 196, 195, 194,
193, 192, 191, 191, 190, 189, 188, 187, 187, 186, 185, 184, 184, 183, 182, 181
```

The row in the table is determined by the low bit of the biased exponent of the argument concatenated with the first two mantissa bits after the hidden bit in the argument. If the argument is a denormal, then it is normalized and the row in the table is determined by the low bit of the exponent after normalization concatenated with the first two mantissa bits after the '1' at the beginning. Which entry in the row is determined by the next four mantissa bits. The decimal number in the table is converted to an 8-bit value, which determines the first eight bits of the first reciprocal square root approximation, including the hidden bit. This process results in a worst case relative error of 2**-7.317. The values in the table cover the range for a pair of exponents starting at just over a power of two with an even biased exponent, such as 0.5 and going up to just under two powers of two higher, such as 2.0.

4.3.12 Multiprocessor Synchronization Option

When multiple processors are used in a system, some sort of communication and synchronization between processors is required. (Note that multiprocessor synchronization is distinct from pipeline synchronization between instructions as represented by the `ISYNC`, `RSYNC`, `ESYNC`, and `DSYNC` instructions, despite the name similarity). In some cases, self-synchronizing communication, such as input and output queues, is used. In other cases, a shared memory model is used for communication, and it is necessary to provide instruction-set support for synchronization because shared memory does not provide the required semantics. The Multiprocessor Synchronization Option is designed for this shared memory case.

- Prerequisites: None
- Incompatible Options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.3.12.1 Memory Access Ordering

The Xtensa ISA requires that valid programs follow a simplified version of the Release Consistency model of memory access ordering. Xtensa implementations may perform

ordinary load and store operations to non-overlapping addresses in any order. Loads and stores to overlapping addresses on a single processor must be executed in program order. This flexibility is appropriate because most memory accesses require only these semantics and some implementations may be able to execute programs significantly faster by exploiting non-program order memory access. While these semantics are appropriate for most loads and stores, order does matter when synchronizing between processors. Xtensa's Multiprocessor Synchronization Option therefore augments ordinary loads and stores with *acquire* and *release* operations, which are respectively loads and stores with more constrained memory ordering semantics relative to each other and relative to ordinary loads and stores.

The Xtensa version of Release Consistency is adapted from *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors* by Gharachorloo et. al. in the Proceedings of the 17th Annual International Symposium on Computer Architecture, 1990, from which the following three definitions are directly borrowed:

- A load by processor *i* is considered *performed with respect to processor k* at a point in time when the issuing of a store to the same address by processor *k* cannot affect the value returned by the load.
- A store by processor *i* is considered *performed with respect to processor k* at a point in time when an issued load to the same address by processor *k* returns the value defined by this store (or a subsequent store to the same location).
- An access is *performed* when it is performed with respect to all processors.

Using these definitions, Xtensa places the following requirements on memory access:

- Before an ordinary load or store access is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed, and
- Before a *release* access is allowed to perform with respect to any other processor, all previous ordinary load, store, *acquire*, and *release* accesses must be performed, and
- Before an *acquire* is allowed to perform with respect to any other processor, all previous *acquire* accesses must be performed.

Many Xtensa implementations will adopt stricter memory orderings for simplicity. However, programs should not rely on any stricter memory ordering semantics than those specified here.

4.3.12.2 Multiprocessor Synchronization Option Architectural Additions

[Multiprocessor Synchronization Option Instruction Additions](#) shows this option's architectural additions.

Table 49: Multiprocessor Synchronization Option Instruction Additions

Instruction ¹	Format	Definition
L32AI	RR18 on page 656	<p>Load 32-bit acquire (8-bit shifted offset)</p> <p>This load will perform before any subsequent loads, stores, or acquires are performed. It is typically used to test the synchronization variable protecting a critical region (for example, to acquire a lock).</p>
S32RI	RR18 on page 656	<p>Store 32-bit release (8-bit shifted offset)</p> <p>All prior loads, stores, acquires, and releases will be performed before this store is performed. It is typically used to write a synchronization variable to indicate that this processor is no longer in a critical region (for example, to release a lock).</p>
<p>1. These instructions are fully described in Instruction Descriptions on page 321.</p>		

4.3.12.3 Inter-Processor Communication with the L32AI and S32RI Instructions

L32AI and S32RI are 32-bit load and store instructions with acquire and release semantics. These instructions are useful for controlling the ordering of memory references in multiprocessor systems, where different memory locations may be used for synchronization and data, so that precise ordering between synchronization references must be maintained. Other load and store instructions may be executed by processor implementations in any order that produces the same uniprocessor result.

The MEMW instruction is somewhat similar in that it enforces load and store ordering, but is less selective. MEMW is intended for implementing C's `volatile` attribute, and not for high performance synchronization between processors.

L32AI is used to load a synchronization variable. This load will be performed before any subsequent load, store, acquire, or release is begun. This ensures that subsequent loads and stores do not see or modify data that is protected by the synchronization variable.

S32RI is used to store to a synchronization variable. This store will not begin until all previous loads, stores, acquires, or releases are performed. This ensures that any loads of the synchronization variable that see the new value will also find all protected data available as well.

Consider the following example:

```
volatile uint incount = 0;
volatile uint outcount = 0;
const uint bsize = 8;
data_t buffer[bsize];
void producer (uint n)
{
    for (uint i = 0; i < n; i += 1) {
        data_t d = newdata();           // produce next datum
        while (outcount == i - bsize); // wait for room
        buffer[i % bsize] = d;         // put data in buffer
        incount = i+1;                 // signal data is ready
    }
}
void consumer (uint n)
{
    for (uint i = 0; i < n; i += 1) {
        while (incount == i);          // wait for data
        data_t d = buffer[i % bsize]; // read next datum
        outcount = i+1;                // signal data read
        usedata (d);                   // use datum
    }
}
```

Here, `incount` and `outcount` are synchronization variables, and `buffer` is a shared data variable. `producer`'s writes to `incount` and `consumer`'s writes to `outcount` must use `S32RI` and `producer`'s reads of `outcount` and `consumer`'s reads of `incount` must use `L32AI`. If `producer`'s write to `incount` were done with a simple `S32I`, the processor or memory system might reorder the write to `buffer` after the write to `incount`, thereby allowing `consumer` to see the wrong data. Similarly, if `consumer`'s read of `incount` were done with a simple `L32I`, the processor or memory system might reorder the read to `buffer` before the read of `incount`, also causing `consumer` to see the wrong data.

4.3.13 Conditional Store Option

In addition to the memory ordering needs satisfied by the Multiprocessor Synchronization Option, a multiprocessor system can require mutual exclusion, which cannot easily be programmed using the Multiprocessor Synchronization Option. The Conditional Store Option is intended to add that capability. It does so by adding a single instruction (`S32C1I`), which atomically stores to a memory location only if its current value is the expected one. A state register (`SCOMPARE1`) is also added to provide the additional operand required. Some implementations also have a state register (`ATOMCTL`) for further control of the atomic operation in cache and on the PIF bus.

- Prerequisites: [Multiprocessor Synchronization Option](#) on page 115
- Incompatible Options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

When the atomic operation reaches the PIF bus, it causes a Read-Compare-Write (RCW) transaction on the PIF, which is different from normal reads and writes.

4.3.13.1 Conditional Store Option Architectural Additions

[Conditional Store Option Processor-State Additions](#) and [Conditional Store Option Instruction Additions](#) show this option's architectural additions.

Table 50: Conditional Store Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
SCOMPARE1	1	32	Conditional store comparison data	R/W	12
ATOMCTL ²	1	6	Atomic Operation Control	R/W	99

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` ([Processor Control Instructions](#)). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.
2. Register exists only in some implementations.

Table 51: Conditional Store Option Instruction Additions

Instruction ¹	Format	Definition
S32C1I	RR18 on page 656	Store 32-Bit compare conditional Stores to a location only if the location contains the value in the SCOMPARE1 register. The comparison of the old value and the store, if equal, is atomic.

1. These instructions are fully described in [Instruction Descriptions](#) on page 321.

4.3.13.2 Exclusive Access with the S32C1I Instruction

`L32AI` and `S32RI` allow inter-processor communication, as in the producer-consumer example in [Inter-Processor Communication with the L32AI and S32RI Instructions](#) on page 117 (barrier synchronization is another example), but they are not efficient for guaranteeing exclusive access to data (for example, locks). Some systems may provide efficient, tailored, application-specific exclusion support. When this is not appropriate, the ISA provides another general-purpose mechanism for atomic updates of memory-based synchronization variables that can be used for exclusion algorithms. The `S32C1I` instruction stores to a location if the

location contains the value in the `SCOMPARE1` register. The comparison of the old value and the conditional store are atomic. For example, an atomic increment could be done as follows:

```
loop:
  l32ai  a3, a2, 0      // current value of memory
  wsr    a3, scompare1 // put current value in SCOMPARE1
  mov    a4, a3        // save for comparison
  addi   a3, a3, 1     // increment value
  s32cli a3, a2, 0     // store new value if memory
                          // still contains SCOMPARE1
  bne    a3, a4, loop  // if value changed, try again
```

In most implementations, `S32C1I` always returns the current value of the memory location, which allows the `L32AI` instruction to be put outside the loop. In a few implementations, under some of the circumstances in which the store is not actually done, `S32C1I` can return the bitwise NOT of the `SCOMPARE1` register instead of the current memory value. In those implementations, if it is important enough to remove the load from the loop, the result can be tested. If it is not equal to the bitwise NOT of the `SCOMPARE1` register, then it is the memory value. See [S32C1I Modification](#) for more information.

Semaphores and other exclusion operations are equally simple to create using `S32C1I`.

There are many possible atomic memory primitives. `S32C1I` was chosen for the Xtensa ISA because it can easily synthesize all other primitives that operate on a single memory location. Many other primitives (for example, test and set, or fetch and add) are not as universal. Only primitives that operate on multiple memory locations are more powerful than `S32C1I`. Note that there can be subtle issues with some algorithms if between a read and an `S32C1I`, there are multiple changes to the target which bring the value back to the original one.

The `SCOMPARE1` register is undefined after reset.

4.3.13.3 Use Models for the `s32c1i` Instruction

Because of its nature as an atomic read-compare-write instruction, the `S32C1I` instruction is unusual in its relationships to local memories, caches, and system memories. Following is a list of ways that the `S32C1I` instruction is able to interact with memory. Some implementations use the `ATOMCTL` Special Register described below to control which way the instruction interacts with each memory type. Other implementations interact in a fixed way with each memory type. Refer to a specific *Xtensa Microprocessor Data Book* for more detailed information on how a specific processor handles `S32C1I` instructions.

- **Local Memory** — Xtensa processors with the Conditional Store Option configured will execute `S32C1I` instructions whose address resolves to a DataRAM address directly on that DataRAM. Unless access to the DataRAM is shared with another master, no external logic is necessary in this case. None of the other ways listed below may be used for addresses resolving to a DataRAM.

- **Exception** — Xtensa processors with the Conditional Store Option and the Exception Option 2 configured can execute the `S32C1I` instruction by taking an exception (`LoadStoreErrorCause`). The exception may be considered an error, or it may be used as a way to emulate the effect of the `S32C1I` instruction. Exception may be the only method available for certain memory types or it may be directed by the `ATOMCTL` register.
- **RCW Transaction** — Xtensa processors with the Conditional Store Option configured can execute the `S32C1I` instruction by sending an RCW transaction on the PIF bus. External logic must then implement the atomic read-compare-write on the memory location. RCW Transaction may be the only method available for certain memory types or it may be directed by the `ATOMCTL` register.

If the address of the RCW transaction targets the Inbound PIF port of another Xtensa processor, the targeted Xtensa processor has the Conditional Store Option and the Data RAM Option configured, and the RCW address targets the DataRAM, the RCW will be performed atomically on the target processor's DataRAM. No external logic other than PIF bus interconnects is necessary to allow an Xtensa processor to atomically access a DataRAM location in another Xtensa processor in this way.

- **Internal Operation** — Xtensa processors with the Conditional Store Option and the Data Cache Option configured can execute the `S32C1I` instruction by allocating and filling the line in the cache and accessing the location atomically there. No external logic is necessary in this case. Internal Operation may be the only method available for certain memory types or it may be directed by the `ATOMCTL` register.

4.3.13.4 The Atomic Operation Control Register (ATOMCTL) under the Conditional Store Option

The `ATOMCTL` register exists in some implementations of the Conditional Store Option to control how the `S32C1I` instruction interacts with the cache and with the PIF bus. Implementations without the `ATOMCTL` register allow only one behavior per memory type. [ATOMCTL Register Fields](#) shows the `ATOMCTL` register, and describes the fields. See [Use Models for the S32C1I Instruction](#) on page 120 for the meaning of the codes in the table.

31		6	5	4	3	2	1	0
	reserved	WB	WT	BY				
	24	2	2	2				

Table 52: ATOMCTL Register Fields

Field	Width (bits)	Definition
WB	2	<p><code>S32C1I</code> to Writeback Cacheable Memory (including Writeback NoAllocate Memory)</p> <p>0 → Exception - <code>LoadStoreErrorCause</code></p>

Field	Width (bits)	Definition
		1 → RCW Transaction 2 → Internal Operation 3 → Reserved
WT	2	S32C1I to Writethrough Cacheable Memory (including Cached-NoAllocate Memory) 0 → Exception - LoadStoreErrorCause 1 → RCW Transaction 2 → Internal Operation ¹ 3 → Reserved
BY	2	S32C1I to Bypass Memory 0 → Exception - LoadStoreErrorCause 1 → RCW Transaction 2 → Reserved 3 → Reserved
1. Some implementations do not implement this case and take an exception (LoadStoreErrorCause) instead.		

ATOMCTL is defined after processor reset as shown in [CPENABLE - Special Register #224](#).

An older, fixed operation, Xtensa processor which operates on all cacheable and bypass regions by RCW transaction may be emulated by setting the ATOMCTL register to 0x15. One which operates only on bypass regions by RCW transaction may be emulated by setting the ATOMCTL register to 0x01.

Bits of the ATOMCTL register are present even when they correspond to a memory type which is not configured in the Xtensa processor. For example, a processor configured without a Data Cache will still contain the fields WB and WT and those fields may contain any value. But in this case, no cacheable memory will be addressable and so it will not be possible to make use of these fields.

4.3.13.5 Memory Ordering and the s32c1i Instruction

With regard to the memory ordering defined for L32AI and S32RI in [Memory Access Ordering](#) on page 115, S32C1I plays the role of both acquire and release. That is, before the atomic pair of memory accesses can perform, all ordinary loads, stores, acquires, and releases must

have performed. In addition, before any following ordinary load, store, acquire, or release can be allowed to perform, the atomic pair of the `S32C1I` must have performed. This allows the conditional store to make atomic changes to variables with ordering requirements, such as the counts discussed in the example in [Inter-Processor Communication with the L32AI and S32RI Instructions](#) on page 117.

4.3.14 Exclusive Access Option

Mutual exclusion is also provided by the Exclusive Access Option, which provides a multiple-instruction method without any increase in interrupt latency. It does so by providing a special load instruction, a special store instruction, and an instruction to wait for the success (or non-success) of the store to appear in a state register. An ordinary branch is typically used to detect the result.

- Prerequisites: or [Exception Option 2](#) on page 126
- Incompatible Options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

The instructions added by this option have no memory ordering properties beyond ordinary loads and stores. If these are needed, additional instructions are required.

4.3.14.1 Exclusive Access Option Architectural Additions

[Exclusive Access Option Processor-State Additions](#) through [Table 55: Exclusive Access Option Instruction Additions](#) on page 124 show this option's architectural additions.

Table 53: Exclusive Access Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
ATOMCTL	1	1	Atomic Operation Control bit[8]	R/W	99
ATOMCTL	1	6	Atomic Operation Control bits[5:0] if the Region Protection Option is configured.	R/W	99

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` ([Processor Control Instructions](#)). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.

Table 54: Exclusive Access Option Constant Additions (Exception Causes)

Exception Cause	Description	Constant Value
ExclusiveErrorCause	Load Exclusive to memory region which does not understand it or exclusive operation to unaligned address. (see Exception Causes)	6'b001011 (decimal 11)

Table 55: Exclusive Access Option Instruction Additions

Instruction ¹	Format	Definition
L32EX	RRR on page 656	32-Bit Load Exclusive Loads from a location and sets a micro-architectural exclusive access mark.
S32EX	RRR on page 656	32-Bit Store Exclusive Stores to a location conditionally based on the micro-architectural exclusive access mark set by L32EX. ATOMCTL [8]. is set to indicate whether or not the conditional store was done. The previous contents of ATOMCTL [8] are saved to an AR register.
GETEX	RRR on page 656	Get Exclusive Exchanges ATOMCTL [8] and the low bit of an AR register and zeros the remaining bits of the AR register.
CLREX	RRR on page 656	Clear Exclusive Clears the micro-architectural exclusive access mark set by L32EX.
<p>1. These instructions are fully described in Instruction Descriptions on page 321.</p>		

4.3.14.2 Exclusive Access with the Exclusive Instructions

The Exclusive Instructions provide a general-purpose mechanism for atomic updates of memory-based synchronization variables that can be used for exclusion algorithms.

For example, an atomic increment could be done as follows:

```
loop:
    l32ex    a3, a2        // current value of memory
    addi    a3, a3, 1     // increment value
    s32ex    a3, a2        // store if memory unmodified
    getex    a3            // acquire store result
    beqz    a3, loop      // if unsuccessful, try again
```

In addition to loading the current value of the memory location, the `L32EX` instruction micro-architecturally marks a block of physical memory for exclusive access. The size of the block is implementation defined. The corresponding `S32EX` stores to the memory location only if no process or thread has stored to the memory location since the `L32EX` instruction was performed and records whether or not it stored in `ATOMCTL[8]`. The micro-architectural exclusive access mark tracks whether any such intervening store has occurred, and the architectural state, `ATOMCTL[8]`, indicates whether or not an atomic access succeeded.

If two `S32EX` instructions are executed without an intervening `L32EX` instruction, the second `S32EX` will fail.

The `GETEX` instruction retrieves the result indicating the success or failure of the atomic access from `ATOMCTL[8]` so that the `BEQZ` can retry the atomic access if it did not succeed. The combination of `S32EX` and `GETEX` is provided as a single instruction in many architectures. They are separated in this option to avoid increasing interrupt latency while waiting for the indication of the result of the store. The `GETEX` instruction can wait in an interruptible state for the completion of the `S32EX` instruction. The pair also saves and restores the architectural state, `ATOMCTL[8]`, so that the atomic sequence may be used without adding a save and restore.

Semaphores and other exclusion operations may also be created simply using Exclusive Instructions.

`ATOMCTL[8]` is clear after reset.

Under the Memory Protection Unit Option, the method used by the `L32EX` and `S32EX` instructions is determined by the memory type field. If the memory region is not shared, the accesses are ordinary in all respects except that they still reference the local monitor bit. If the memory region is shared but not cacheable, system bus mechanisms and global monitors are used if they are available. If the memory region is shared and cacheable, configurations including hardware coherence carry out the operations in the local cache. If the processor is not able to correctly execute these instructions, it raises the `ExclusiveErrorCause` exception.

Under the Region Protection Option, if the access is to a Bypass region and `ATOMCTL[1:0]==2'b01`, system bus mechanisms and global monitors are used if they are available. Otherwise, the `ATOMCTL` bits to use are chosen in the same way based on the memory type as under the Conditional Store Option with `2'b00` causing the

`ExclusiveErrorCause` exception to be raised and `2'b10` causing the access to be an ordinary access except that it still references the local monitor bit.

4.4 Options for Interrupts and Exceptions

The options in this section have the primary function of adding and controlling the behavior of the processor in the presence of exceptional conditions. These conditions include representatives of at least the following broad categories:

- Instruction **exceptions** are unusual situations or errors encountered in the execution of the current instruction stream.
- **Interrupts** are requests from outside the instruction stream that, if enabled, can start the processor executing a different instruction stream.
- **Machine checks** are failures of the processor hardware or related hardware that need special handling to avoid causing the overall system to fail.
- **Debug** conditions do not arise from the execution of the program or the surrounding hardware, but rather from the desire of another agent to track the execution of the processor.
- **Reset** redirects the processor from any state, usually the undefined state after power-on, and starts it on a known execution path.

There are many ways of handling these conditions ranging from ignoring the conditions or freezing the clock and asserting an output signal to multi-threaded self-handling of exceptional conditions.

The Exception Option 2 provides for the self-handling of instruction exceptions and reset. Its self-handling mechanisms for these can be extended by the Relocatable Vector Option and the Unaligned Exception Option. In addition, it provides a foundation for additional options such as the Interrupt Option, the High-Priority Interrupt Option, or the Timer Interrupt Option. Again, the Debug Option can be added to provide for hardware debugging.

4.4.1 Exception Option 2

The Exception Option 2 implements basic functions needed in the management of all types of exceptional conditions. These conditions are handled by the processor itself by redirecting execution to an exception vector to handle the condition with the possibility of returning to continue execution at the original code stream. The option only fully implements the management of a subset of exceptional conditions. Additional options providing additional exception types use the Exception Option 2 as a foundation.

- Prerequisites: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.4.1.1 Exception Option 2 Architectural Additions

[Exception Option 2 Constant Additions \(Exception Causes\)](#) through [Exception Option 2 Instruction Additions](#) show this option's architectural additions.

Table 56: Exception Option 2 Constant Additions (Exception Causes)

Exception Cause	Constant Value
IllegalInstructionCause	6'b000000 (decimal 0)
SyscallCause	6'b000001 (decimal 1)
InstructionFetchErrorCause	6'b000010 (decimal 2)
LoadStoreErrorCause	6'b000011 (decimal 3)

Table 57: Exception Option 2 Processor-Configuration Additions

Parameter	Description	Valid Values
NDEPC	Existence (number) of DEPC	0..1
ResetVector	Reset exception vector (PC of first instruction executed after reset)	32-bit address
UserExceptionVector	Vector for exceptions and level-1 interrupts when PS.EXCM = 0 and PS.UM = 1	32-bit address
KernelExceptionVector	Vector for exceptions and level-1 interrupts when PS.EXCM = 0 and PS.UM = 0	32-bit address
DoubleExceptionVector	Vector for exceptions when PS.EXCM = 1	32-bit address

Table 58: Exception Option 2 Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
EPC [1]	1	32	Exception program counter ²	R/W	177
EXCCAUSE	1	6	Cause of last exception ³	R/W	232
EXCSAVE [1]	1	32	Save location for last exception ²	R/W	209
PS.EXCM	1	4	Exception mode (see <i>PS Register Fields</i>)	R/W	230
PS.UM	1	1	User vector mode (see <i>PS Register Fields</i>)	R/W	230
EXCVADDR	1	32	Virtual address that caused last fetch, load, or store exception	R/W	238
DEPC	1	32	Double exception PC (exists if NDEPC=1)	R/W	192

1. Special Registers are accessed with *RSR*, *WSR*, and *XSR* (*Processor Control Instructions*). Processor state is listed in *Table 127: Alphabetical List of Processor State* on page 266.
2. The EPC[i] and EXCSAVE[i] registers for interrupts above level 1 are part of the High-Priority Interrupt Option (*High-Priority Interrupt Option Processor-State Additions*).
3. See *Exception Causes* for the format of this register and *Exception and Interrupt Information Registers by Vector* for which vectors have causes reported in this register.

Table 59: Exception Option 2 Instruction Additions

Instruction ¹	Format	Definition
EXCW	<i>RRR</i> on page 656	Exception wait Waits for any exceptions of previously executed instructions to occur.

Instruction ¹	Format	Definition
SYSCALL	RRR on page 656	System call Generates an exception.
RFE	RRR on page 656	Returns from the KernelExceptionVector exception.
RFDE	RRR on page 656	Returns from double exception (uses EPC if NDEPC=0)
illegal instruction	—	Illegal instruction executed
1. These instructions are fully described in Instruction Descriptions on page 321.		

4.4.1.2 Exception Causes under the Exception Option 2

A broad set of interrupts and exceptions can be handled by the processor itself under the Exception Option 2. [Instruction Exceptions under the Exception Option 2](#) through [Debug Conditions under the Exception Option 2](#) list the types of exceptional conditions other than reset that can be handled under the Exception Option 2 either natively or with the help of an additional option. In each table, the first column contains the name of the condition. The second column contains a description of the condition and the third column contains both the option required for the condition to be handled and the name of the vector to which execution will be redirected. Reset is provided by the Exception Option 2 and redirects execution to `ResetVector`.

Table 60: Instruction Exceptions under the Exception Option 2

Condition	Description	Required Option & Vector
Illegal instruction	Attempt to execute an illegal instruction or a legal instruction under illegal conditions	Exception Option 2 on page 126 General vector ¹
System call	Attempt to execute the SYSCALL instruction	Exception Option 2 on page 126 General vector ¹
Instruction fetch error	Internal physical address or a data error during instruction fetch	Exception Option 2 on page 126 General vector ¹
Load or store error	Internal physical address or data error during load or store	Exception Option 2 on page 126 General vector ¹

Condition	Description	Required Option & Vector
Unaligned data exception	Attempt to load or store data at an address which cannot be handled due to alignment	Unaligned Exception Option on page 148 General vector ¹
Privileged instruction	Attempt to execute a privileged operation without sufficient privilege	MMU Option on page 217 General vector ¹
Memory access prohibited	Attempt to access data or instructions at a prohibited address	Region Protection Option on page 196 or MMU Option on page 217 — General vector ¹
Memory privilege violation	Attempt to access data or instructions without sufficient privilege	MMU Option on page 217 General vector ¹
Address translation failure	Memory access needs translation information it does not have available	MMU Option on page 217 or Memory Protection Unit Option on page 205 General vector ¹
PIF bus error	Address or data error external to the processor on the PIF bus ²	General vector ¹
Window exception	Attempt to execute an instruction needing AR values moved between registers and stack	Windowed Register Option on page 240 WindowOverflow ³ , or WindowUnderflow ³
Alloca exception	Attempt to execute an instruction, such as MOVSP, that needs AR values moved from one stack location to another	Windowed Register Option on page 240 General vector ¹
Coprocessor disabled	Attempt to execute an instruction requiring the state of a disabled coprocessor	Coprocessor Context Option on page 149 General vector ¹
ExclusiveErrorCause	Load exclusive to memory region, which does not understand it or exclusive operation to unaligned address	Exclusive Store Option General vector ¹
<p>1. General vector means. <code>DoubleExceptionVector</code> if <code>PS.EXCM</code> is set. Otherwise it means <code>UserExceptionVector</code> if <code>PS.UM</code> is set or <code>KernelExceptionVector</code> if <code>PS.UM</code> is clear.</p>		

Condition	Description	Required Option & Vector
<p>2. Imprecise errors on writes are not included.</p> <p>3. n can take on the values 4, 8, or 12 in each of overflow and underflow making a total of 6 vectors.</p>		

Table 61: Interrupts under the Exception Option

Condition	Description	Required Option & Vector
Level-1 interrupt	Level or edge interrupt pin assertion handled as part of general vector with software check	Interrupt Option on page 151 General vector ¹
Level-1 SW interrupt	Version of level-1 interrupt caused by software using <code>WSR.INTSET</code>	Interrupt Option on page 151 General vector ¹
Medium-Level interrupt	Level/edge interrupt pin assertion handled with special interrupt level, masked on stack unusable	High-Priority Interrupt Option on page 157 InterruptVector[2..6] ²
Medium-Level SW interrupt	Version of medium level interrupt caused by software using <code>WSR.INTSET</code>	High-Priority Interrupt Option on page 157 InterruptVector[2..6] ²
High-Level interrupt	Level/edge interrupt pin assertion handled with special interrupt level, extra stack care needed	High-Priority Interrupt Option on page 157 InterruptVector[2..6] ²
High-level SW interrupt	Version of high level interrupt caused by software using <code>WSR.INTSET</code>	High-Priority Interrupt Option on page 157 InterruptVector[2..6] ²
Non-maskable interrupt	Edge triggered interrupt pin that cannot be masked by software	High-Priority Interrupt Option on page 157 InterruptVector[2..7] ²
Peripheral interrupt	Internal hardware (e.g., timers) causes one of the above interrupts without an external pin	Timer Interrupt Option on page 161 (asserts another interrupt type)
<p>1. General vector means <code>DoubleExceptionVector</code> if <code>PS.EXCM</code> is set. Otherwise it means <code>UserExceptionVector</code> if <code>PS.UM</code> is set or <code>KernelExceptionVector</code> if <code>PS.UM</code> is clear.</p>		

Condition	Description	Required Option & Vector
2. Medium and high level interrupts may use levels any level 2..6 not used for debug conditions. NMI is one level higher than the highest medium, high, or debug level.		

Table 62: Machine Checks under the Exception Option 2

Condition	Description	Required Option & Vector
ECC/parity error	An access to cache or local memory produced an ECC or parity error	<i>Memory ECC/Parity Option</i> on page 168 MemoryErrorVector

Table 63: Debug Conditions under the Exception Option 2

Condition	Description	Required Option & Vector
ICOUNT exception	An instruction would have incremented the ICOUNT register to zero.	<i>Debug Option</i> on page 256 InterruptVector[dbg] ¹
BREAK exception	Attempt to execute the BREAK or BREAK.N instruction.	<i>Debug Option</i> on page 256 InterruptVector[dbg] ¹
Instruction breakpoint	Attempt to execute an instruction matching one of the instruction breakpoint registers	<i>Debug Option</i> on page 256 InterruptVector[dbg] ¹
Data breakpoint	Attempt to load or store to a data location matching one of the data breakpoint registers.	<i>Debug Option</i> on page 256 InterruptVector[dbg] ¹
Debug interrupt	An interrupt through OCD	<i>Debug Option</i> on page 256 ² InterruptVector[dbg] ¹
1. Debug exceptions use an interrupt level provided by the High-Priority Interrupt Option. That level is labeled "dbg" in this table. 2. The debug interrupt is actually created by the OCD Option under the Debug Option.		

4.4.1.3 The Processor Status Register (PS) under the Exception Option 2

The PS register contains miscellaneous fields that are grouped together primarily so that they can be saved and restored easily for interrupts and context switching. *PS Register Format* shows its layout and *PS Register Fields* describes its fields. *Processor Status Special*

[Register](#) on page 283 describes the fields of this register in greater detail. The processor initializes these fields on processor reset: `PS.INTLEVEL` is set to 15, if it exists and `PS.EXCM` is set to 1, and the other fields are set to zero.

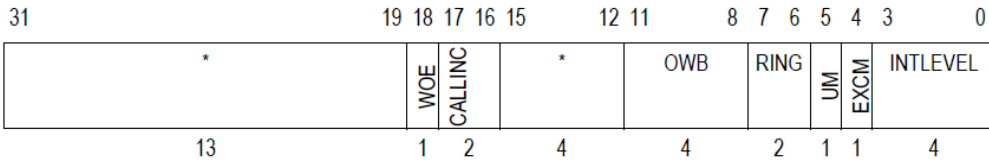


Figure 5: PS Register Format

Table 64: PS Register Fields

Field	Width (bits)	Definition [Required Option]
INTLEVEL	4	Interrupt-level disable [Interrupt Option] Used to compute the current interrupt level of the processor (Value of Variables under the Exception Option 2 on page 134).
EXCM	1	Exception mode [Exception Option 2 on page 126] 0 → normal operation 1 → exception mode Overrides the values of certain other PS fields (Value of Variables under the Exception Option 2 on page 134)
UM	1	User vector mode [Exception Option 2 on page 126] 0 → kernel vector mode — exceptions do not need to switch stacks 1 → user vector mode — exceptions need to switch stacks This bit does not affect protection. It is modified by software and affects the vector used for a general exception.
RING	2	Privilege level [MMU Option on page 217 or Memory Protection Unit Option on page 205]
OWB	4	Old window base [Windowed Register Option on page 240] The value of <code>WindowBase</code> before window overflow or underflow.
CALLINC	2	Call increment [Windowed Register Option on page 240] Set to window increment by <code>CALL</code> instructions. Used by <code>ENTRY</code> to rotate window.

Field	Width (bits)	Definition [Required Option]
WOE	1	Window overflow-detection enable [<i>Windowed Register Option</i> on page 240] 0 → overflow detection disabled 1 → overflow detection enabled Used to compute the current window overflow enable (<i>Value of Variables under the Exception Option 2</i> on page 134)
*		Reserved for future use. Writing a non-zero value to these fields results in undefined processor behavior.

4.4.1.4 Value of Variables under the Exception Option 2

The fields of the PS register listed in *PS Register Fields* affect many functions in the processor through these variables:

The current interrupt level (`CINTLEVEL`) defines which levels of interrupts are currently enabled and which are not. Interrupts at levels above `CINTLEVEL` are enabled. Those at or below `CINTLEVEL` are disabled. To enable a given interrupt, `CINTLEVEL` must be less than its level, and its `INTENABLE` bit must be 1. The level is defined by:

```
CINTLEVEL ← max (PS.EXCM*EXCMLEVEL, PS.INTLEVEL)
```

`PS.EXCM` and `PS.INTLEVEL` are part of the PS register in *PS Register Fields*. `EXCMLEVEL` is defined in *High-Priority Interrupt Option Processor-Configuration Additions*. `CINTLEVEL` is also used by the Debug Option.

The current exception mask (`CEXCM`) defines whether the exception mode is currently in effect. When it is set, certain PS register fields are overridden. It is defined by:

```
CEXCM ← PS.EXCM
```

The current ring (`CRING`) determines which ASIDs from the RASID register will cause a privilege violation. ASIDs with position (in RASID) equal to or greater than `CRING` may be used in translation while those with position less than `CRING` will cause a privilege violation. Privileged instructions may only be executed if `CRING` is zero. `CRING` is defined by:

```
CRING ← if (MMU Option or Memory Protection Unit Option configured && PS.EXCM = 0) then  
PS.RING else 0
```

`PS.EXCM` and `PS.RING` are part of the PS register in *PS Register Fields*.

The current window overflow enable (`CWOE`) defines whether window overflow exceptions are currently enabled. It is defined by:

```
CWOE ← if PS.EXCM then 0 else PS.WOE
```

`PS.EXCM` and `PS.WOE` are part of the `PS` register in *PS Register Fields*.

The current loop enable (`CLOOPENABLE`) determines whether the loop-back function of the zero-overhead loop instruction is enabled or not.

```
CLOOPENABLE ← PS.EXCM = 0
```

`PS.EXCM` is part of the `PS` register in *PS Register Fields*.

4.4.1.5 The Exception Cause Register (`EXCCAUSE`) under the Exception Option 2

After an exception that redirects execution to one of the general exception vectors (`UserExceptionVector`, `KernelExceptionVector`, or `DoubleExceptionVector`), the `EXCCAUSE` register contains a value that specifies the cause of the last exception.

EXCCAUSE Register shows the `EXCCAUSE` register. *Exception Causes* describes the 6-bit binary-value encodings for the register. `EXCCAUSE` is undefined after processor reset.

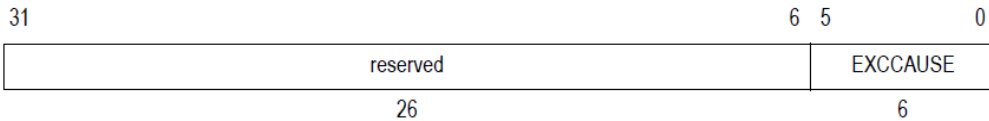


Figure 6: EXCCAUSE Register

Table 65: Exception Causes

EXC-CAUSE Code	Cause Name	Cause Description [Required Option]	EXC-VADDR Loaded
0	IllegalInstructionCause	Illegal instruction [Exception Option 2 on page 126]	No
1	SyscallCause	SYSCALL instruction [Exception Option 2 on page 126]	No
2	InstructionFetchErrorCause	Processor internal physical address or data error during instruction fetch [Exception Option 2 on page 126]	Yes

EXC-CAUSE Code	Cause Name	Cause Description [Required Option]	EXC-VADDR Loaded
3	LoadStoreErrorCause	Processor internal physical address or data error during load or store [Exception Option 2 on page 126]	Yes
4	Level1InterruptCause	Level-1 interrupt as indicated by set level-1 bits in the INTERRUPT register [Interrupt Option on page 151]	No
5	AllocaCause	MOVSP instruction, if caller's registers are not in the register file [Windowed Register Option on page 240]	No
6	IntegerDivideByZeroCause	QUOS, QUOU, REMS, or REMU divisor operand is zero [32-bit Integer Divide Option on page 90]	No
7	CSRParityErrorCause	A CSR Parity Error was detected and the Processor halted [CSR Parity Option]. This code helps the debugger to see what happened.	No
8	PrivilegedCause	Attempt to execute a privileged operation when CRING \neq 0 [MMU Option on page 217 or Memory Protection Unit Option on page 205]	No
9	LoadStoreAlignmentCause	Load or store to an unaligned address [Unaligned Exception Option on page 148]	Yes
10	ExternalRegisterPrivilegeCause	RER or WER has accessed an External Register requiring privilege but CRING \neq 0 [Exception Option 2 on page 126]	Yes

EXC-CAUSE Code	Cause Name	Cause Description [Required Option]	EXC-VADDR Loaded
11	ExclusiveErrorCause	Load Exclusive to memory region which does not understand it or exclusive operation to unaligned address [Exclusive Access Option on page 123].	Yes
12	InstrPIFDataErrorCause	PIF data error during instruction fetch	Yes
13	LoadStorePIFDataErrorCause	Synchronous PIF data error during LoadStore access	Yes
14	InstrPIFAddrErrorCause	PIF address error during instruction fetch	Yes
15	LoadStorePIFAddrErrorCause	Synchronous PIF address error during LoadStore access	Yes
16	InstTLBMissCause	Error during Instruction TLB refill [MMU Option on page 217]	Yes
17	InstTLBMultiHitCause	Multiple instruction TLB entries matched [MMU Option on page 217 or Memory Protection Unit Option on page 205]	Yes
18	InstFetchPrivilegeCause	An instruction fetch referenced a virtual address at a ring level less than CRING [MMU Option on page 217]	Yes
19		Reserved for Cadence	
20	InstFetchProhibitedCause	An instruction fetch referenced a page mapped with an attribute that does not permit instruction fetch [Region Protection Option on page 196 or MMU Option on page 217 or Memory Protection Unit Option on page 205]	Yes

EXC-CAUSE Code	Cause Name	Cause Description [Required Option]	EXC-VADDR Loaded
21..23		Reserved for Cadence	
24	LoadStoreTLBMissCause	Error during TLB refill for a load or store [MMU Option on page 217]	Yes
25	LoadStoreTLBMultiHitCause	Multiple TLB entries matched for a load or store [MMU Option on page 217 or Memory Protection Unit Option on page 205]	Yes
26	LoadStorePrivilegeCause	A load or store referenced a virtual address at a ring level less than CRING [MMU Option on page 217]	Yes
27		Reserved for Cadence	
28	LoadProhibitedCause	A load referenced a page mapped with an attribute that does not permit loads [Region Protection Option on page 196 or MMU Option on page 217 or Memory Protection Unit Option on page 205]	Yes
29	StoreProhibitedCause	A store referenced a page mapped with an attribute that does not permit stores [Region Protection Option on page 196 or MMU Option on page 217 or Memory Protection Unit Option on page 205]	Yes
30..31		Reserved for Cadence	
32..39	Coprocessor n Disabled	Coprocessor n instruction when cpn disabled. n varies 0..7 as the cause varies 32..39 [Coprocessor Context Option on page 149]	No
40..63		Reserved	

Exceptions that redirect execution to other vectors that do not use `EXCCAUSE` may either report details in a different cause register or may have only a single cause and no need for additional cause information.

4.4.1.6 The Exception Virtual Address Reg (`EXCVADDR`) under the Exception Option 2

The exception virtual address (`EXCVADDR`) register contains the virtual byte address that caused the most recent fetch, load, or store exception. [Exception Causes](#) shows, for every exception cause value, whether or not the exception virtual address register will be set. This register is undefined after processor reset. Because `EXCVADDR` may be changed by any TLB miss, even if the miss is handled entirely by processor hardware, code that counts on it not changing value must guarantee that no TLB miss is possible by using only static translations for both instruction and data accesses. [EXCVADDR Register Format](#) shows the `EXCVADDR` register format.

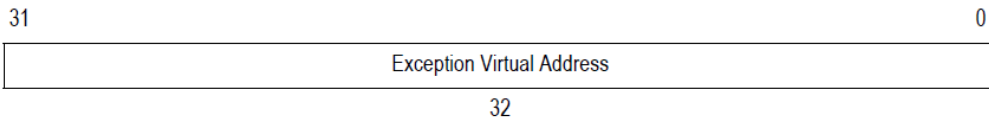


Figure 7: EXCVADDR Register Format

4.4.1.7 The Exception Program Counter (`EPC`) under the Exception Option 2

The exception program counter (EPC) register contains the virtual byte address of the instruction that caused the most recent exception or the next instruction to be executed in the case of a level-1 interrupt. This instruction has not been executed. Software may restart execution at this address by using the `RFE` instruction after fixing the cause of the exception or handling and clearing the interrupt. This register is undefined after processor reset and its value might change whenever `PS.EXCM` is 0.

The Exception Option 2 defines only one EPC value (`EPC[1]`). The High-Priority Interrupt Option extends the EPC concept by adding one EPC value per high-priority interrupt level (`EPC[2..NLEVEL+NNMI]`).

[EPC Register Format for Exception Option 2](#) shows the EPC register format.

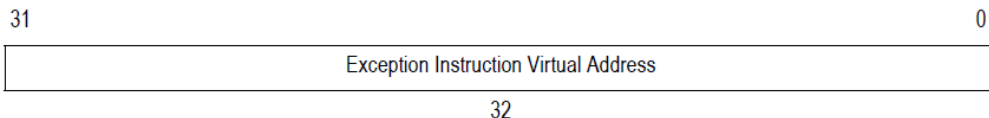


Figure 8: EPC Register Format for Exception Option 2

4.4.1.8 The Double Exception Program Counter (`DEPC`) under the Exception Option 2

The double exception program counter (DEPC) register contains the virtual byte address of the instruction that caused the most recent double exception. A double exception is one that is raised when `PS.EXCM` is set. This instruction has not been executed. Many double

exceptions cannot be restarted, but those that can may be restarted at this address by using an `RFDE` instruction after fixing the cause of the exception.

The `DEPC` register exists only if the configuration parameter `NDEPC=1`. If `DEPC` does not exist, the `EPC` register is used in its place when a double exception is taken and when the `RFDE` instruction is executed. The consequence is that it is not possible to recover from most double exceptions. `NDEPC=1` is required if both the Windowed Register Option and the MMU Option are configured. `DEPC` is undefined after processor reset.

[DEPC Register Format](#) shows the DEPC register format.

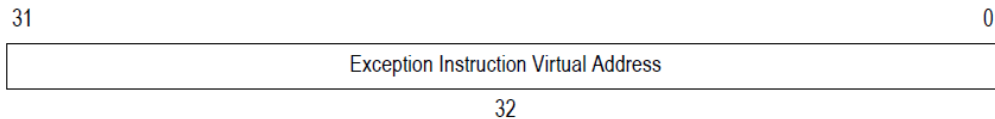


Figure 9: DEPC Register Format

4.4.1.9 The Exception Save Register (`EXCSAVE`) under the Exception Option 2

The exception save register (`EXCSAVE[1]`) is simply a read/write 32-bit register intended for saving one `AR` register in the exception vector software. This register is undefined after processor reset and there are many software reasons its value might change whenever `PS.EXCM` is 0.

The Exception Option 2 defines only one exception save register (`EXCSAVE[1]`). The High-Priority Interrupt Option extends this concept by adding one `EXCSAVE` register per high-priority interrupt level (`EXCSAVE[2..NLEVEL+NNMI]`).

[EXCSAVE Register Format](#) shows the `EXCSAVE` register format.

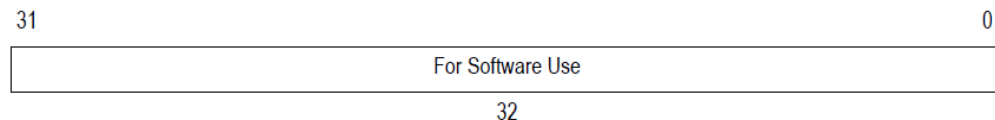


Figure 10: EXCSAVE Register Format

4.4.1.10 Handling of Exceptional Conditions under the Exception Option 2

Under the Exception Option 2, exceptional conditions are handled by saving some state and redirecting execution to one of a set of exception vector locations as listed in [Instruction Exceptions under the Exception Option 2](#) through [Debug Conditions under the Exception Option 2](#) along with `ResetVector`. This section looks at this process from the other end and describes how the code at a vector can determine the nature of the exceptional condition that has just occurred.

[Exception and Interrupt Information Registers by Vector](#) shows, for each vector, how the code can determine what has happened. The first column lists the possible vectors, not just for the Exception Option 2 itself, but also for other options that add on to the Exception Option 2. For

vectors which can be reached for more than one cause, the second column indicates the register containing the main indicator of that cause. The third column indicates other registers that may contain secondary information under that vector. The last column shows the option that is required for the vector and the other listed registers to exist.

The three exception vectors that use EXCCAUSE for the primary cause information form a set called the “general vector.” If PS.EXCM is set when one of the exceptional conditions is raised, then the processor is already handling an exceptional condition and the exception goes to the DoubleExceptionVector. Only a few double exceptions are recoverable, including a TLB miss during a register window overflow or underflow exception. For these, EXCCAUSE (and EXCSAVE in *Exception and Interrupt Exception Registers by Vector*) must be well enough understood not to need duplication. Otherwise (PS.EXCM clear), if PS.UM is set the exception goes to the UserExceptionVector, and if not the exception goes to the KernelExceptionVector. The Exception Option 2 effectively defines two operating modes: user vector mode and kernel vector mode, controlled by the PS.UM bit. The combination of user vector mode and kernel vector mode is provided so that the user vector exception handler can switch to an exception stack before processing the exception, whereas the kernel vector exception handler can continue using the kernel stack.

Single or multiple high-priority interrupts can be configured for any hardware prioritized levels 2..6. These will redirect to the InterruptVector[i] where “i” is the level. One of those levels, often the highest one, can be chosen as the debug level and will redirect execution to InterruptVector[d] where “d” is the debug level. The level one higher than the highest high-priority interrupt can be chosen as an NMI, which will redirect execution to InterruptVector[n] where “n” is the NMI level (2..7).

Table 66: Exception and Interrupt Information Registers by Vector

Vector	Main Cause	Other Information	Required Option
ResetVector	—	—	<i>Exception Option 2</i> on page 126
UserExceptionVector	EXCCAUSE	INTERRUPT, EXCVADDR	<i>Exception Option 2</i> on page 126
KernelExceptionVector	EXCCAUSE	INTERRUPT, EXCVADDR	<i>Exception Option 2</i> on page 126
DoubleExceptionVector	EXCCAUSE	EXCVADDR	<i>Exception Option 2</i> on page 126
WindowOverflow4	—	—	<i>Windowed Register Option</i> on page 240

Vector	Main Cause	Other Information	Required Option
WindowOverflow8	—	—	<i>Windowed Register Option</i> on page 240
WindowOverflow12	—	—	<i>Windowed Register Option</i> on page 240
WindowUnderflow4	—	—	<i>Windowed Register Option</i> on page 240
WindowUnderflow8	—	—	<i>Windowed Register Option</i> on page 240
WindowUnderflow12	—	—	<i>Windowed Register Option</i> on page 240
MemoryErrorVector	MESR	MECR, MEVADDR	<i>High-Priority Interrupt Option</i> on page 157
InterruptVector[i] ¹	INTERRUPT	—	<i>High-Priority Interrupt Option</i> on page 157
InterruptVector[d] ²	DEBUGCAUSE	—	<i>Debug Option</i> on page 256
InterruptVector[n] ³	—	—	<i>High-Priority Interrupt Option</i> on page 157

1. "i" indicates an arbitrary interrupt level. Medium- and high-level interrupts may be levels 2..6.
2. "d" indicates the debug level. It may be levels 2..6 but is usually the highest level other than NMI.
3. "n" indicates the NMI level. It may be levels 2..7. It must be the highest level but contiguous with other levels.

In addition to these characteristics of Vectors, when the Relocatable Vector Option is configured, the vectors are divided into two groups and within each group are required to be in increasing address order as listed below:

Static Vector Group:

- ResetVector
- MemoryErrorVector

Dynamic Vector Group:

- WindowOverflow4

- WindowUnderflow4
- WindowOverflow8
- WindowUnderflow8
- WindowOverflow12
- WindowUnderflow12
- InterruptVector[2]
- InterruptVector[3]
- InterruptVector[4]
- InterruptVector[5]
- InterruptVector[6]
- InterruptVector[7]
- KernelExceptionVector
- UserExceptionVector
- DoubleExceptionVector

Exception and Interrupt Exception Registers by Vector shows, for each vector in the first column, which registers are involved in the process of taking the exception and returning from it for that vector. Since there is no return from the `ResetVector`, it has no entries in the other four columns of this table. Otherwise all entries have a second column entry of where the PC is saved and a fifth column entry of the instruction which should be used for returning. The third column shows where the current PS register value is saved before being changed, while the fourth column shows where the handler may find a scratch register. Note that the general vector entries and the window vector entries modify the PS only in ways that their respective return instructions undo, and therefore there is no required PS save register. The window vector entries do not need scratch space because they are loading and storing a block of AR registers that they can use for scratch where they need it.

Table 67: Exception and Interrupt Exception Registers by Vector

Vector	PC	PS	Scratch	Return Instr.
<code>ResetVector</code>	—	—	—	—
<code>UserExceptionVector</code>	EPC	—	EXCSAVE	RFE
<code>KernelExceptionVector</code>	EPC	—	EXCSAVE	RFE
<code>DoubleExceptionVector</code>	DEPC	—	EXCSAVE	RFDE

Vector	PC	PS	Scratch	Return Instr.
WindowOverflow 4	EPC	—	—	RFWO
WindowOverflow 8	EPC	—	—	RFWO
WindowOverflow 12	EPC	—	—	RFWO
WindowUnderflow w4	EPC	—	—	RFWU
WindowUnderflow w8	EPC	—	—	RFWU
WindowUnderflow w12	EPC	—	—	RFWU
MemoryErrorVector	MEPC	MEPS	MESAVE	RFME
InterruptVector r[i] ¹	EPCi ¹	EPSi ¹	EXCSAVEi ¹	RFIi ¹
InterruptVector r[d] ²	EPCd ²	EPSd ²	EXCSAVED ²	RFId ²
InterruptVector r[n] ³	EPCn ³	EPSn ³	EXCSAVEn ³	RFIn ³

1. "i" indicates an arbitrary interrupt level. Medium- and high-level interrupts may be levels 2..6.
2. "d" indicates the debug level. It may be levels 2..6 but is usually the highest level other than NMI.
3. "n" indicates the NMI level. It may be levels 2..7. It must be the highest level but contiguous with other levels.

The taking of an exception under the Exception Option 2 has the following semantics:

```

procedure Exception(cause)
  if (PS.EXCM & NDEPC=1) then
    DEPC ← PC
    nextPC ← DoubleExceptionVector

```



```

elseif PS.EXCM then
    EPC[1] ← PC
    nextPC ← DoubleExceptionVector
elseif PS.UM then
    EPC[1] ← PC
    nextPC ← UserExceptionVector
else
    EPC[1] ← PC
    nextPC ← KernelExceptionVector
endif
EXCCAUSE ← cause
PS.EXCM ← 1
endprocedure Exception

```

4.4.1.11 Exception Priority under the Exception Option 2

In implementations where instruction execution is overlapped (for example, via a pipeline), multiple instructions can cause exceptions. In this case, priority is given to the exception caused by the *earliest instruction*.

When a given instruction causes multiple exceptions, the priority order for choosing the exception to be reported is listed below from highest priority to lowest. In cases where it is possible to have more than one occurrence of the same cause within the same instruction, the priority among the occurrences is undefined.

Pre-Instruction Exceptions:

- Non-maskable interrupt
- High-priority interrupt (including debug exception for `DEBUG INTERRUPT`)
- Level1InterruptCause
- Debug exception for `ICOUNT`
- Debug exception for `IBREAK`

Fetch Exceptions:

- Instruction-fetch translation errors
 - InstTLBMultiHitCause
 - InstTLBMissCause
 - InstFetchPrivilegeCause
 - InstFetchProhibitedCause
- InstructionFetchErrorCause (Instruction-fetch address or instruction data errors)
- ECC/parity exception for Instruction-fetch

Decode Exceptions:

- IllegalInstructionCause
- PrivilegedCause
- SyscallCause (`SYSCALL` instruction)
- Debug exception for `BREAK` (`BREAK`, `BREAK.N` instructions)

Execute Register Exceptions:

- Register window overflow
- Register window underflow (`RETW`, `RETW.N` instructions)
- `AllocaCause` (`MOVSP` instruction)
- `CoprocessorDisabledCause`

Execute Data Exceptions:

- Divide by Zero
- `PCValueErrorCause`

Execute Memory Exceptions:

- `LoadStoreAlignmentCause` (in the absence of the Hardware Alignment Option)
- Debug exception for `DBREAK`
- `IHI`, `PITLB`, `IPF`, or `IPFL`, or `IHU` target translation errors, in order of priority:
 - `InstTLBMultiHitCause`
 - `InstTLBMissCause`
 - `InstFetchPrivilegeCause`
 - `InstFetchProhibitedCause`
- Load, store, translation errors, in order of priority:
 - `LoadStoreTLBMultiHitCause`
 - `LoadStoreTLBMissCause`
 - `LoadStorePrivilegeCause`
 - `StoreProhibitedCause`
 - `LoadProhibitedCause`
- `InstructionFetchErrorCause` (`IPFL` target address or data errors)
- `LoadStoreAlignmentCause` (in the presence of the Hardware Alignment Option)
- `ExclusiveErrorCause`
- `LoadStoreErrorCause` (Load or store external address or data errors)
- ECC/parity exception for all accesses except instruction-fetch

Exceptions are grouped in the priority list by what information is necessary to determine whether or not the exception is to be raised. The pre-instruction exceptions may be evaluated before the instruction begins because they require nothing but the PC of the instruction. Fetch exceptions are encountered in the process of fetching the instruction. Decode exceptions may be evaluated after obtaining the instruction itself. Execute register exceptions require internal register state and execute memory exceptions involve the process of accessing the memory on which the instruction operates.

Exceptions are not necessarily precise. On some implementations, some exceptions are raised after subsequent instructions have been executed. In such implementations, the `EXCW` instruction can be used to prevent unwanted effects of imprecise exceptions. The `EXCW`

instruction causes the processor to wait until all previous instructions have taken their exceptions, if any.

Interrupts have an implicit `EXCW`; when an interrupt is taken, all instructions prior to the instruction addressed by `EPC` have been executed and any exceptions caused by those instructions have been raised. Interrupts are listed at the top of the priority list. Because the relative cycle position of an internal instruction and an interrupt pin assertion is not well-defined, the priority of interrupts with respect to exceptions is not truly well-defined either.

4.4.2 Relocatable Vector Option

This option splits Exception Vectors into two groups and adds a choice of two base addresses for one group and a Special Register as a base for the other group.

- Prerequisites: [Exception Option 2](#) on page 126
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

Under the Relocatable Vector Option, exception vectors are more restricted than they are without it. The vectors are organized into two groups, a "Static" group and a "Dynamic" group. Within each group there is a required order among the vectors which exist. The list immediately after [Exception and Interrupt Information Registers by Vector](#) indicates both the group and the order within the group. Some implementations may place an upper bound on the size of each group of vectors as measured by the difference between the address of the highest numbered vector in the group and the address of the lowest numbered vector in the group.

The Static group of vectors is not movable under software control. There are two base addresses for the Static group called the default and alternate base addresses. An input pin of the processor is sampled at reset to determine which of the two configured addresses will be used. If the `ExternalResetVector` configuration parameter is `False`, both base addresses are chosen at configuration time. If the `ExternalResetVector` configuration parameter is `True`, the default base address is chosen at configuration time and the alternate base address is taken from input pins at reset time. The base address will not change after reset. The offsets from this base are also chosen at configuration time and will not change.

The Dynamic group of vectors is movable under software control. The Special Register, `VECBASE`, described in [Table 158: VECBASE - Special Register #231](#) on page 296, holds the current base for the Dynamic group. The special register resets to a value set by the designer at configuration time but is freely writable using the `WSR.VECBASE` instruction. The offsets from the base must increase in the order indicated by [Exception and Interrupt Exception Registers by Vector](#) and are also set by the designer at configuration time.

4.4.2.1 Relocatable Vector Option Architectural Additions

[Relocatable Vector Option Processor-Configuration Additions](#) and [Relocatable Vector Option Processor-State Addition](#) shows this option's architectural additions.

Table 68: Relocatable Vector Option Processor-Configuration Additions

Parameter	Description	Valid Values
ExternalResetVector	Causes one of the selectable reset vectors to be from external pins instead of a configuration time address.	True, False

Table 69: Relocatable Vector Option Processor-State Addition

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number
VECBASE	1	32	Vector base	R/W	231

- Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.

Some implementations include bit[0] of the `VECBASE` register as a Lock bit. In these implementations, the Lock bit may be set by writing to the register but may only be cleared by reset. When it is set, nothing in the `VECBASE` register may be changed until after the Lock bit is cleared by reset. When the Lock bit is set, it is not part of the base address for vectors. In implementations which do not support the Lock bit, it is hardwired to zero.

4.4.3 Unaligned Exception Option

This option causes an exception to be raised on any unaligned memory access whether it is generated by core architecture memory instructions, by optional instructions, or by a designer's TIE instructions. With system software cooperation, occasional unaligned accesses can be handled correctly.

Cache line oriented instructions such as prefetch and cache management instructions will not raise the unaligned exception. Special instructions such as `LICW` that use a generated address for something other than an actual memory address also will not raise the exception. Individual instruction listings list the unaligned exception when it can be raised by that instruction.

Memory access instructions will raise the exception when address and size indicate it. Any address that is not a multiple of the size associated with the instruction will raise the unaligned exception whether or not the access crosses any particular size boundary. For example, an `L16UI` instruction that generates the address `32'h00000005`, will raise the unaligned exception, even though the access is entirely within a single 32-bit access.

The exception cause register will contain `LoadStoreAlignmentCause` as indicated below and the exception virtual address register will contain the virtual address of the unaligned access.

- Prerequisites: [Exception Option 2](#) on page 126

- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.4.3.1 Unaligned Exception Option Architectural Additions

[Unaligned Exception Option Constant Additions \(Exception Causes\)](#) shows this option's architectural additions.

Table 70: Unaligned Exception Option Constant Additions (Exception Causes)

Exception Cause	Description	Constant Value
LoadStoreAlignmentCause	Load or store to an unaligned address. (see Exception Causes)	6'b001001 (decimal 9)

4.4.4 Coprocessor Context Option

A coprocessor is a combination of additional state, instructions and logic that operates on that state, including moves and the setting of Booleans for branch true/false operations. The Coprocessor Context Option is general in nature: it adds state that is shared by all coprocessors. After the Coprocessor Context Option is added, specific coprocessors, such as the Floating-Point Coprocessor Option, can be constructed so that an operating system can protect the state of the coprocessor from modification using the CPENABLE register in [Coprocessor Context Option Processor-State Additions](#) below.

- Prerequisites: [Exception Option 2](#) on page 126,
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

Most options which use the Coprocessor Context Option will also work without it but the state cannot then be protected by the operating system.

4.4.4.1 Coprocessor Context Option Architectural Additions

[Coprocessor Context Option Exception Additions](#) and [Coprocessor Context Option Processor-State Additions](#) show this option's architectural additions. Under the Exception Option 2, the exception taken is listed in the last column of [Table 71: Coprocessor Context Option Exception Additions](#) on page 150.

Table 71: Coprocessor Context Option Exception Additions

Exception	Description	EXCCAUSE value
Coprocessor0Disabled	Coprocessor 0 instruction while cp0 disabled	32
Coprocessor1Disabled	Coprocessor 1 instruction while cp1 disabled	33
Coprocessor2Disabled	Coprocessor 2 instruction while cp2 disabled	34
Coprocessor3Disabled	Coprocessor 3 instruction while cp3 disabled	35
Coprocessor4Disabled	Coprocessor 4 instruction while cp4 disabled	36
Coprocessor5Disabled	Coprocessor 5 instruction while cp5 disabled	37
Coprocessor6Disabled	Coprocessor 6 instruction while cp6 disabled	38
Coprocessor7Disabled	Coprocessor 7 instruction while cp7 disabled	39

Table 72: Coprocessor Context Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
CPENABLE	1	8	Coprocessor enable bits	R/W	224

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` (*Processor Control Instructions*). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.

4.4.4.2 Coprocessor Context Switch

The TIE compiler automatically generates for each coprocessor the assembly code to save the state associated with a coprocessor to memory and to restore coprocessor state from memory.

The `CPENABLE` register allows a “lazy” context switch of the coprocessor state. Any instruction that references coprocessor *n* state (not including the shared Boolean registers) when that

coprocessor's enable bit (bit *n*) is clear raises a `CoprocessorDisabled` exception. `CPENABLE` can be cleared on context switch, and the exception used to unload the previous task's coprocessor state and load the current task's. The appropriate `CPENABLE` bit is then set by the exception handler, which then returns to execute the coprocessor instruction. An `RSYNC` instruction must be executed after writing `CPENABLE` before executing any instruction that references state controlled by the changed bits of `CPENABLE`. This register is undefined after reset.

If a single instruction references state from more than one coprocessor not enabled in `CPENABLE`, then one of `CoprocessorDisabled` exceptions is raised. The prioritization among multiple `CoprocessorDisabled` exceptions is implementation-specific.

4.4.5 Interrupt Option

The Interrupt Option implements level-1 interrupts. These are asynchronous exceptions on processor input signals or software exceptions. They have the lowest priority of all interrupts. Level-1 interrupts are handled differently than the high-priority interrupts at priority levels 2 through 6 or NMI. The Interrupt Option is a prerequisite for the High-Priority Interrupt Option, Timer Interrupt Option, and Debug Option.

Certain aspects of high-priority interrupts are specified along with those of level-1 interrupts in the Interrupt Option. Specifically, the following parameters are specified:

- `NINTERRUPT`—Total number of level-1 plus high-priority interrupts.
- `INTTYPE[0..NINTERRUPT-1]`—Interrupt type (level, edge, software, or internal) for level-1 plus high-priority interrupts.
- `INTENABLE`—Interrupt-enable mask for level-1 plus high-priority interrupts.
- `INTERRUPT`—Interrupt-request register for level-1 plus high-priority interrupts.

Nevertheless, high-priority interrupts specified in the Interrupt Option are not operational without implementation of the High-Priority Interrupt Option.

- Prerequisites: [Exception Option 2](#) on page 126
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.4.5.1 Interrupt Option Architectural Additions

[Interrupt Option Constant Additions \(Exception Causes\)](#) through [Interrupt Option Instruction Additions](#) show this option's architectural additions.

Table 73: Interrupt Option Constant Additions (Exception Causes)

Exception Cause	Description	Constant Value
<code>Level1InterruptCause</code>	Level-1 interrupt (see Exception Causes)	<code>6'b000100</code> (decimal 4)

Table 74: Interrupt Option Processor-Configuration Additions

Parameter	Description	Valid Values
NINTERRUPT	Number of level-1, high-priority, and non-maskable interrupts	1..32
INTTYPE[0..NINTERRUPT-1]	Interrupt type for level-1, high-priority, and non-maskable interrupts Specifying Interrupts on page 153	See Interrupt Types
LEVEL[0..NINTERRUPT-1]	Priority level of level-1 interrupts ¹	1
<p>1. This parameter has a fixed, implicit value. The parameter associates the level-1 interrupts with their interrupt priority (level) which, by definition, is always level 1 (lowest priority), The parameter must be explicitly specified only for the high-priority interrupts (High-Priority Interrupt Option Processor-Configuration Additions), each of which can be assigned different priority levels, from 2 to 15.</p>		

Table 75: Interrupt Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
PS.INTLEVEL	1	4	Interrupt-level disable (see PS Register Fields)	R/W	See PS Register Fields
INTENABLE	1	NINTERRUPT	Interrupt enable mask (Level-1 and high-priority interrupts) There is one bit for each level-1 and high-priority interrupt, except non-maskable interrupt (NMI) and Debug interrupt. To enable a given interrupt, CINTLEVEL (Exception Option 2 Processor-State Additions) must be less than the level assigned by LEVEL[i] to that interrupt, and the	R/W	228

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
			INTENABLE bit for that interrupt must be set to 1.		
INTERRUPT (the mnemonics INTERRUPT, INTSET, and INTCLEAR are used depending on the type of access)	1	NINTERRUPT	Interrupt request register (level-1 and high-priority interrupts) This holds pending level-1 and high-priority interrupt requests. There is 1 bit per pending interrupt, except non-maskable interrupt (NMI). If the bit is set to 1, an interrupt request is pending. External level interrupt bits are not writable.	R or R/W ²	226 for read, 226 for set, and 227 for clear
<p>1. Special Registers are accessed with <code>RSR</code>, <code>WSR</code>, and <code>XSR</code> (<i>Processor Control Instructions</i>). Processor state is listed in Table 127: Alphabetical List of Processor State on page 266.</p> <p>2. Level-sensitive interrupt bits are read-only, edge-triggered interrupt bits are read/clear, and software interrupt bits are read/write. Two register numbers are provided for software modification to the INTERRUPT register: one that sets bits, and one that clears them.</p>					

Table 76: Interrupt Option Instruction Additions

Instruction ¹	Format	Definition
RSIL	RRR on page 656	Read and set interrupt level
WAITI	RRR on page 656	Wait for interrupt
<p>1. These instructions are fully described in Instruction Descriptions on page 321.</p>		

4.4.5.2 Specifying Interrupts

Interrupt types (`INTTYPE` in [Interrupt Option Processor-Configuration Additions](#)) can be any of the values listed in [Interrupt Types](#) or an implementation defined type. The column labeled “Priority” shows the possible range of priorities for the interrupt type. The column labeled

“Pin” indicates whether there is an Xtensa core pin associated with the interrupt, while the column labeled “Bit” indicates whether or not there is a bit in the `INTERRUPT` and `INTENABLE` Special Registers corresponding to the interrupt. The last two columns indicate how the interrupt may be set and how it may be cleared.

Table 77: Interrupt Types

Type	Priority ¹	Pin?	Bit?	How Interrupt is Set	How Interrupt is Cleared
Level	1 to N	Yes	Yes	Signal level from device	At device
Edge	1 to N	Yes	Yes	Signal rising edge	<code>WSR.INTCLEAR</code> '1'
NMI	N+1	Yes	No	Signal rising edge	Automatically cleared by HW
Software	1 to N	No	Yes	<code>WSR.INTSET</code> '1'	<code>WSR.INTCLEAR</code> '1'
Timer	1 to N	No	Yes	<code>CCOUNT=CCOMPAREn</code>	<code>WSR.CCOMPAREn</code>
Debug ²	2 to N	No ²	No	Debug hardware ²	Automatically cleared by HW
WriteErr	1 to N	No	Yes	Bus error on write	<code>WSR.INTCLEAR</code> '1'
Profile	1 to N	No	Yes	Profiling Interrupt	Clear in Profiling Logic

1. Possible priorities where N is NLEVEL
2. See [Debug Option](#) on page 256 for more detail

A variable number (`NINTERRUPT`) of interrupts can be defined during processor configuration. External interrupt requests are signaled to the processor by either level-sensitive or edge-triggered inputs. Software can test these interrupt requests at any time by reading the `INTERRUPT` register. An arbitrary number of software interrupts, not tied to an external signal, can also be configured. Level-1 interrupts use either the `UserExceptionVector` or `KernelExceptionVector` defined in [Exception Option 2 Processor-Configuration Additions](#), depending on the current setting of the `PS.UM` bit.

Software can manipulate the interrupt-enable bits (INTENABLE register) and then set PS.INTLEVEL back to 0 to re-enable other interrupts, and thereby create arbitrary prioritizations. This is illustrated by the following C++ code:

```

class Interrupt {
public:
    uint32_t bit;
    void handler();
};

class Level1Interrupt {
    const uint NPRIORITY = 4;           // number of priority groupings of level1 interrupts
    struct InterruptGroup {
        uint32_t allbits;               // all INTERRUPT register bits at this priority
        uint32_t mask;                 // mask of interrupt bits at this priority and lower
        vector<Interrupt> intlist;      // list of interrupts at this priority
    } priority[NPRIORITY];
public:
    void handler();
};

// Called for all Level1 Interrupts
void
Level1Interrupt::handler ()
{
    // determine software priority of this level1 interrupt
    uint32_t interrupts = rsr(INTERRUPT);
    uint p;
    for (p = NPRIORITY-1; (interrupts & priority[p].allbits) == 0; p -= 1) {
        if (p == 0)
            return;
    }
    // found interrupts at priority p
    uint32_t save_enable = rsr(INTENABLE); // save interrupt enables
    wsr (INTENABLE, save_enable &~ priority[p].mask); // disable lower-priority ints
    // no xSYNC instruction should be necessary here because INTENABLE and
    // PS.INTLEVEL are both written and both used in the same pipe stages
    uint32_t save_ps = rsil (0); // save PS, then set level to 0
    // now higher-priority level1 interrupts are enabled
    // service all the priority p interrupts
    do {
        // first service the priority p interrupts we read earlier
        for (vector<Interrupt>::iterator i = priority[p].intlist.begin();
            i != priority[p].intlist.end(); i++) {
            if (interrupts & i->bit) {
                // interrupt i is asserted
                i->handler(); // call i's handler
                // this should clear the interrupt condition before it
            }
        }
        interrupts &= ~i->bit; // clear i's bit from request
        if ((interrupts & priority[p].allbits) == 0) // early check for done
            break;
    }
    // check if any more priority p interrupts arrived while we were servicing the previous
    batch
    interrupts = rsr(INTERRUPT);
    while ((interrupts & priority[p].allbits) == 0);
    // no more priority p interrupts
}

```

```

wsr (PS, save_ps);           // return to PS.INTLEVEL=1, disabling
                             // all level1 interrupts, before returning
wsr (INTENABLE, save_enable); // restore original enables to allow lower
                             // priority level1 interrupts
// return to general exception handler
}

```

4.4.5.3 The Level-1 Interrupt Process

With respect to level-1 interrupts, the processor takes an interrupt when any level-1 interrupt, i , satisfies:

```

INTERRUPTi and INTENABLEi and (1 > CINTLEVEL)

```

Level-1 interrupts use the `UserExceptionVector` and `KernelExceptionVector`, implemented by the Exception Option 2 ([Exception Option 2 Processor-Configuration Additions](#)). The interrupt cause is reported as `Level1InterruptCause` ([Exception Causes](#)). The interrupt handler can determine which level-1 interrupt caused the exception by doing an RSR of the `INTERRUPT` register and ANDing with the contents of the `INTENABLE` register. The exact semantics of the check for interrupts is given in [Checking for Interrupts](#) on page 160.

The process of taking an interrupt does not clear the interrupt request. The process does set `PS.EXCM` to 1, which disables level-1 interrupts in the interrupt handler. Typically, `PS.EXCM` is reset to 0 by the handler, after it has set up the stack frame and masked the interrupt. This allows other level-1 interrupts to be serviced. For level-sensitive interrupts, the handler must cause the source of the interrupt to deassert its interrupt request before re-enabling the interrupt. For edge-triggered interrupts or software interrupts, the handler clears the interrupt condition by writing to the `INTCLEAR` register.

The `WAITI` instruction sets the current interrupt level in the `PS.INTLEVEL` register. In some implementations it also powers down the processor's logic, and waits for an interrupt. After executing the interrupt handler, execution continues with the instruction following the `WAITI`.

The `INTENABLE` register and the software and edge-triggered bits of the `INTERRUPT` register are undefined after processor reset.

4.4.5.4 Use of Interrupt Instructions

The `RSIL` instruction reads the `PS` register and sets the interrupt level. It is typically used as follows:

```

RSIL      a2, newlevel
code to be executed at newlevel
WSR      a2, PS

```

A `SYNC` instruction is not required after the `RSIL`.

4.4.6 High-Priority Interrupt Option

The High-Priority Interrupt Option implements a configurable number of interrupt levels between level 2 and level 6, and an optional non-maskable interrupt (NMI) at an implicit infinite priority level. Like level-1 interrupts, high-priority interrupts are external, internal or software interrupts. Unlike level-1 interrupts, however, each high-priority interrupt level has its own interrupt vector and special registers dedicated for saving state (`EPC[level]`, `EPS[level]` and `EXCSAVE[level]`). This allows much lower latency interrupts as well as very efficient handler mechanisms. The `EPC`, `EPS` and `EXCSAVE` registers are undefined after reset.

Certain aspects of high-priority interrupts are specified along with those of level-1 interrupts in the Interrupt Option, including the total number of level-1 plus high-priority interrupts (`NINTERRUPT`), the interrupt type for level-1 plus high-priority interrupts (`INT-TYPE`), the interrupt-enable mask for level-1 plus high-priority interrupts (`INTENABLE`), and the interrupt-request register for level-1 plus high-priority interrupts (`INTERRUPT`).

- Prerequisites: [Interrupt Option](#) on page 151
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.4.6.1 High-Priority Interrupt Option Architectural Additions

[High-Priority Interrupt Option Processor-Configuration Additions](#) through [High-Priority Interrupt Option Instruction Additions](#) show this option's architectural additions.

Table 78: High-Priority Interrupt Option Processor-Configuration Additions

Parameter	Description	Valid Values
<code>NLEVEL</code>	Number of high-priority interrupt levels	2..6 ¹
<code>EXCMLEVEL</code>	Highest level masked by <code>PS.EXCM</code>	1.. <code>NLEVEL</code> ²
<code>NNMI</code>	Number of non-maskable interrupts (NMI)	0 or 1
<code>LEVEL[0..NINTERRUPT-1]</code>	Priority levels of interrupts	1.. <code>NLEVEL</code> ³
<code>InterruptVector[2..NLEVEL+NNMI]</code>	High-priority interrupt vectors	32-bit address, aligned on a 4-byte boundary
<code>LEVELMASK[1..NLEVEL-1]</code>	Interrupt-level masks	computed ⁴

1. An interrupt's "level" expresses its priority. The `NLEVEL` parameter defines the number of total interrupt levels (including level 1). Without the High-Priority Interrupt Option, `NLEVEL` is fixed at 1. With the High-Priority Interrupt Option, `NLEVEL` \geq 2.

Parameter	Description	Valid Values
	<p>2. In the presence of the Debug Option, EXCMLEVEL must be less than DEBUGLEVEL.</p> <p>3. This parameter associates interrupt levels (priorities) with interrupt numbers. level-1 interrupts, by definition, are always priority level 1 (lowest priority), and are defined in Interrupt Option Processor-Configuration Additions>. Non-maskable interrupts (NMI) have many characteristics of the level NLEVEL+1. There is no level 0.</p> <p>4. This is computed as: LEVELMASK[j]i = (LEVEL[i] = j), where j is the level specified for interrupt i, and the width of each LEVELMASK is NINTERRUPT. Thus, there are NLEVEL-1 masks (one for each high-priority interrupt level), and each mask is NINTERRUPT bits wide. A bit number set to 1 in a LEVELMASK means that the corresponding interrupt number has that priority level. The masks are used in the formal semantics to test whether an interrupt is taken on a given instruction (Checking for Interrupts on page 160).</p>	

Table 79: High-Priority Interrupt Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ⁷
EPC [2..NLEVEL +NNMI]	NLEVEL +NNMI-1	32	Exception program counter	R/W	178-183
EPS [2..NLEVEL +NNMI]	NLEVEL +NNMI-1	32	Exception processor status	R/W	194-199
EXCSAVE [2..NLEVEL +NNMI]	NLEVEL +NNMI-1	32	Save Location for high-priority interrupt handler	R/W	210-215
<p>1. Special Registers are accessed with RSR, WSR, and XSR (Processor Control Instructions). Processor state is listed in Table 127: Alphabetical List of Processor State on page 266.</p>					

Table 80: High-Priority Interrupt Option Instruction Additions

Instruction ⁷	Format	Definition
RFI	RRR on page 656	Return from high-priority interrupt
<p>1. These instructions are fully described in Instruction Descriptions on page 321.</p>		

4.4.6.2 Specifying High-Priority Interrupts

The total number of level-1 plus high-priority interrupts (`NINTERRUPT`) and the interrupt type for level-1 plus high-priority interrupts (`INTTYPE`) are specified in [Interrupt Option Processor-Configuration Additions](#). The type of each high-priority interrupt level may be edge-triggered, levelsensitive, timer, write-error, or software.

The interrupt-enable mask for level-1 plus high-priority interrupts (`INTENABLE`) and the interrupt-request register for level-1 plus high-priority interrupts (`INTERRUPT`) are specified in [Interrupt Option Processor-State Additions](#).

The total number of interrupt levels is `NLEVEL+NNMI` (see [High-Priority Interrupt Option Processor-Configuration Additions](#)). Specific interrupt numbers are assigned interrupt levels using the `LEVEL` parameter in [High-Priority Interrupt Option Processor-Configuration Additions](#). A non-maskable interrupt may be configured with the `NNMI` parameter in [High-Priority Interrupt Option Processor-Configuration Additions](#). The non-maskable interrupt signal, if implemented, will be edge-triggered. Unlike other edge-triggered interrupts, there is no need to reset the NMI interrupt by writing to `INTCLEAR`.

4.4.6.3 The High-Priority Interrupt Process

Each high-priority interrupt level has three registers used to save processor state, as shown in [High-Priority Interrupt Option Processor-State Additions](#). The processor sets `EPC[i]` and `EPS[i]` when the interrupt is taken. `EXCSAVE[i]` exists for software. The `RFI` instruction reverses the interrupt process, restoring processor state from `EPC[i]` and `EPS[i]`.

The number of high-priority interrupt levels is expected to be small, due to the cost of providing separate exception-state registers for each level. Interrupt numbers that share level 1 are not limited to a single priority, because software can manipulate the interruptenables bits (`INTENABLE` register) to create arbitrary prioritizations.

The processor takes an interrupt only when some interrupt `i` satisfies:

```
INTERRUPTi and INTENABLEi and (level[i] > CINTLEVEL)
```

where `level[i]` is the configured interrupt level of interrupt number `i`. Each level of high-priority interrupt has its own interrupt vector (`InterruptVector` in [High-Priority Interrupt Option Processor-Configuration Additions](#)). Interrupt numbers that share a level (and associated vector) can read the `INTERRUPT` register (and `INTENABLE`) with the `RSR` instruction to determine which interrupt(s) raised the exception. The non-maskable interrupt (NMI), if implemented, is taken regardless of the current interrupt level (`CINTLEVEL`) or of `INTENABLE`.

The value of `CINTLEVEL` is set to at least `EXCMLEVEL` whenever `PS.EXCM=1`. Thus, all interrupts at level `EXCMLEVEL` and below are masked during the time `PS.EXCM=1`. This is done to allow high-level language coding with the Windowed Register Option of interrupt handlers for interrupts whose level is not greater than `EXCMLEVEL`. High-priority interrupts with levels at

or below `EXCMLEVEL` are often called medium-priority interrupts. The interrupt latency is somewhat lower for levels greater than `EXCMLEVEL`, but handlers are more flexible for those whose level is not greater than `EXCMLEVEL`.

There are other conditions besides those in this section that can postpone the taking of an interrupt. For more descriptions on these, refer to a specific *Xtensa Microprocessor Data Book*.

4.4.6.4 Checking for Interrupts

The example below checks for interrupts. This is the `checkInterrupts()` procedure called in the code example shown in [Interrupt Option Special Registers](#) on page 303. The procedure itself checks for interrupts and takes the highest priority interrupt that is pending.

The `chkinterrupt()` function for non-NMI levels returns one if:

- the current interrupt level is not masking the interrupt (`CINTLEVEL < level`)
- the interrupt is asserted (`INTERRUPT`)
- the corresponding interrupt enable is set (`INTENABLE`), and
- the interrupt is of the current level (`LEVELMASK[level]`)

For NMI level interrupts, the no masking is done, but the edge sensor (made from `NMIinput` and `lastNMIinput`) is explicitly included to avoid repeating the NMI every cycle.

The `takeinterrupt()` function saves PC and PS in registers and changes them to take the interrupt.

```
procedure checkInterrupts()
    if chkinterrupt(NLEVEL+NNMI) then
        takeinterrupt[NLEVEL+NNMI]
    elseif chkinterrupt(NLEVEL+NNMI-1) then
        .
        .
        .
    elseif chkinterrupt(2) then
        takeinterrupt[2]
    elseif chkinterrupt(1) then
        Exception (Level1InterruptCause)
    endif
endprocedure checkInterrupts
```

where `chkinterrupt` and `takeinterrupt` are defined as:

```
function chkinterrupt(level)
    if level ← NLEVEL+1 and NNMI = 1 then
        chkinterrupt ← NMIinput = 1 and LastNMIinput = 0
        lastNMIinput ← NMIinput
    elseif level ≤ NLEVEL then
        chkinterrupt ← (CINTLEVEL < level) and
            ((LEVELMASK[level] and INTERRUPT and INTENABLE) ≠ 0)
```



```

else
    chkinterrupt ← 0
endif
endfunction chkinterrupt

function takeinterrupt(level)
    EPC[level] ← PC
    EPS[level] ← PS
    PC ← InterruptVector[level]
    PS.INTLEVEL ← level
    PS.EXCM ← 1
endfunction takeinterrupt

```

4.4.7 Timer Interrupt Option

The Timer Interrupt Option is an in-core peripheral option for Xtensa processors. The Timer Interrupt Option can be used to generate periodic interrupts from a 32-bit counter and up to three 32-bit comparators. One counter period typically represents a number of seconds of elapsed time, depending on the clock rate at which the processor is configured.

- Prerequisites: [Interrupt Option](#) on page 151,
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.4.7.1 Timer Interrupt Option Architectural Additions

[Timer Interrupt Option Processor-Configuration Additions](#) and [Timer Interrupt Option Processor-State Additions](#) show this option’s architectural additions.

Table 81: Timer Interrupt Option Processor-Configuration Additions

Parameter	Description	Valid Values
NCCOMPARE	Number of 32-bit comparators	0..3 ^{1,2}
TIMERINT[0..NCCOMPARE-1]	Interrupt number for each comparator	0..NINTERRUPT-1 ³

1. The comparison registers can easily be multiplexed among multiple uses, so more than one comparator is usually not useful unless each comparator uses a different `TIMERINT` interrupt level.
2. `NCCOMPARE=0` with the Timer Interrupt Option specifies that `CCOUNT` exists, but there are no `CCOMPARE` registers or interrupts.
3. Under the Exception Option 2, `NINTERRUPT` is defined in the Interrupt Option, [Interrupt Option Processor-Configuration Additions](#).

Table 82: Timer Interrupt Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ⁷
CCOUNT	1	32	Processor-clock count	R/W ²	234
CCOMPARE	NCCOMPARE	32	Processor-clock compare (CCOUNT value at which an interrupt is generated)	R/W ³	240-242-243

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` (*Processor Control Instructions*). Processor state is listed in *Table 127: Alphabetical List of Processor State* on page 266.
2. This register is not normally written except after reset; it is writable primarily for testing purposes.
3. Under Exception Option 2, writing `CCOMPARE` clears a pending interrupt.

4.4.7.2 Clock Counting and Comparison

The `CCOUNT` register increments on every processor-clock cycle.

Under the Exception Option 2, when `CCOUNT = CCOMPARE[i]`, a `TIMERINT[i]` interrupt request is generated. Although `CCOUNT` continues to increment and thus matches for only one cycle, the interrupt request is remembered until the interrupt is taken. In spite of this, timer interrupts are cleared by writing `CCOMPARE[i]`, not by writing `INTCLEAR`. Interrupt configuration determines the interrupt number and level. It is automatically an Internal interrupt type (the `INTTYPE[i]` configuration parameter, *Interrupt Option Processor-Configuration Additions*).

For most applications, only one `CCOMPARE` register is required, because it can easily be shared for multiple uses. Applications that require a greater range of counting than that provided by the 32-bit `CCOMPARE` register can maintain a 64-bit cycle count and compare the upper bits in software.

`CCOUNT` and `CCOMPARE[0..NCCOMPARE-1]` are undefined after processor reset.

4.5 Options for Local Memory

The options in this section have the primary function of adding different kinds of memory, such as RAMs, ROMs, or caches to the processor. The added memories are tightly integrated into the processor pipeline for highest performance.

4.5.1 General Cache Option Features

This subsection describes general characteristics of caches that are referred to in multiple later subsections about specific cache options.

4.5.1.1 Cache Terminology

In the cache documentation a “line” is the smallest unit of data that can be moved between the cache and other parts of the system. If the cache is “direct-mapped,” each byte of memory may be placed in only one position in the cache. In a direct-mapped cache, the “index” refers to the portion of the address that is necessary to identify the cache line containing the access.

A cache is “set-associative” if there is more than one location in the cache into which any given line may be placed. It is “N-way set-associative” if there are N locations into which any given line may be placed. The set of all locations into which one line may be placed is called a “set” and the “index” refers to the portion of the address that is necessary to identify the set containing the access. The various locations within the set that are capable of containing a line are called the “ways” of the set. And the union of the Nth way of each set of the cache is the Nth “way” of the cache.

For example, a 4-way set-associative, 16k-byte cache with a 32-byte line size contains 512 lines. There are 128 sets of 4 lines each. The index is a 7-bit value that would most likely consist of Address<11:5> and is used to determine what set contains the line. The cache consists of 4 ways, each of which is 4k-bytes in size. A set represents 128 bytes of storage made up of four lines of 32 bytes each.

4.5.1.2 Cache Tag Format

[Instruction and Data Cache Tag Format for Xtensa](#) shows the instruction- and data-cache tag format for Xtensa. The number of bits in the tag is a configuration parameter. So that all lines may be differentiated, the tag field must always be at least $32 - \log_2(\text{CacheBytes} / \text{CacheWayCount})$ bits wide. If an MMU with pages smaller than a way of the cache is used, the tag field must also be at least $32 - \log_2(\text{MinPageSize})$ bits wide. The actual tag field size is the maximum of these two values. The bits used in the tag field are the upper bits of the virtual address left justified in the register (the most significant bit of the register represents the most significant bit of the virtual address, bit 31). For example:

- A 16 kB direct-mapped cache would have an 18-bit tag field.
- A 16 kB 2-way associative cache would have a 19-bit tag field.
- A 16 kB 2-way associative cache in conjunction with an MMU with a 4kB minimum page size would have a 20-bit tag field.

The V bit is the line valid bit; 0 → invalid, 1 → valid. The flag field may contain additional bits such as:

- A Dirty bit (in a data cache) to indicate that the line is dirty
- A Lock bit to indicate that the line is locked and may not be replaced

- A bit to indicate replacement priority among ways.

See a specific *Xtensa Microprocessor Data Book* for the exact bits and their order for a particular implementation.

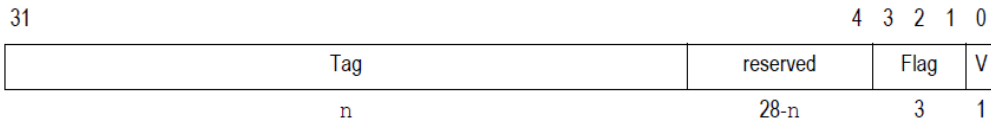


Figure 11: Instruction and Data Cache Tag Format for Xtensa

[Instruction and Data Cache Tag Address Format for Xtensa](#) shows the instruction- and data-cache tag address format for the Xtensa processor. This address format is used when accessing the Tag RAMs using the `LDCT/SDCT` and `LICT/SICT` instructions. The "Index" portion is the portion of the address that is used to identify the "set" of the cache which contains the line of interest. The lowest numbered bit of the address that is part of the index is "l" using the definitions below while the highest bit of address that is part of the index is "s-w-1" using the definitions below. This is similar to the portion of the address that is used as an Index into the Tag RAM but some cache structures may make the latter somewhat different from the "Index." The "RAM" portion is used to determine which RAM is accessed for a read. The "Way" portion is used to determine which way of the cache is used.

The widths of the various pieces use the following definitions:

- $s = \text{ceil}(\log_2(\text{number-of-bytes-in-cache}))$
- $w = \text{ceil}(\log_2(\text{number-of-ways-in-cache}))$
- $l = \log_2(\text{number-of-bytes-in-a-cache-line})$

When there is more than a single Tag RAM, `SDCT` or `SICT` causes all copies of Tag RAM to be written at the same time, regardless of the "RAM" field of the address. `LDCT` or `LICT` reads only a single RAM. Which one is given by the "RAM" field of the address.

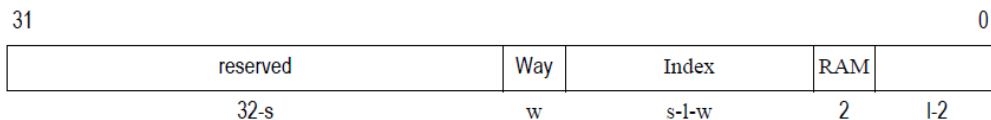


Figure 12: Instruction and Data Cache Tag Address Format for Xtensa

4.5.2 Instruction Cache Option

The Instruction Cache Option adds on-chip first-level instruction cache. The Instruction Cache Option also adds a few new instructions for prefetching and invalidation.

- Prerequisites: [Processor Interface Option](#)
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.5.3 Data Cache Option

The Data Cache Option adds on-chip first-level data cache. It supports prefetching, writing back, and invalidation.

The data-cache prefetch read/write/once instructions have been provided to improve performance, not to affect the processor state. Therefore, some implementations may choose to implement these instructions as no-op instructions. In general, the performance improvement from using these instructions is implementation-dependent. In some implementations, these instructions check whether the line containing the specified address is present in the data cache, and if not, begin the transfer of the line from memory.

- Prerequisites: [Processor Interface Option](#)
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.5.4 General RAM/ROM Option Features

The RAM and ROM options both provide internal memories that are part of the processor's address space and are accessed with the same timing as cache. These memories should not be confused with system RAM and ROM located outside of the processor, which are often larger, and may be used for both instructions and data, and shared between processors and other processing elements.

The basic configuration parameters are the size and base address of the memory. It is possible to configure cache, RAM, and ROM independently for both instruction and data, however some implementations may require an increased clock period if multiple instruction or multiple data memories are specified, or if the memory sizes are large. It is sometimes appropriate for the system designer to instead place RAMs and ROMs external to the processor and access these through the cache.

Every Instruction and Data RAM and ROM is always required to be naturally aligned (aligned on a boundary of a power of two which is equal to or larger than the size of the RAM/ROM) in physical address space. The mapping from virtual address space to physical address space must have the property that the Index bits of the RAM/ROM are identity mapped. This is a slightly less restrictive condition than requiring that the RAM/ROM must be contiguous and naturally aligned in virtual address space but this latter condition will always meet the requirement.

Instruction RAM can be referenced as data only by the `L32I`, `L32I.N`, `L32R`, `S32I` and `S32I.N` instructions and Instruction ROM referenced as data only by the `L32I`, `L32I.N` and `L32R` instructions. This functionality is available only when the [Instruction Memory Access Option](#) on page 167 is configured. In addition, this functionality is provided for initialization and test purposes, for which performance is not critical, so these operations may be significantly slower on some Xtensa implementations.

Most Xtensa code makes extensive use of L32R instructions, which load values from a location relative to the current PC. Under some circumstances, there are special requirements when L32R instructions access Instruction RAM/ROM. See [Instruction Memory Access Option](#) on page 167 below and [Performance of L32R Instruction](#) for more information on these circumstances.

[RAM/ROM Access Restrictions](#) summarizes the restrictions on instruction and data RAM and ROM access. The exceptions listed assume no memory protection exception has already been raised on the access.

Table 83: RAM/ROM Access Restrictions

Memory	Instruction Fetch	L32R L32I L32I.N	Other Loads	S32I S32I.N	Other Stores
InstROM	ok	ok or LSE ¹	undefined	LSE ³	LSE ³
InstRAM	ok	ok or LSE ¹	undefined	ok or LSE ¹	undefined
DataROM	IFE ²	ok	ok	LSE ³	LSE ³
DataRAM	IFE ²	ok	ok	ok	ok

1. ok Reduced performance on some Xtensa implementations. On implementations where the Instruction Memory Access Option is not configured, this raises a load store error exception under Exception Option 2 .
2. Instruction fetch error exception under Exception Option 2.
3. Load store error exception under Exception Option 2.

4.5.5 Instruction RAM Option

This option provides an internal, read-write instruction memory. It is typically useful as the only processor instruction store (no instruction cache) when all of the code for an application will fit in a small memory, or as an additional instruction store in parallel with the cache for code that must have constant access time for performance reasons.

- Prerequisites: None
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.5.6 Instruction ROM Option

This option provides an internal, read-only instruction memory. It is typically useful as the only processor instruction store (no instruction cache) when all of the code for an application will

fit in a small memory, or as an additional instruction store in parallel with the cache for code that must have constant access time for performance reasons. Because ROM is read-only, only code that is not subject to change should be put here.

- Prerequisites: None
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.5.7 Instruction Memory Access Option

This option allows access to the Instruction RAM and/or Instruction ROM by certain load and store instructions. Without the Instruction Memory Access Option, the contents of Instruction RAM and Instruction ROM are available inside the processor only for Instruction fetch. With the Instruction Memory Access Option, Instruction RAM may be accessed by the `L32I`, `L32I.N`, `L32R`, `S32I`, and `S32I.N` instructions. Similarly, with the Instruction Memory Access Option, Instruction ROM may be accessed by the `L32I`, `L32I.N`, and `L32R` instructions. Other load or store instructions may never access Instruction RAM or Instruction ROM.

- Prerequisites: [Instruction RAM Option](#) on page 166 or [Instruction ROM Option](#) on page 166
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

Because the instructions `L32I`, `L32I.N`, `S32I`, and `S32I.N` are intended for initialization and test purposes only, their performance may be significantly slower on many Xtensa implementations. See [Performance of L32R Instruction](#) for more on the performance of the `L32R` instruction, which is intended for loading literal values from memory.

4.5.8 Data RAM Option

This option provides an internal, read-write data memory. It is typically useful as the only processor data store (no data cache) when all of the data for an application will fit in a small memory, or as an additional data store in parallel with the cache for data that must be constant access time for performance reasons.

- Prerequisites: None
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.5.9 Data ROM Option

This option provides an internal, read-only data memory. It is typically useful as an additional data store in parallel with the cache for data that must be constant access time for performance reasons.

- Prerequisites: None
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

4.6 Hardware Alignment Option

The Hardware Alignment Option adds hardware to the processor which allows loads and stores to work correctly at any arbitrary alignment. It does this by making multiple accesses where necessary and combining the results. Unaligned accesses are still slower than aligned accesses, but this option is more efficient than the Unaligned Exception Option with software handler. In addition, the Hardware Alignment Option will work in situations where a software handler is difficult to write (for example, a load and operate instruction).

- Prerequisites: [Unaligned Exception Option](#) on page 148
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

The Hardware Alignment Option builds on the Unaligned Exception Option so that almost all potential `LoadStoreAlignmentCause` exceptions are handled transparently by hardware instead. A few situations, which are never expected to happen in real software, still raise a `LoadStoreAlignmentCause` exception. In order to properly handle all TLB misses and other exceptions, the priority of the `LoadStoreAlignmentCause` exception is lower when the Hardware Alignment Option is present than when it is not. Exception priorities are listed in [Exception Priority under the Exception Option 2](#) on page 145.

A `LoadStoreAlignmentCause` exception may still be raised in some implementations with the Hardware Alignment Option if the address of a load or store instruction is not a multiple of its size and any of the following conditions is also true:

- The instruction is one of `L32AI`, `S32RI`, or `S32CI`.
- The memory type for either portion is `XLMI`, `IRAM`, or `IROM`.
- The memory types (cache, `DataRAM`, bypass) of the two portions differ.
- The cache attribute for either portion is `Isolate`.
- The column labeled "Meaning for Cache Access" in either [Region Protection Option Attribute Field Values](#) or [MMU Option Attribute Field Values](#) is different for the two portions of the access.

4.7 Memory ECC/Parity Option

The Memory ECC/Parity Option allows the local memories and caches of Xtensa processors to be protected against errors by either parity or error correcting code (ECC). It does not affect the processor interface and system memories must maintain their own error detection

and correction. Local memories must be wide enough to contain the additional bits required. The generation and checking of parity or ECC is done in the Xtensa core through a combination of hardware and software mechanisms.

- Prerequisites: [Exception Option 2](#) on page 126
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

Each memory may be protected or not protected individually. All protected instruction memories must use a single protection type (parity or ECC). Likewise, all protected data memories must use a single protection type. For parity protection, data memories require one additional bit per byte while instruction memories require one additional bit per four bytes and cache tags require one additional bit per tag. For ECC protection, instruction memories require 7 additional bits per 32-bit word, data memories require either 5 additional bits per byte or 7 additional bits per 32-bit word, and cache tags require 7 additional bits per tag.

The core computes parity or ECC bits on every store without doing a read-modify-write except when data memories have 7 additional bits per 32-bit word and stores that affect part of a 32-bit word must do a read-modify-write. On every load or instruction fetch, these bits are checked and an exception is raised for parity errors or for uncorrectable ECC errors. For correctable errors, a control bit in the memory error status register ([Memory ECC/Parity Option Processor-State Additions](#)) indicates whether to raise an exception or simply correct the value to be used (but not the value in memory) and continue. In addition, correctable ECC errors assert an output pin which may be used as an interrupt. Implementations may or may not implement hardware correction. If they do not implement it, the exception is always raised.

Some implementations include the address bits used to index into the memory in the ECC calculation. This does not change any operation except that there are additional ECC syndromes that indicate address errors instead of data errors. This addition allows the detection of transient errors in the driving of the address in addition to other errors.

4.7.1 Memory ECC/Parity Option Architectural Additions

[Memory ECC/Parity Option Processor-Configuration Additions](#) through [Memory ECC/Parity Option Instruction Additions](#) show this option's architectural additions. It is recommended that the memory error handler be placed in memory which is also unlikely to have a memory error. For example, this might be ROM or uncacheable memory that is ECC corrected on the fly when accessed.

Table 84: Memory ECC/Parity Option Processor-Configuration Additions

Parameter	Description	Valid Values
MemoryErrorVector	Exception vector for memory errors	32-bit address

Parameter	Description	Valid Values
	Each RAM/Cache has configuration additions valid when the Memory ECC/Parity Option is configured	

Table 85: Memory ECC/Parity Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
MEPC	1	32	Memory error PC register	R/W	106
MEPS	1	32	Memory error PS register	R/W	107
MESAVE	1	32	Memory error save register	R/W	108
MESR	1	19	Memory error status register	R/W	109
MECR	1	22	Memory error check register	R/W	110
MEVADDR	1	32	Memory error virtual address register	R/W	111

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` (*Processor Control Instructions*). Processor state is listed in *Table 127: Alphabetical List of Processor State* on page 266.

Table 86: Memory ECC/Parity Option Instruction Additions

Instruction ¹	Format	Definition
R _F ME	<i>RRR</i> on page 656	Return from memory error

1. These instructions are fully described in *Instruction Descriptions* on page 321.

4.7.2 Memory Error Information Registers

Three registers are used to maintain information about a memory error. They are updated for memory errors which do not raise an exception, as well as those which do. The memory error status register (`MESR`), shown in *MESR Register Format* with further description in *MEVADDR*

Contents, contains control bits that control the operation of memory errors and status bits that hold information about memory errors that have occurred.

Under normal operation, check bits are always calculated and written to local memories. When ECC is enabled, an uncorrectable error, or a correctable error for which the `MESR.DataExc` or `MESR.InstExc` bit is set, will raise an exception whenever it is encountered during either a load or a dirty castout. Inbound PIF operations return an error when appropriate but the error will not be noted by the local processor. Correctable errors during a dirty castout when `MESR.DataExc` is clear may, in some implementations, correct the error on the fly without setting `MESR.RCE` or associated status.

When ECC is enabled and either the `MESR.DataExc` bit or the `MESR.InstExc` bit is clear or the `MESR.MemE` bit is set, hardware may be able to correct an error without raising an exception. This may cause `MESR.RCE` (along with many other fields), `MESR.DLCE`, or `MESR.ILCE` to be set by hardware at an arbitrary time.

In addition, an external pin reflects the state of `MESR.RCE` and can be connected to an interrupt input on the Xtensa processor itself or on another processor. This interrupt may be at a much lower priority than the memory error exception handler, but it can still repair the memory itself and/or log the error much as the memory error exception handler might. `MESR.RCE` must be cleared by software to return the external pin to zero and to re-arm the mechanism for recording correctable errors.

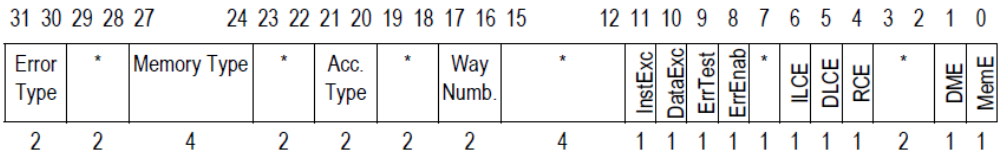


Figure 13: MESR Register Format

Table 87: MESR Register Fields

Field	Width (bits)	Definition
MemE	1	Memory error. 0 → Memory error exception not in progress. 1 → Memory error exception in progress. Set on taking memory error exception. Cleared by <code>RFME</code> instruction. Software reads and writes <code>MemE</code> normally.
DME	1	Double Memory error.

Field	Width (bits)	Definition
		<p>0 → Normal memory error exception.</p> <p>1 → Current memory error exception encountered during a Memory error exception.</p> <p>Set on taking memory error exception while MemE is set. Hardware does not clear. Software reads and writes DME normally.</p>
RCE (ECC ¹)	1	<p>Recorded correctable error. (Exists only if ECC is configured.)</p> <p>0 → Status refers to something else.</p> <p>1 → Status refers to an error corrected by hardware.</p> <p>RCE means that status refers to a correctable memory error that has been fixed in hardware. Status, here, means the group of state that contains information about a memory error. It consists of the status fields of MESR (Way Number, Access Type, Memory Type, and Error Type) and the contents of the MECCR and MEVADDR registers. The recorded information may be used to fix the error in the memory copy or to log the error.</p> <p>RCE is set by hardware whenever MemE is clear, RCE is clear, and a correctable error is fixed in hardware. RCE is cleared by hardware when a memory exception is raised as the recorded information is lost and either DLCE or ILCE is set in its place. Software reads and writes RCE normally.</p>
DLCE (ECC ¹)	1	<p>Data lost correctable error. (Exists only if ECC is configured.)</p> <p>0 → No information has been lost about data hardware corrected memory errors.</p>

Field	Width (bits)	Definition
		<p>1 → Information has been lost about data hardware corrected memory errors.</p> <p>DLCE means that there has been a correctable error on a data (execute) access which has not been recorded because 1) it happened during a memory error exception (MemE set), 2) a memory error exception happened before it was recorded (RCE now cleared), or 3) it happened after another correctable error and before that error was recorded (RCE also set).</p> <p>DLCE is set by hardware whenever any data (execute) correctable error is fixed in hardware but MemE or RCE is set and the new Access Type is not instruction fetch. DLCE is also set by hardware when any memory exception is raised with RCE set and with the current Access Type is not instruction fetch. DLCE is never cleared by hardware. Software reads and writes DLCE normally.</p>
ILCE (ECC ¹)	1	<p>Instruction fetch (Ifetch) lost correctable error. (Exists only if ECC is configured.)</p> <p>0 → No information has been lost about ifetch hardware corrected memory errors.</p> <p>1 → Information has been lost about ifetch hardware corrected memory errors.</p> <p>ILCE means that there has been a correctable error on an Ifetch access which has not been recorded because 1) it happened during a memory error exception (MemE set), 2) a memory error exception happened before it was recorded (RCE now cleared), or 3) it happened after another correctable error and before that error was recorded (RCE also set).</p>

Field	Width (bits)	Definition
		<p>ILCE is set by hardware whenever any lfetch correctable error is fixed in hardware but MemE or RCE is set and the new Access Type is instruction fetch. ILCE is also set by hardware when any memory exception is raised with RCE set and with the current Access Type is instruction fetch. ILCE is never cleared by hardware. Software reads and writes ILCE normally.</p>
ErrEnab	1	<p>Enable Memory ECC/Parity Option errors.</p> <p>0 → Memory errors are disabled. 1 → Memory errors are enabled.</p> <p>When ErrEnab is set, memory error exceptions and corrections are enabled. When ErrEnab is clear, the same values are written to memories, but no checks and no exceptions are raised on memory reads. Operation is undefined when both ErrEnab and ErrTest are set. ErrEnab is not modified by hardware.</p>
ErrTest	1	<p>Memory error test mode.</p> <p>0 → Normal memory error operation. 1 → Special memory error test operation.</p> <p>When the ErrTest bit of the MESR register is set, certain load instructions write the actual check bits which have been read from memory into the fields of the MECR register. The load instructions which do this are L32I/L32I.N from local memory, L32I/L32I.N from an isolate mode address, and every LICT, LICW, LDCT, or LDCW instruction. Similarly, when the ErrTest bit of the MESR register is set, certain store instructions use the fields of the MECR register as</p>

Field	Width (bits)	Definition
		<p>the source of check bits to write. The store instructions which do this are S32I/S32I.N to local memory, S32I/S32I.N to an isolate mode address, and every SICT, SICW, SDCT, or SDCW instruction.</p> <p>Operation of other memory access instructions and cache fills/castouts is not defined when ErrTest is set. Slave Port accesses to local memories are unaffected by the setting of the ErrTest bit. When ErrTest is clear, writes to memory compute appropriate check bits for each write, and reads from memory do not affect the MECR register (unless a memory error is detected). Operation is not defined if both ErrEnab and ErrTest are set. ErrTest is not modified by hardware.</p>
DataExc (ECC ¹)	1	<p>Data exception. (Exists only if ECC is configured.)</p> <p>0 → No exception on hardware correctable data memory errors.</p> <p>1 → Memory error exception on hardware correctable data memory errors.</p> <p>Set by software to cause memory errors which might be handled in hardware on data accesses to raise the memory error exception instead. This bit is forced to 1 (cannot be cleared) if hardware is unable to handle any data access errors. If MemE is set, no exception is raised for errors which hardware can handle even if DataExc is set. DataExc is not modified by hardware.</p>
InstExc (ECC ¹)	1	<p>Instruction exception. (Exists only if ECC is configured.)</p> <p>0 → No exception on hardware correctable instruction fetch memory errors.</p>

Field	Width (bits)	Definition
		<p>1 → Memory error exception on all instr. fetch memory errors.</p> <p>Set by software to cause memory errors which might be handled in hardware on instruction fetches to raise the memory error exception instead. This bit is forced to 1 (cannot be cleared) if hardware implementation will not handle any instruction fetch errors or if hardware is unable to detect any instruction fetch errors. If MemE is set, no exception is raised for errors which hardware can handle even if InstExc is set. InstExc is not modified by hardware.</p>
Way Number	2	<p>Cache way number of a memory error. (Exists only if a multiway cache is configured.)</p> <p>When RCE or MemE is set and the Memory Type field points to a cache, this field contains the cache way number containing the error.</p> <p>Way Number is set by hardware whenever MemE is clear, RCE is clear, and a correctable error is fixed in hardware or whenever a memory exception is raised.</p>
Access Type	2	<p>Access type of an access with memory error.</p> <p>0 → Memory error during load or store</p> <p>1 → Memory error during instruction fetch</p> <p>2 → Memory error during instruction memory access (such as IPFL or IHI)</p> <p>3 → Memory error during dirty line castout</p> <p>When RCE or MemE is set, this field contains an indication of the access type which caused the memory error.</p>

Field	Width (bits)	Definition
		<p><code>Access Type</code> is set by hardware whenever <code>MemE</code> is clear, <code>RCE</code> is clear, and a correctable error is fixed in hardware or whenever a memory exception is raised.</p>
Memory Type	4	<p>Memory type to which the access with memory error was directed.</p> <p>0 → Error in instruction RAM 0. 1 → Error in data RAM 0. 2 → Error in instruction cache data array. 3 → Error in data cache data array 4 → Error in instruction RAM 1. 5 → Error in data RAM 1. 6 → Error in Instruction cache tag array. 7 → Error in data cache tag array 8-15 → Reserved</p> <p>When <code>RCE</code> or <code>MemE</code> is set, this field contains a pointer to the memory which caused the memory error.</p> <p><code>Memory Type</code> is set by hardware whenever <code>MemE</code> is clear, <code>RCE</code> is clear, and a correctable error is fixed in hardware or whenever a memory exception is raised.</p>
Error Type	2	<p>Type of memory error.</p> <p>0 → Reserved 1 → Parity error 2 → Correctable ECC error 3 → Uncorrectable ECC error</p> <p>When <code>RCE</code> or <code>MemE</code> is set, this field contains an indicator of the type of memory error which caused the memory error.</p> <p><code>Error Type</code> is set by hardware whenever <code>MemE</code> is clear, <code>RCE</code> is clear, and a correctable error is fixed</p>

Field	Width (bits)	Definition
		in hardware or whenever a memory exception is raised.
*		Reserved for future use Writing a non-zero value to one of these fields results in undefined processor behavior. These bits read as undefined.
<p>1. In some implementations the bits used with ECC may exist as state bits without effect even when only parity is configured.</p>		

The memory error check register ($MECR$), shown in *MECR Register Format* with further description in *MECR Register Fields*, contains syndrome bits that indicate what error occurred. For data memories with 8-bit ECC, all four check fields are used, so that all bytes may be covered. For instruction memories, for cache tags, or for data memories with 32-bit ECC, only the Check 0 field is used.

When the `ErrEnab` bit of the `MESR` register is set and the `RCE` or `MemE` bit of the `MESR` register is turned on, this register contains one or four error syndromes. For parity memories, an error syndrome is '1' corresponding to a parity error or '0' corresponding to no parity error. For ECC memories, an error syndrome is a set of bits equal in length to the number of check bits corresponding to the associated portion of memory. A syndrome is zero when its portion of memory has no error. A non-zero syndrome gives more information about which bit or bits are in error. The exact encoding depends on the implementation. See the *Xtensa Microprocessor Data Book* for more information on the encoding.

When the `ErrTest` bit of the `MESR` register is set, certain load instructions write the actual check bits which have been read from memory into the fields of the `MECR` register. The load instructions which do this are `L32I/L32I.N` from local memory, `L32I/L32I.N` from an isolate mode address, and every `LICT`, `LICW`, `LDCT`, or `LDCW` instruction. Similarly, when the `ErrTest` bit of the `MESR` register is set, certain store instructions use the fields of the `MECR` register as the source of check bits to write. The store instructions which do this are `S32I/S32I.N` to local memory, `S32I/S32I.N` to an isolate mode address, and every `SICT`, `SICW`, `SDCT`, or `SDCW` instruction. Operation of other memory access instructions and cache fills/castouts is not defined when `ErrTest` is set. Operation is not defined if both `ErrEnab` and `ErrTest` are set.

Error addresses are reported with reference to the 32-bit word containing the error regardless of the size of the access and for all errors `MEVADDR` contains an address aligned to 32-bits. For data memories with 8-bit ECC, the check field(s) in `MECR` corresponding to the damaged byte(s) contains a non-zero syndrome. For tag memories, instruction memories and data memories with 32-bit ECC, the Check 0 field of `MECR` contains the syndrome for the entire word. For tag memories with multiple copies (to allow for multiple lookups per cycle), which copy of tag has the error is indicated by the two bits of `MEVADDR` just below the cache line

index bits. Errors in portions of the word not actually used by the access may or may not be reported in ME_{CCR}.

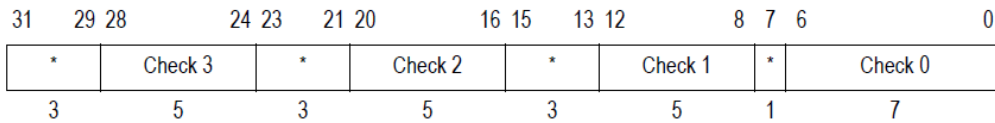


Figure 14: ME_{CCR} Register Format

Table 88: ME_{CCR} Register Fields

Field	Width (bits)	Definition
Check 3	5	<p>Check bits for the high order byte of a 32 bit data word.</p> <p>This field is valid for accesses to data RAM and data cache. It contains 5 check bits for ECC memories and 1 check bit (at the right end of the field) for parity memories. The field is associated with the highest address byte in little endian processors and the lowest address byte in big endian processors.</p>
Check 2	5	<p>Check bits for the next high order byte of a 32 bit data word.</p> <p>This field is valid for accesses to data RAM and data cache. It contains 5 check bits for ECC memories and 1 check bit (at the right end of the field) for parity memories. The field is associated with the second highest address byte in little endian processors and the second lowest address byte in big endian processors.</p>
Check 1	5	<p>Check bits for the next low order byte of a 32 bit data word.</p> <p>This field is valid for accesses to data RAM and data cache. It contains 5 check bits for ECC memories and 1 check bit (at the right end of the field) for parity memories. The field is associated with the second lowest address byte in little endian processors and the second highest address byte in big endian processors.</p>
Check 0	7	<p>Check bits for the low order byte of a 32 bit data word.</p> <p>For accesses to data RAM and data cache with 8-bit ECC or parity, this field contains five check bits for ECC memories and one check bit (at the right end of the field) for parity memories and is associated with the lowest address byte in little endian processors and the highest address byte in big endian processors.</p> <p>For accesses to instruction RAM, instruction cache, all cache tags, and data RAM/cache with 32-bit ECC, this</p>

Field	Width (bits)	Definition
		field contains seven check bits for ECC memories and one check bit (at the right end of the field) for parity memories and covers the whole 32-bit word or tag..
*		Reserved for future use Writing a non-zero value to one of these fields results in undefined processor behavior. These bits read as undefined.

The memory error virtual address register (*MEVADDR*), shown in [MEVADDR Register Format](#), contains address information regarding the location of the error. [MEVADDR Contents](#) details its contents as a function of two fields of the *MESR* register. For errors in cache tags and for errors in castout data, *MEVADDR* contains only index information. Along with the *Way Number* field in *MESR*, this allows the incorrect memory bits to be located. For errors in instructions or data being accessed, *MEVADDR* contains the full virtual address used by the instruction. Along with other status information, *MEVADDR* is written when the *ErrEnab* bit of the *MESR* register is set and the *RCE* or *MemE* bit of the *MESR* register is turned on.

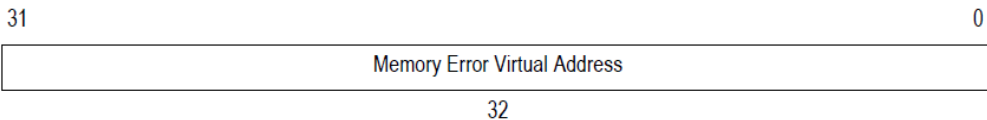


Figure 15: MEVADDR Register Format

Table 89: MEVADDR Contents

MESR Memory Type	MESR Access Type	MEVADDR Contents
Instruction RAM <i>n</i>		Full virtual address used in instruction.
Data RAM <i>n</i>		Full virtual address used in instruction.
Instruction cache tag array		Index bits are valid, other bits are undefined.
Instruction cache data array		Full virtual address used in instruction. ¹
Data cache tag array		Index bits are valid, other bits are undefined.
Data cache data array	LoadStore	Full virtual address used in instruction. ¹
Data cache data array	Castout	Index bits are valid, other bits are undefined.

MESR Memory Type	MESR Access Type	MEVADDR Contents
1. For LICW instructions or Isolate cache attributes, only the index and way bits along with lower order bits are valid.		

4.7.3 The Exception Registers

Three of the new registers created by this option are used in order to be able to take a memory error exception at any time and return. As an exception, memory error cannot be masked except by the `MESR.ErrEnab` bit. Whenever the exception is taken, the PC of the instruction taking the error is saved in the `MEPC` register, the PS register is saved in the `MEPS` register, and the `MESAVE` register is available for software use in the exception handler.

When an actual memory error exception is taken, the `MEPC` and `MEPS` registers are loaded with the original values of `PC` and `PS`, and then `PS.INTLEVEL` is raised to `NLEVEL` so that all interrupts except NMI are masked and the `PS.EXCM` bit is set so that an ordinary exception will cause a double exception. When hardware corrects a correctable memory error, these actions are not taken, allowing memory error corrections even in the memory error exception handler.

A memory error exception may be taken at any time. This means that, even without hardware correction, a memory error can be handled any time except during a memory error handler. With hardware correction, only an uncorrectable memory error taken during a handler for another uncorrectable memory error is fatal.

A Parity or ECC error in a cache tag memory makes it difficult to figure out whether the line should be valid, dirty, or locked. A tag may appear to represent a line it does not represent. Two tags may even appear to represent the same line. In a small percentage of cases, a Parity or ECC error may cause a General Exception(`LoadStoreErrorCause`) exception instead of a Memory Error Exception.

4.7.4 Memory Error Semantics

Memory errors have the following semantics:

```

procedure MemoryError
  return if !MESR.ErrEnab
  exc ← ParityError | UncorrectableECCError
  exc ← 1 if !MESR.MemE1 & MESR.InsExc & AccessType = IFetch
  exc ← 1 if !MESR.MemE & MESR.DatExc & AccessType ≠ IFetch
  MESR.ILCE ← 1 if exc & MESR.RCE & MESR.AccessType = IFetch
  MESR.DLCE ← 1 if exc & MESR.RCE & MESR.AccessType ≠ IFetch
  MESR.ILCE ← 1 if !exc & MESR.RCE & AccessType = IFetch

```

¹ Some implementations hardwire `MESR.InsExc` to '1' and never correct in hardware. For these implementations the exception occurs regardless of `MESR.MemE`.

```

MESR.DLCE ← 1 if !exc & MESR.RCE & AccessType ≠ IFetch
MESR.ILCE ← 1 if !exc & MESR.MemE & AccessType = IFetch
MESR.DLCE ← 1 if !exc & MESR.MemE & AccessType ≠ IFetch
if exc | !MESR.RCE then
    MESR.WayNumber ← WayNumber
    MESR.AccessType ← AccessType
    MESR.MemoryType ← MemoryType
    MESR.ErrorType ← ErrorType
    MECR ← CheckBits
    if MESR.AccessType = Castout then
        MEVADDR ← Undefined|CacheIndex|Undefined
    elsif MESR.MemoryType = Tag then
        MEVADDR ← Undefined|CacheIndex|Undefined
    else
        MEVADDR ← VAddr
    endif
    MESR.RCE ← !exc
endif
if exc then
    MESR.DME ← MESR.MemE
    MESR.MemE ← 1
    MEPC ← PC
    MEPS ← PS
    nextPC ← MemoryErrorExceptionVector
    PS.INTLEVEL ← NLEVEL
    PS.EXCM ← 1
endif
endprocedure MemoryError

```

5. Options for Memory Protection and Translation

Topics:

- [Overview of Memory Management Concepts](#)
- [The Memory Access Process](#)
- [Region Protection Option](#)
- [Region Translation Option](#)
- [Memory Protection Unit Option](#)
- [MMU Option](#)

Xtensa processors employ one of the options in this section for memory protection and translation. The introduction in [Overview of Memory Management Concepts](#) on page 184 provides background information for the options in this section. The Region Protection Option described in [Region Protection Option](#) on page 196 provides control of memory by 512 MB regions. Within each region, accessibility, cacheability, and characteristics of cacheability can be controlled. The Region Translation Option described in [Region Translation Option](#) on page 202 builds on that and adds a translation table with an entry for each region so that virtual addresses in that region can be translated to corresponding physical addresses in any of the 512 MB regions. The Memory Protection Unit Option, described in [Memory Protection Unit Option](#) on page 205, is a more flexible protection option but without translation. It never misses and does not use a page table. The MMU Option described in [MMU Option](#) on page 217 is a full paging memory management unit. It supports hardware refill of the TLB from page tables in memory.

5.1 Overview of Memory Management Concepts

[Overview of Memory Translation](#) on page 184 gives an overview of the basic memory translation scheme used in Xtensa processors. [Overview of Memory Protection](#) on page 186 gives an overview of the basic memory protection scheme used in Xtensa processors, and [Overview of Attributes](#) on page 189 gives an overview of the concept of attributes. These subsections take a broader view of the overall process and indicate the direction future memory protection and translation options may take.

5.1.1 Overview of Memory Translation

This subsection presents an overview of the thinking behind the memory translation in the available options. It also provides insight into the kinds of extensions that are likely in the future.

The available memory protection and translations options that support virtual-to-physical address translation do so via an instruction TLB and a data TLB. (“TLB” was originally an acronym for translation lookaside buffer, but this meaning is no longer entirely accurate; in this document TLB simply means the translation hardware.) These two hardware structures may, in some configurations, act as translation caches that are refilled by hardware from a common page table structure in memory. In other configurations, a TLB may be self-sufficient for its translations, and no page tables are required.

A TLB consists of several entries, each of which maps one page (the page size may vary with each entry). Virtual-to-physical address translation consists of searching the TLB for an entry that matches the most significant bits of the virtual address and replacing those bits with bits from the TLB entry. The least significant bits of the virtual address are identical between the virtual and physical addresses. The translation input and output are called the virtual page number (VPN) and the physical page number (PPN) respectively. The TLB search also involves matching the address space identifier (ASID) bits of the TLB entry to one of the current ASIDs stored in the `RASID` register (more on this below). The number of bits not translated is determined by the page size, which can be dynamically programmed from a set of configuration specified values. The TLB entry also supplies some attribute bits for the page, including bits that determine the cacheability of the page’s data, whether it is writable or not, and so forth. This is illustrated in [Virtual-to-Physical Address Translation](#).

It is illegal for more than one TLB entry to match both the virtual address and the ASID. This is true even if the entries have different ASIDs which match at different ring levels. Software is responsible for making sure the address range of all TLB entries visible according to the ASID values in the `RASID` register never overlap. Implementations may detect this situation and take a MultiHit exception in this situation to aid in debugging.

The instruction and data TLBs can be configured independently for most parameters, which is appropriate because the instruction and data references of processors can have fairly different requirements, and in some systems additional flexibility may be appropriate on one

but not the other. However, when the two TLBs both refill from the common memory page table, the associated parameters are shared.

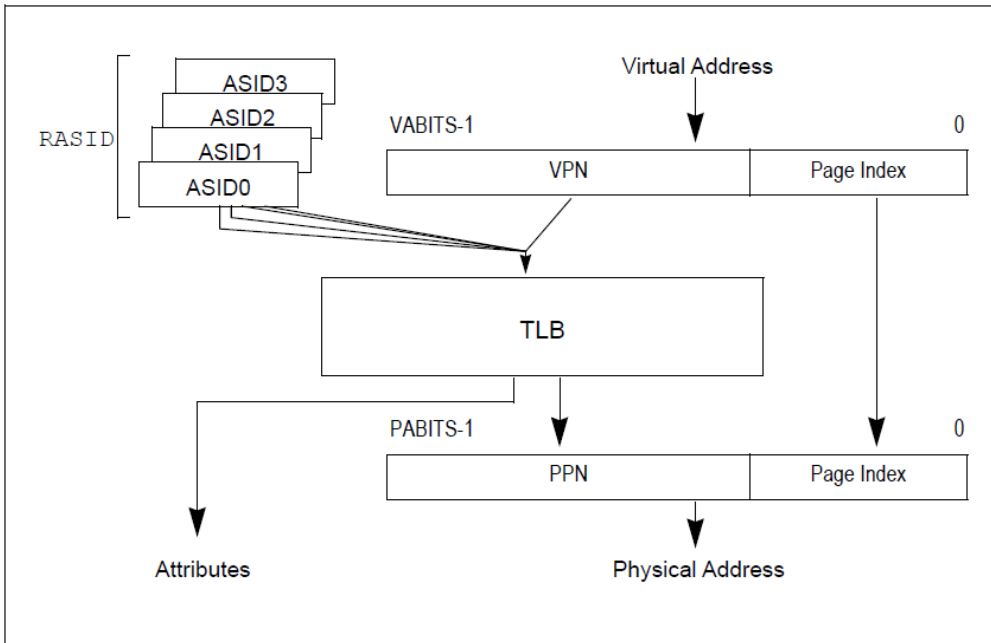


Figure 16: Virtual-to-Physical Address Translation

Xtensa implementations may perform virtual-to-physical address translation in parallel or series with cache, RAM, ROM, and XLMI access. However, the translated physical address is always used to decide which cache, RAM, or ROM access to use. Thus caches are potentially virtually indexed, even though they are always physically tagged. When the number of cache index bits (that is $\log_2(\text{CacheBytes}/\text{WayCount})$) is greater than a page index and the same physical memory is mapped at multiple virtual addresses, there is the possibility of multiple cache locations being used for the same physical memory line, which can lead to the multiple views of memory being inconsistent. In such a system, software typically avoids this situation by restricting the virtual addresses for multiply mapped physical memory. This software restriction is often referred to as “page coloring.” If physically indexed caches are necessary (and generally they are not), the system designer may configure the TLBs such that cache index is a physical address by using a large page size or a high cache associativity so that the cache index bits are within the portion of the virtual and physical addresses that are identical.

The TLBs are N-way set-associative structures with heterogeneous “ways” and a configurable N. Each way has its own parameters, such as the number of entries, page size(s), constant or variable virtual address, and constant or variable physical address and attributes. It is the ability to specify constant translations in some or all of the ways that allows Xtensa’s TLBs to span smoothly from a fixed memory map to a fully programmable one. Fully

or partially constant entries can be converted to logic gates in the TLB at significantly lower cost than a run-time programmable way. In addition, even processors with generally programmable MMUs often have a few hardwired translations. Xtensa can easily represent these hardwired translations with its constant TLB entries. Xtensa actually requires a few constant TLB entries to provide translation in some circumstances, such as at reset and during exception handling.

The virtual address input to the TLBs is actually the catenation of an address space identifier (ASID) specified in a processor register with the 32-bit virtual address from the fetch, load, or store address calculation. ASIDs allow software to change the address space seen by the processor (for example, on a context switch) with a simple register write without changing the TLB contents. The TLB stores an ASID with each entry, and so can simultaneously hold translations for multiple address spaces. The number of ASID bits is configurable. ASIDs are also an integral part of protection, as they specify the accessibility of memory by the processor at different privilege levels, as described in the next section.

Xtensa TLBs do not have a separate valid bit in each entry. Instead, a reserved ASID value of 0 is used to indicate an invalid entry. This can be viewed as saving a bit, or as almost doubling the number of ASIDs for the same number of hardware bits stored in a TLB entry.

Non-constant ways may be configured as AutoRefill. If no entry matching an access is found in a TLB with one or more AutoRefill ways, the processor will attempt to load a page table entry (PTE) from memory and write it into an entry of one of the AutoRefill ways. A TLB with no AutoRefill ways does not use the page table.

Each way of a TLB is configured with a list of page sizes (expressed as the number of bits in a page index). If the list has one element, the page size for that way is fixed. If the list has more than one element, the page size of the way may be varied at runtime via the `ITLBCFG` or `DTLBCFG` registers. When AutoRefill ways have programmable page size, the PTE has a page size field (the value is an index into the `PTEPageSizes` configuration parameter), and hardware refill restricts the refill way selection to ways programmed with a page size matching the page size in the PTE. When looking up an address in the TLB, each way's page size determines which bits are used to select one of the way's entries for comparison: $vAddr_p + \log_2(IndexCount) - 1..P$ is the way index where P is the number of bits configured or programmed for the way page size.

5.1.2 Overview of Memory Protection

Many processors implement two levels of privilege, often called kernel and user, so that the most privileged code need not depend on the correctness of less privileged code. The operating system kernel has access to the entire processor, but disables access to certain features while application code runs to prevent the application from accessing or corrupting the kernel or other applications. This mechanism facilitates debugging and improves system reliability.

Some processors implement multiple levels of decreasing privilege, called rings, often with elaborate mechanisms for switching between rings. The Xtensa processor provides a

configurable number of rings (`RingCount`), but without the elaborate ring-to-ring transition mechanisms. When configured with two rings, it provides the common kernel/user modes of operation, with Ring 0 being kernel and Ring 1 being user. With three or four rings configured, the Xtensa processor provides the same functionality as more advanced processors, but with the requirement that ring-to-ring transitions must be provided by Ring 0 (kernel) software.

With the Region Protection Option, or with the MMU Option and `RingCount = 1`, the Xtensa processor has a single level of privilege, and all instructions are always available.

With `RingCount > 1`, software executing with `CRING = 0` (see [PS Register Fields](#) and the description of `PS.EXCM`) is able to execute all Xtensa instructions; other rings may only execute non-privileged instructions. The only distinction between the rings greater than zero is those created by software in the virtual-to-physical translations in the page table. The name “ring” is derived from an accessibility diagram for a single process such as that shown in [A Single Process’ Rings](#). At Ring 0 (that is, when `CRING = 0`), the processor can access all of the current process’ pages (that is, Ring 0 to `RingCount-1` pages). At Ring 1 it can access all Ring 1 to `RingCount-1` pages. Thus, when the processor is executing with Ring 1 privileges, its address space is a subset of that at Ring 0 privilege, as [A Single Process’ Rings](#) illustrates. This concentric nesting of privilege levels continues to ring `RingCount-1`, which can access only ring `RingCount-1` pages.

It is illegal for more than one TLB entry to match both the virtual address and the ASID. This is true even if the entries have different ASIDs which match at different ring levels. One ring’s mapping cannot not override another.

It is illegal for two or more TLB entries to match a virtual address, even if they are at different ring levels; one ring’s mapping cannot not override another.

Systems that require only traditional kernel/user privilege levels can, of course, configure `RingCount` to be 2. However, rings can also be useful for sharing. Many operating systems implement the notion of multiple threads sharing an address space, except for a small number of per-thread pages. Such a system could use Ring 0 for the shared kernel address space, Ring 1 for per-process kernel address space, Ring 2 for shared application address space, and Ring 3 for per-thread application address space.

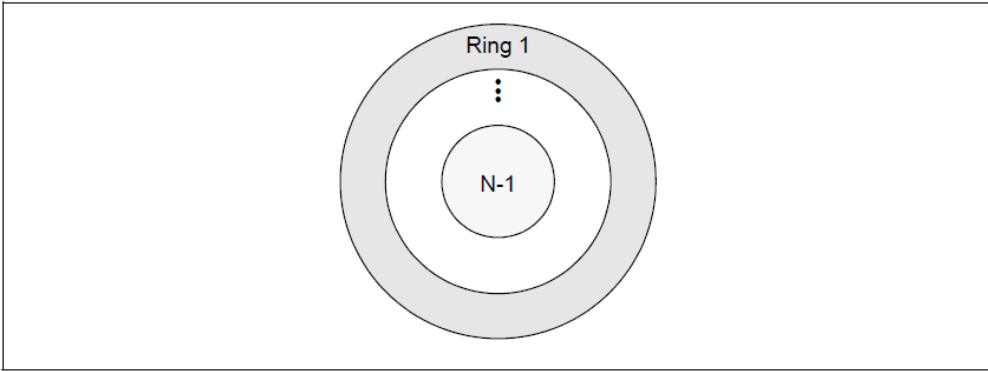


Figure 17: A Single Process' Rings

Each Xtensa ring has its own ASID. Ring 0's ASID is hardwired to 1. The ASIDs for Rings 1 to `RingCount-1` are specified in the `RASID` register. The ASIDs for each ring in `RASID` must be different. Each ASID has a single ring level, though there may be many ASIDs at the same ring level (except Ring 0). This allows nested privileges with sharing such as shown in [Nested Rings of Multiple Processes with Some Sharing](#). The ring number of a page is not stored in the TLB; only the ASID is stored. When a TLB is searched for a virtual address match, the ASIDs of all rings specified in `RASID` are tried. The position of the matching ASID in `RASID` gives the ring number of the page. If the page's ring number is less than the processor's current ring number (`CRING`), then the access is denied with an exception (either `InstFetchPrivilegeCause` or `LoadStorePrivilegeCause`, as appropriate).

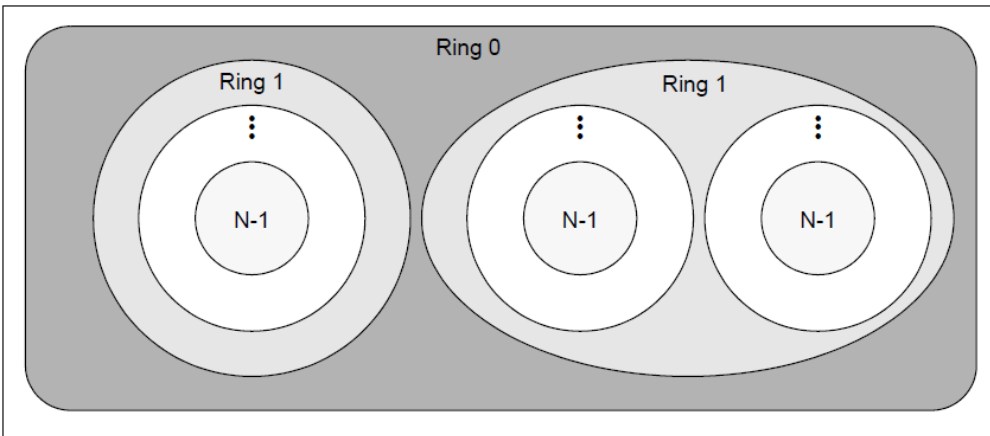


Figure 18: Nested Rings of Multiple Processes with Some Sharing

Why not store the ring number of the page in the TLB, and then use a single ASID for all rings, instead of having an ASID per ring? Because the latter allows sharing of TLB entries, and the former does not. For example, it is desirable at the very least to reuse the same TLB entries for all kernel mapped addresses, instead of having the same PTEs loaded into the

TLB with different ASIDs. The Xtensa mechanism is more general than adding a “global” bit to each entry (to ignore the ASID match) in that it allows finer granularity, as *Nested Rings of Multiple Processes with Some Sharing* illustrates, not just all or nothing.

The kernel typically assigns ASIDs dynamically as it runs code in different address spaces. When no more ASIDs are available for a new address space, the kernel flushes the Instruction and Data TLBs, and begins assigning ASIDs anew. For example, with `ASIDBits = 8` and `RingCount = 2`, a TLB flush need occur at most every 254 context switches, if every context switch is to a new address space.

Note that `CRING = 0` is the only requirement for privileged instructions to execute and `CRING` is the only field that controls access to memory. The `PS.UM` bit is named User Vector Mode and has nothing to do with privilege for either instructions or memory access. It controls only which exception vector is taken for general exceptions.

5.1.3 Overview of Attributes

Both page table entries (PTEs) and TLB entries store attribute bits that control whether and how the processor accesses memory. The number of potential attributes required by systems is large; to encode all the access capabilities required by any potential system would make this field too big to fit into a 4-byte PTE. However, the subset of values required for any particular system is usually much smaller. Each memory protection and translation option has a set of attributes, each of which encodes a set of capabilities from *Access Characteristics Encoded in the Attributes* for loads along with a set for stores and a set for instruction fetches. More capabilities are likely to be added in future implementations.

Table 90: Access Characteristics Encoded in the Attributes

Characteristic	Description	Used by
Invalid	Exception on access	Fetch, Load, Store
Isolate	Read/write cache contents regardless of tag compare	Fetch, Load, Store
Bypass	Ignore cache contents regardless of tag compare — always access memory for this page	Fetch, Load, Store
No-allocate	Do not refill cache on miss	Fetch, Load, Store
Write-through	Write memory in addition to DataCache	Store
Guarded	Access bytes on this page exactly when required by the program (i.e. neither speculative references to	Load ¹

Characteristic	Description	Used by
	reduce latency nor multiple accesses are allowed).	
1. Instruction fetch is always non-guarded. Stores are always guarded.		

The assignment of capabilities to the attribute field of PTEs may be done with only one encoding for each distinct set of capabilities, or in such a way that each characteristic has its own bit, or anything in between. Often, single bits are used for a valid bit and a write-enable. For a valid bit, all of the attribute values with this bit zero would specify the Invalid characteristic so that any access causes an `InstFetchProhibitedCause`, `LoadProhibitedCause`, or `StoreProhibitedCause` exception, depending on the type of access. Similarly for the write-enable bit, all attribute values with write-enable zero would specify the Invalid characteristic to cause a `StoreProhibitedCause` exception on any store.

For systems that implement demand paging, software requires a page dirty bit to indicate that the page has been modified and must be written back to disk if it is replaced. This may be provided by creating a write-enable bit as described above, and using it as the per-page dirty bit. The first write to a clean (non-dirty) page causes a `StoreProhibitedCause` exception. The exception handler checks one of the software bits, which indicates whether the page is really writable or not; if it is, it then sets the hardware write-enable bit in both the TLB and the page table, and continues execution.

5.2 The Memory Access Process

All accesses to memory, whether to cache, local memories, XLMI, or PIF and whether caused by instruction fetch, the instructions themselves, or hardware TLB refill, follow certain steps. Following is a short description of these steps; each is discussed in more detail in [Choose the TLB](#) on page 191 through [Direct the Access to Cache](#) on page 196.

- 1. Choose the TLB:** Determine from the instruction opcode or the reason for hardware access, which TLB if any, is used for the access (see [Choose the TLB](#) on page 191 for details).
- 2. Lookup in the TLB:** In that TLB, find an entry whose virtual page number matches the upper bits of the virtual address of the access and, for appropriate options, whose `ASID` matches one of the entries in the `RASID` register. Exactly one match is needed to continue beyond this point, although exceptions may be handled and the memory access process restarted (see [Lookup in the TLB](#) on page 192 for details).
- 3. Check the access rights:** If the attribute is invalid or, for appropriate options, if the ring corresponding the `ASID` matched in the `RASID` register is too low, raise an exception. The operating system may, among other choices, modify the TLB entries and retry the access (see [Check the Access Rights](#) on page 193 for details).

4. **Direct the access to local memory:** If the physical address of the access matches an instruction RAM or ROM, a data RAM or ROM, or an XLMI port then direct the access to that local memory or XLMI. An exception is possible at this stage for certain conditions, such as attempting to write to a ROM (see [Direct the Access to Local Memory](#) on page 193 for details).
5. **Direct the access to PIF:** For the given cache configuration and using the attribute, determine whether to execute the required access on the processor interface bus (PIF) and make that access if necessary (see [Direct the Access to PIF](#) on page 196 for details).
6. **Direct the access to cache:** Using the cache that corresponds to the TLB in Step 1 above, look up the memory location in the cache, using the value if it is there. If not, fill the cache from the PIF and then do the access (see [Direct the Access to Cache](#) on page 196 for details).

Logically, the steps are done in order. The TLB lookup is done first (in steps 1 through 3 above) and the memory access afterwards (in steps 4 through 6 above). For performance reasons, they are actually done in parallel. This has two consequences:

1. First, the virtual and physical addresses of an access to an XLMI port must be identical so that the full address can be provided at the desired time.
2. Second, for all other local memory accesses and cacheable addresses, the index bits of the cache or local memory must be the same in both virtual and physical address. This means that caches which contain ways larger than the smallest page size in the system require “page coloring” as described in [Overview of Memory Translation](#) on page 184.

For local memories, the second consequence requires a similar restriction on how they can be mapped. Note that local memories do not require that sequential virtual pages be mapped to sequential physical pages, but only that each virtual page be mapped to a physical page with which it shares the values of index bits.

For the purposes of understanding exceptions raised by memory accesses, all the steps above are done sequentially and the first exception encountered takes priority over later ones. For performance reasons, again, all steps are done in parallel and the results prioritized afterward.

The above steps are further expanded in the following subsections.

5.2.1 Choose the TLB

Several instructions do not actually address memory. They simply use the bits of an address to access a cache and do something directly to it. The following groups of instructions have this property:

- III, IIU
- DII, DIU, DIWB, DIWBI
- LICT, SICT, LICW, SICW
- LDCT, SDCT

For each of these instructions, no TLB is accessed and the remainder of the steps are not followed. No memory access exceptions are possible as the addresses are not really addresses but only pointers to cache locations.

For the data accesses of instructions `IHI`, `IHU`, `IPF`, and `IPFL`, as well as all instruction fetches, the instruction TLB is used for subsequent steps.

For the data accesses of all other instructions and for the hardware TLB refill accesses (regardless of which TLB is being refilled) the data TLB is used for subsequent steps.

The above choices are reflected in [Local Memory Accesses](#) in the second column.

For compatibility the two TLBs should never give conflicting translations or protection attributes for any access as future processors may implement them with only a single set of entries.

5.2.2 Lookup in the TLB

Each TLB lookup takes a virtual address as an operand and produces a physical address, a lookup ring, and attributes as a result. This process is described in more detail in [Overview of Memory Translation](#) on page 184. Each way of the TLB is read using the appropriate address bits for that way as index bits. For variable sized ways, the `ITLBCFG` or `DTLBCFG` register helps determine which address bits are the index bits.

For options without `ASIDs` (Region Protection Option), a way matches the access if its virtual page number (VPN) matches the VPN of the access. The lookup ring produced is defined to be 0.

For options with `ASIDs` (MMU Option), a way matches the access if its Virtual Page Number (VPN) matches the VPN of the access and the `ASID` of the way matches one of the `ASIDs` in the `RASID` register. The lookup ring is determined by which `ASID` in the `RASID` register is matched. Because the four entries in the `RASID` register are required to be different and non-zero, the lookup ring is well determined.

There should not be a match for more than one of the ways. Under the Exception Option 2, this condition raises an `InstTLBMultiHitCause` or a `LoadStoreTLBMultiHitCause` exception as a debugging aid.

If none of the ways match, options without auto-refill ways (Region Protection Option) will raise an `InstTLBMissCause` or a `LoadStoreTLBMissCause` exception under the Exception Option 2 so that system software can take appropriate action and possibly retry the access. Options with auto-refill ways (MMU Option) will, automatically in hardware, use `PTEVADDR` to access page tables in memory and replace an entry in one of the auto-refill ways. The access will then be automatically retried. An error of any sort during the automatic refill process will raise an `InstTLBMissCause` or a `LoadStoreTLBMissCause` exception to be raised so that system software can take appropriate action and possibly retry the access.

If no exception is raised, the physical page number and attributes of the matching entry along with the lookup ring defined above are the results of the lookup and the access continues with the next step.

5.2.3 Check the Access Rights

First, the lookup ring of the entry is checked against the ring of the access. The ring of the access is usually `CRING`, but for `L32E` and `S32E`, for example, it is `PS.RING` instead. If the lookup ring of the entry is smaller than the ring of the access, an `InstFetchPrivilegeCause` or a `LoadStorePrivilegeCause` exception is raised under the Exception Option 2. This situation means that an instruction has attempted access to a region of memory at a lower numbered ring than the one for which it has privilege.

Second, the attribute of the lookup is checked for validity. If the attribute is not valid, an exception is raised. Under the Exception Option 2, if the access chose the Instruction TLB in [Choose the TLB](#) on page 191, it raises an `InstFetchProhibitedCause` exception. Under the Exception Option 2, if it chose the data TLB, it raises either a `LoadProhibitedCause` exception or a `StoreProhibitedCause` exception, depending on whether it was a load or a store.

If no exception is raised, the access continues with the next step using the physical address and the attribute (which is known to be valid for access, but may still affect how caches are used).

5.2.4 Direct the Access to Local Memory

The physical address of each access is compared to the address ranges of any instruction RAM, instruction ROM, data RAM, data ROM that may exist in the processor. [Local Memory Accesses](#) indicates what will happen in the case that an access initiated by what is indicated in the Instruction column (which will use the TLB in the second column) if its address compares to an (abbreviated) option in one of the last six columns. OK means the access is completed normally. NOP means the access is completed but by its nature does nothing. IFE and LSE mean that an exception is raised. TLBI and TLBD mean that an `InstTLBMissCause` or a `LoadStoreTLBMissCause` exception is raised. Undef means the behavior is not defined.

Table 91: Local Memory Accesses

Instruction	TLB Used ¹	InstRAM	InstROM	DataRAM	DataROM	XLMI
Instruction-fetch	ITLB	OK	OK	IFE ²	IFE ²	IFE ²
IHI, IHU, IPF	ITLB	NOP	NOP	NOP	NOP	NOP
III, IIU	none	—	—	—	—	—

Instruction	TLB Used ¹	InstRAM	InstROM	DataRAM	DataROM	XLMI
IPFL	ITLB	IFE ²	IFE ²	IFE ²	IFE ²	IFE ²
L32I, L32R	DTLB	Note ³	Note ³	OK	OK	OK
L8UI, L16SI, L16UI, L32AI, L32E, FP Loads, MAC16 Loads	DTLB	LSE ⁴	LSE ⁴	OK	OK	OK
LICT, LICW, LDCT, LDCW	none	—	—	—	—	—
S32I	DTLB	Note ³	LSE ⁴	OK	LSE ⁴	OK
S8I, S16I, S32E, S32RI, FP Stores	DTLB	LSE ⁴	LSE ⁴	OK	LSE ⁴	OK
S32C1I	DTLB	LSE ⁴	LSE ⁴	OK ⁵	LSE ⁴	Undef
SICT, SICW, SDCT, SDCW	none	—	—	—	—	—
DHI, DHU, DHWB, DHWBI, DCI, DCWB, DCWBI	DTLB	NOP	NOP	NOP	NOP	NOP
DII, DIU, DIWB, DIWBI	none	—	—	—	—	—

Instruction	TLB Used ¹	InstRAM	InstROM	DataRAM	DataROM	XLMI
DPFR, DPFRO, DPFW, DPFWO	DTLB	NOP	NOP	NOP	NOP	NOP
DPFL	DTLB	LSE ⁴	LSE ⁴	LSE ⁴	LSE ⁴	LSE ⁴
Hardware ITLB Refill	DTLB	TLBI ⁶	TLBI ⁶	OK	OK	OK
Hardware DTLB Refill	DTLB	TLBD ⁶	TLBD ⁶	OK	OK	OK
Designer defined loads	DTLB	LSE ⁴	LSE ⁴	OK	OK	OK
Designer defined stores	DTLB	LSE ⁴	LSE ⁴	OK	LSE ⁴	OK

1. As described in [Choose the TLB](#) on page 191
2. Raises an InstFetchErrorCause exception under the Exception Option 2 .
3. Raises an InstFetchErrorCause exception under the Exception Option 2 . In addition, these accesses may be slow in some implementations. See [Assembler Syntax](#) for more detail.
4. Raises a LoadStoreErrorCause exception under the Exception Option 2 .
5. Works in newer implementations but in some older implementations raises an exception.
6. Under the Exception Option 2, raises an exception - InstTLBMissCause or a LoadStoreTLBMissCause depending on the original access.

Using the definition of guarded in [Access Characteristics Encoded in the Attributes](#), instruction-fetch accesses are never guarded. Stores are always guarded. Loads to instruction RAM, instruction ROM, Data RAM, and Data ROM are never guarded. These ports are assumed to be connected only to devices with memory semantics so that no guarding is needed for loads. Loads to XLMI are only guarded in the sense that the load will be retired only under the conditions for a guarded access. For all these memories, assertion of the memory enable is no guarantee that the load was needed.

If none of the comparisons produces a match, the access continues with the next step using the physical address and the attribute.

5.2.5 Direct the Access to PIF

The access is sent to the processor interface if any of the following is true:

- The attribute indicates that the cache should be bypassed.
- The chosen TLB in [Choose the TLB](#) on page 191 and in [Local Memory Accesses](#) is the ITLB .
- The chosen TLB in [Choose the TLB](#) on page 191 and in [Local Memory Accesses](#) is the DTLB .

Using the definition of guarded in [Access Characteristics Encoded in the Attributes](#), instruction-fetch accesses to the PIF are never guarded. Stores to the PIF are always guarded. Loads that are sent to the PIF under this section (without being cached) are guarded if the attribute says that they should be.

If the conditions of this section are not met, the access is cached and continues with the next step using the physical address and the attribute.

5.2.6 Direct the Access to Cache

The access is cached. The attribute determines how the cache operates, including the possibility of a write-through to the PIF.

The concept of guarding cannot be carried out for loads through the cache. Extra bytes have been loaded simply to fill the cache line and the line may have been filled long before the access. Inherently, the line is filled a different number of times than an access is executed and the line may be invalidated or evicted at any time and refilled later. Caching should not be used on ranges of memory address where guarding is important.

5.3 Region Protection Option

The simplest of the options, the Region Protection Option, provides a protection field for each of the eight 512 MB regions in the address space. The field can allow access to the region and it can set caching characteristics for the region, such as whether or not the cache is used and if it is write-through or write-back.

- Prerequisites: [Exception Option 2](#) on page 126
- Incompatible options: [Memory Protection Unit Option](#) on page 205, [MMU Option](#) on page 217
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

This simple option is built from the capabilities discussed in the introduction ([Overview of Memory Management Concepts](#) on page 184). It uses `RingCount = 1`, so the processor can always execute privileged instructions. It sets `ASIDBits` to 0, which disables the ASID feature. The instruction and data TLBs are programmed to each have one way of eight entries, and the VPNs (virtual page numbers) and PPNs (physical page numbers) of these

entries are constant and hardwired to the identity map (that is, PPN = VPN). Only the attributes are not constant; they are writable using the `WITLB` and `WDTLB` instructions.

5.3.1 Region Protection Option Architectural Additions

[Region Protection Option Exception Additions](#) through [Region Protection Option Instruction Additions](#) show this option's architectural additions.

Table 92: Region Protection Option Exception Additions

Exception	Description	EXCCAUSE value
<code>InstFetchProhibitedCause</code>	Instruction fetch is not allowed in region	20
<code>LoadProhibitedCause</code>	Load is not allowed in region	28
<code>StoreProhibitedCause</code>	Store is not allowed in region	29

Table 93: Region Protection Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Access
ITLB Entries	8	4	Instruction TLB entries	R/W	see Region Protection Option Instruction Additions
DTLB Entries	8	4	Data TLB entries	R/W	see Region Protection Option Instruction Additions

Table 94: Region Protection Option Instruction Additions

Instruction ¹	Format	Definition
<code>IDTLB</code>	RRR on page 656	Invalidate data TLB entry
<code>IITLB</code>	RRR on page 656	Invalidate instruction TLB entry
<code>PDTLB</code>	RRR on page 656	Probe data TLB
<code>PITLB</code>	RRR on page 656	Probe instruction TLB

Instruction ¹	Format	Definition
RDTLB0	RRR on page 656	Read data TLB virtual
RDTLB1	RRR on page 656	Read data TLB translation
RITLB0	RRR on page 656	Read instruction TLB virtual
RITLB1	RRR on page 656	Read instruction TLB translation
WDTLB	RRR on page 656	Write data TLB
WITLB	RRR on page 656	Write instruction TLB

1. These instructions are fully described in [Instruction Descriptions](#) on page 321.

5.3.2 Formats for Accessing Region Protection Option TLB Entries

During normal operation when instructions and data are being accessed from memory, only lookups are being done in the TLBs. For maintenance of the TLBs, however, the entries in the TLBs are accessed by the instructions in [Region Protection Option Instruction Additions](#). Note that unused bits at Bit 12 and above are ignored on write, and zero on read, so that those bits may simply contain the address for access to all ways of both TLBs. Unused bits at Bit 11 and below are required to be zero on write and undefined on read for forward compatibility.

The format of the `as` register used in all instructions in the table is shown in [Region Protection Option Addressing \(`as`\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#). The upper three bits are used as an index among the TLB entries just as they would be when addressing memory. They are the Virtual Page Number (VPN) or upper three bits of address. The remaining bits are ignored.

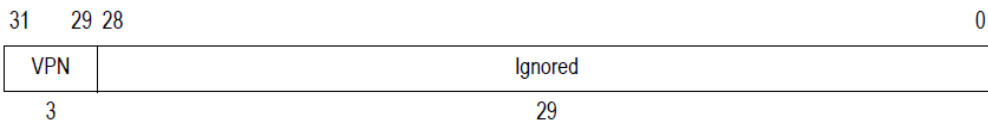


Figure 19: Region Protection Option Addressing (`as`) Format for `WxTLB`, `RxTLB1`, & `PxTLB`

The `WITLB` and `WDTLB` instructions write the TLB entries. The `as` register is formatted according to [Region Protection Option Addressing \(`as`\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#), while the `at` register is formatted according to [Region Protection Option Data \(`at`\) Format for `WxTLB`](#). The attribute for the region is described in detail in [Region Protection Option Memory Attributes](#) on page 200. The remaining bits are ignored or required to be zero.

After modifying any TLB entry with a `WITLB` instruction, an `ISYNC` must be executed before executing any instruction from that region. In the special case of the `WITLB` changing the attribute of its own region, the `ISYNC` must immediately follow the `WITLB` and both must be within the same memory region and, if the region is cacheable, within the same cache line.

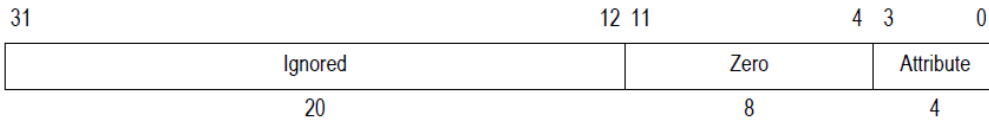


Figure 20: Region Protection Option Data (at) Format for `WxTLB`

The `RITLB0` and `RDTLB0` instructions exist under this option but do not return interesting information because the entire VPN is used as an index. The `as` register is formatted according to [Region Protection Option Addressing \(as\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#). The read instructions return zero in the `at` register.

The `RITLB1` and `RDTLB1` instructions return the `at` data format in [Region Protection Option Data \(at\) Format for `RxTLB1`](#). The Attribute for the region is described in detail in [Region Protection Option Memory Attributes](#) on page 200. The VPN is returned in the upper three bits as the Physical Page Number (PPN) because there is no translation. The remaining bits are zero or undefined. The `as` register is formatted according to [Region Protection Option Addressing \(as\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#).

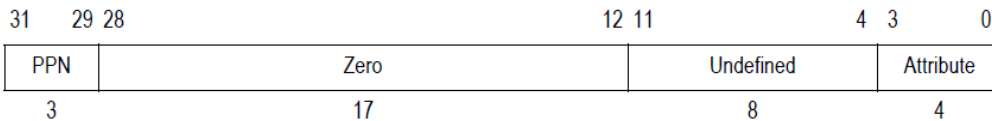


Figure 21: Region Protection Option Data (at) Format for `RxTLB1`

The `PITLB` and `PDTLB` instructions exist under this option but do not return interesting information because all accesses hit in the respective TLBs and the TLBs have only a single way. The `as` register is formatted according to [Region Protection Option Addressing \(as\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#). The TLB probe instructions return the `at` data format in [Region Protection Option Data \(at\) Format for `PxTLB`](#). The VPN is returned in the upper bits. The low bit is set because the probe always hits and the remaining bits are zero or undefined.

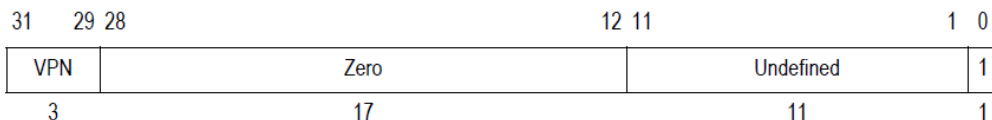


Figure 22: Region Protection Option Data (at) Format for `PxTLB`

The `IITLB` and `IDTLB` instructions exist under this option and their `as` register is formatted according to [Region Protection Option Addressing \(*as*\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#), but they have no effect because the entries cannot be removed from the respective TLBs.

5.3.3 Region Protection Option Memory Attributes

The memory attributes written into the TLB entries by the `WxTLB` instructions and read from them by the `RxTLB1` instructions control access to memory and, where there is a cache, how the cache is used. [Region Protection Option Attribute Field Values](#) shows the meanings of the attributes for instruction fetch, data load, and data store. For a more detailed description of the memory access process and the place of these attributes in it, see [The Memory Access Process](#) on page 190.

The first column in [Region Protection Option Attribute Field Values](#) indicates the attribute from the TLB while the remaining columns indicate various effects on the access. The columns are described in the following bullets:

- **Attr** — the value of the 4-bit Attribute field of the TLB entry.
- **Rights** — whether the TLB entry may successfully translate a data load, a data store, or an instruction fetch.
 - The first character is an `r` if the entry is valid for a data load and a dash ("-") if not.
 - The second character is a `w` if the entry is valid for a data store and a dash ("-") if not.
 - The third character is an `x` if the entry is valid for an instruction fetch and a dash ("-") if not.

If the translation is not successful, an exception is raised.

Local memory accesses (including XLMI) consult only the Rights column.

- **Meaning for Cache Access** — the verbal description of the type of access made to the cache.
- **Access Cache** — indicates whether the cache provides the data.
 - The first character is an `h` if the cache provides the data when the tag indicates hit and a dash ("-") if it does not.
 - The second character is an `m` if the cache provides the data when the tag indicates a miss and a dash ("-") if it does not. This capability is used only for Isolate mode.
- **Fill Cache** — indicates whether an allocate and fill is done to the cache if the tag indicates a miss.
 - The first character is an `r` if the cache is filled on a data load and a dash ("-") if it is not.
 - The second character is a `w` if the cache is filled on a data store and a dash ("-") if it is not.
 - The third character is an `x` if the cache is filled on an instruction fetch and a dash ("-") if it is not.
- **Guard Load** — refers to the guarded attribute as described in [Access Characteristics Encoded in the Attributes](#). Stores are always guarded and instruction fetches are never

guarded, but loads are guarded where there is a “yes” in this column. Local memory loads are not guarded.

- **Write Thru** — indicates whether a write is done through the PIF interface.
 - The first character is an _h if a Write Thru occurs when the tag indicates hit and a dash (“-”) if it does not.
 - The second character is an _m if a Write Thru occurs when the tag indicates a miss and a dash (“-”) if it does not.

Writes to local memories are never Write-Thru. In most implementations, a write-thru will only occur after any needed cache fill is complete.

Table 95: Region Protection Option Attribute Field Values

Attr	Rights	Meaning for Cache Access	Access Cache	Fill Cache	Guard Load	Write Thru
0	rw-	Cached, No Allocate	h-	---	-	hm
1	rwX	Cached, WrtThru	h-	r-x	-	hm
2	rwX	Bypass cache	--	---	yes	hm
3	--x	Cached ¹	h-	--x	-	--
4	rwX	Cached, WrtBack alloc	h-	rwX ²	-	-- ²
5	rwX	Cached, WrtBack noalloc ¹	h-	r-x	-	-m ²
6	rwX	Bypass cache - Bufferable ⁵	--	---	yes	hm
7-13	---	Reserved ³	—	—	—	—
14	rw-	Cache Isolated ⁴	hm	---	-	--
15	---	illegal ³	--	---	-	--

1. Attribute not supported in all implementations. Refer to a specific *Xtensa Microprocessor Data Book* for supported attributes.

Attr	Rights	Meaning for Cache Access	Access Cache	Fill Cache	Guard Load	Write Thru
<ol style="list-style-type: none"> 2. If the Data Cache is not configured as writeback, entries for Attributes 4 & 5 are replaced by the corresponding ones for Attribute 1 3. Raises exception. Under the Exception Option 2, EXCCAUSE is set to InstFetchProhibitedCause, LoadProhibitedCause, or StoreProhibitedCause depending on access type. 4. For test only, implementation dependent, uses data cache like local memories and ignores tag. 5. The same as Attribute 2 except that, if the external bus supports it, the accesses are marked as Bufferable 						

All attribute entries in the ITLB and DTLB are set to cache bypass (4'h2) after reset.

After changing the attribute of any memory region with a `WITLB` instruction, an `ISYNC` must be executed before executing any instruction from that region. In the special case of the `WITLB` changing the attribute of its own region, the `ISYNC` must immediately follow the `WITLB` and both must be within the same cache line.

After changing the attribute of a region by `WDTLB`, the operation of loads from and stores to that region are undefined until a `DSYNC` instruction is executed.

5.4 Region Translation Option

Building on the Region Protection Option is the Region Translation Option, which adds a virtual-to-physical translation on the upper three bits of the address. Thus, each of the eight 512 MB regions, in addition to the attributes provided by the Region Protection Option, may be redirected to access a different region of physical address space.

- Prerequisites: [Region Protection Option](#) on page 196
- Incompatible options: [Memory Protection Unit Option](#) on page 205, [MMU Option](#) on page 217
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

With this option, the Physical Page Numbers (PPNs) of each of the TLB entries is now writable instead of constant and identity mapped. In this way, the same region of memory may be accessed with different attributes by the use of different virtual addresses.

This simple option is built from the capabilities discussed in the introduction (see [Overview of Memory Management Concepts](#) on page 184). It uses `RingCount = 1`, so the processor can always execute privileged instructions. It sets `ASIDBits` to 0, which disables the ASID feature. The instruction and data TLBs are programmed to each have one way of eight

entries, and only the attributes and Physical Page Numbers (PPNs) are not constant; they are writable using the `WITLB` and `WDTLB` instructions.

5.4.1 Region Translation Option Architectural Additions

There are no new exceptions, no new state registers, and no new Instructions added to those in the Region Protection Option. The TLB entries contain three additional bits of state.

Access to these bits is described in [Region Translation Option Formats for Accessing TLB Entries](#) on page 203.

5.4.2 Region Translation Option Formats for Accessing TLB Entries

During normal operation when instructions and data are being accessed from memory, only lookups are being done in the TLBs. For maintenance of the TLBs, however, the entries in the TLBs are accessed by the instructions in [Region Protection Option Instruction Additions](#). Note that unused bits at Bit 12 and above are ignored on write and zero on read so that those bits may simply contain the address for access to all ways of both TLBs. Unused bits at Bit 11 and below are required to be zero on write and undefined on read for forward compatibility.

The register formats used by the TLB instructions are very similar to those described in [Formats for Accessing Region Protection Option TLB Entries](#) on page 198 for the Region Protection Option. The only difference is the presence of a Physical Page Number (PPN) in the upper three bits of the `WxTLB`, `RxTLB1`, and `PxTLB` register formats.

The format of the `as` register used in all instructions in the table is shown in [Region Translation Option Addressing \(*as*\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#). The upper three bits are used as an index among the TLB entries just as they would be when addressing memory. They are the Virtual Page Number (VPN) or upper three bits of address. The remaining bits are ignored.

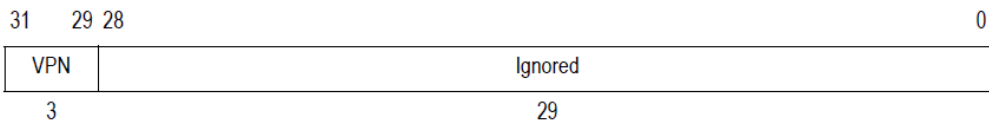


Figure 23: Region Translation Option Addressing (*as*) Format for `WxTLB`, `RxTLB1`, & `PxTLB`

The `WITLB` and `WDTLB` instructions write the TLB entries. The `as` register is formatted according to [Region Translation Option Addressing \(*as*\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#), while the `at` register is formatted according to [Region Translation Option Data \(*at*\) Format for `WxTLB`](#). The attribute for the region is described in detail in [Region Translation Option Memory Attributes](#) on page 205. The remaining bits are ignored or required to be zero.

After modifying any TLB entry with a `WITLB` instruction, an `ISYNC` must be executed before executing any instruction from that region. In the special case of the `WITLB` changing the attribute of its own region, the `ISYNC` must immediately follow the `WITLB` and both must be within the same memory region and, if the region is cacheable, within the same cache line.

After modifying any TLB entry with a `WDTLB` instruction, the operation of loads from and stores to that region are undefined until a `DSYNC` instruction is executed.

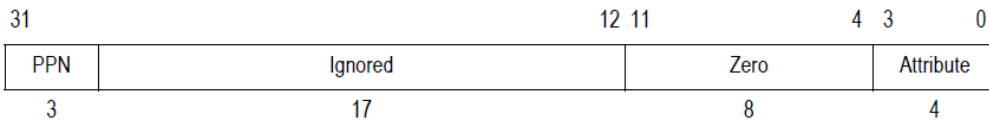


Figure 24: Region Translation Option Data (at) Format for `WxTLB`

The `RITLB0` and `RDTLB0` instructions exist under this option but do not return interesting information because the entire VPN is used as an index. The `as` register is formatted according to [Region Translation Option Addressing \(as\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#). The read instructions return zero in the `at` register.

The `RITLB1` and `RDTLB1` instructions return the `at` data format in [Region Translation Option Data \(at\) Format for `RxTLB1`](#). The attribute for the region is described in detail in [Region Translation Option Memory Attributes](#) on page 205. The Physical Page Number (PPN) is returned in the upper three bits. The remaining bits are zero or undefined. The `as` register is formatted according to [Region Translation Option Addressing \(as\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#).

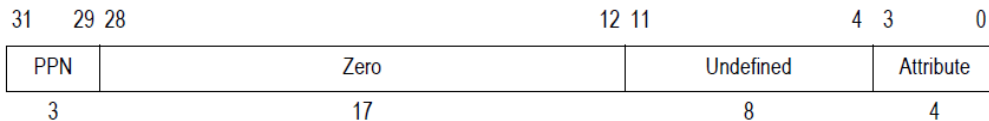


Figure 25: Region Translation Option Data (at) Format for `RxTLB1`

The `PITLB` and `PDTLB` instructions return the `at` data format in [Region Translation Option Data \(at\) Format for `PxTLB`](#). The Virtual Page Number (VPN) is returned in the upper bits. The low bit is set because the probe always hits, and the remaining bits are zero or undefined. The `as` register is formatted according to [Region Translation Option Addressing \(as\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#). These instructions work for their intended purpose, but do not provide useful information under this simple option because the TLBs always hit and have only a single way.

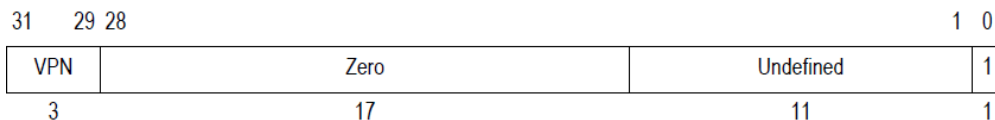


Figure 26: Region Translation Option Data (at) Format for `PxTLB`

The `IITLB` and `IDTLB` instructions exist under this option and their `as` register is formatted according to [Region Translation Option Addressing \(as\) Format for `WxTLB`, `RxTLB1`, & `PxTLB`](#), but they have no effect because the entries cannot be removed from the respective TLBs.

5.4.3 Region Translation Option Memory Attributes

The memory attributes written into the TLB entries by the `WxTLB` instructions and read from them by the `RxTLB1` instructions are exactly the same as under the Region Protection Option.

As with the Region Protection Option, all attributes in both TLBs are set to cache bypass (`4'b0010`) after reset. In addition, the translation entries in both TLBs are set to identity map after reset.

5.5 Memory Protection Unit Option

The Memory Protection Unit Option or MPU is a combined instruction and data memory protection unit with more protection flexibility than the Region Protection Option or the Region Translation Option but without any translation capability. It does no demand paging and does not reference a memory-based page table.

- Prerequisites: [Exception Option 2](#) on page 126
- Incompatible options: [Region Protection Option](#) on page 196, [MMU Option](#) on page 217, [Extended L32R Option](#) on page 86
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

This option is less closely related to the capabilities discussed in the introduction ([Overview of Memory Management Concepts](#) on page 184). It sets `ASIDBits` to 0, which disables the ASID feature. It uses `RingCount = 2` and only Ring 0 may execute privileged instructions.

One TLB is used both for instructions and for data. The TLB consists of a group of programmable foreground segments and a group of configuration determined background segments. The background segments cover all of memory while the foreground segments cover a programmable portion of memory. The portions of memory not covered by enabled foreground segments use the background segments to determine properties. Thus properties are determined for all of memory.

Two types of memory properties are provided by the MPU. One is an access rights field, which determines whether or not the current access may proceed. The other is a memory type field, which determines the system characteristics of the memory such as shareability or cacheability. The same two properties are provided by either a foreground segment or a background segment.

5.5.1 Memory Protection Unit Option Architectural Additions

[Memory Protection Unit Option Processor-Configuration Additions](#) through [Memory Protection Unit Option Instruction Additions](#) show this option's architectural additions.

Table 96: Memory Protection Unit Option Processor-Configuration Additions

Parameter	Description	Valid Values
NFOREGROUNDSEGMENTS	Number of Foreground Segments in the TLB	0-32
MINSEGMENTSIZ	Each Foreground Segment in the TLB is a multiple of this size in Bytes	32B, 64B, 128B, ... 4GB
MPULOCKABLE	MPU Entries are individually lockable and multiple hits are priority encoded if True	True, False
MPUEXECUTEONLY	Access Rights encodings include execute only encodings if True	True, False

Table 97: Memory Protection Unit Option Exception Additions (Exception Option 2)

Exception	Description	EXCCAUSE Value
LoadStoreErrorCause ¹	Inconsistent Cache Disable Bit	3
PrivilegedCause	Privileged instruction attempted with CRING ≠ 0	8
InstTLBMultiHitCause	Instruction fetch finds multiple entries in TLB	17
InstFetchProhibitedCause	Instruction fetch is not allowed in region	20
LoadStoreTLBMultiHitCause	Load/store finds multiple entries in TLB	25
LoadProhibitedCause	Load is not allowed in region	28
StoreProhibitedCause	Store is not allowed in region	29
1. Error only occurs in configurations that include the CACHEADDRDIS Special Register.		

Table 98: Memory Protection Unit Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
PS.RING	1	2 ²	Privilege level	R/W	230
MPUENB	1	NFOREGROUN D-SEGMENTS	Enables for Foreground Segments	R/W	90
MPUCFG	1	7	TLB configuration	R or R/W	92
ERACCESS	1	16	External Register Access Control	R/W	95
CACHEADDRDIS ³	1	8	Cache Region Disables	R/W	98
TLB Entries	NFOREGROUN D-SEGMENTS	32- LOG2(MINSEG- MENTSIZESIZE)	TLB entries	R/W	<i>Memory Protection Unit Option Instruction Additions</i>

1. TLB Entries are not Special Registers, but are accessed by the instructions in *Memory Protection Unit Option Instruction Additions*.
2. This field is 2 bits wide under the Exception Option 2.
3. The Special Register, `CACHEADDRDIS` does not appear in newer configurations, which do not require this power management in software.
4. Processor state is listed in *Table 127: Alphabetical List of Processor State* on page 266.

Table 99: Memory Protection Unit Option Instruction Additions

Instruction ¹	Format	Definition
PPTLB	<i>RRR</i> on page 656	Probe Protection TLB
RPTLB0	<i>RRR</i> on page 656	Read Protection TLB Entry Address
RPTLB1	<i>RRR</i> on page 656	Read Protection TLB Entry Info
WPTLB	<i>RRR</i> on page 656	Write Protection TLB

1. These instructions are fully described in *Instruction Descriptions* on page 321.

5.5.2 Memory Protection Unit Option Register Formats

This section describes the register formats of the registers in the Memory Protection Unit Option.

5.5.2.1 MPUCFG

The `MPUCFG` register is read-write if the Secure Mode Bit Option ([Secure Mode Bit Option](#)) is configured and read-only if it is not. [Memory Protection Unit Option Format for `MPUCFG`](#) shows the fields of the `MPUCFG` register. The low eight bits are always present but can never be written. They show the number of Foreground Segments configured in the MPU, which may be used by software to determine how to read and write the MPU.

In addition, if the Secure Mode Bit Option ([Secure Mode Bit Option](#)) is configured, the `NSWRDIS` bit exists as well. It controls whether the MPU can be modified in non-secure mode. If `NSWRDIS` is clear, the MPU may be changed in privileged mode (secure or non-secure). If `NSWRDIS` is set, the MPU may only be changed in secure mode. Writes to any part of the MPU, including the Enable and Lock bits, will be dropped silently in non-secure, privileged mode. `NSWRDIS` locks in the set position. Any write in privileged mode may set `NSWRDIS`, but it cannot be cleared except by reset.

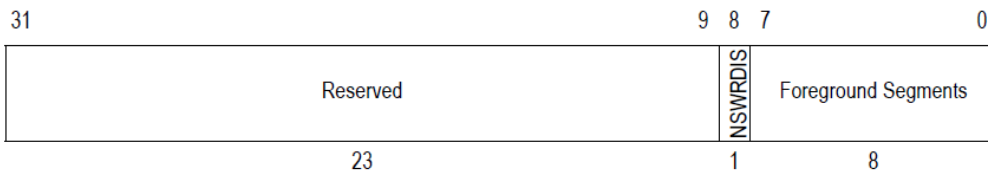


Figure 27: Memory Protection Unit Option Format for `MPUCFG`

5.5.2.2 MPUENB

This register contains a single bit for each Foreground Segment configured in the MPU. The number of valid bits of the register may be determined by reading the `MPUCFG` register. Bit zero of the register corresponds to segment zero of the MPU and so on. Each bit of `MPUENB` is also accessible using the `WPTLB` and `RPTLB0` instructions for the individual segment as indicated in [Memory Protection Unit Option Addressing \(as\) Format for `WPTLB`](#) through [Memory Protection Unit Option Data \(at\) Format for `RPTLB0`](#) below.

If the bit corresponding to a Foreground Segment is set, that segment may be used to match an address and its access rights and memory type will be used. If the bit corresponding to a Foreground Segment is clear that segment will not match. If no segment matches, the access rights and memory type of the Background Segment that corresponds to the address will be used.

The `MPUENB` register resets to zero, which causes all accesses to use Background Segments initially.

5.5.2.3 ERACCESS

This register contains a single bit for each External Register Access region used by the `WER` and `WER` instructions. The regions are always accessible when `CRING = 0` but when `CRING ≠ 0` a privileged portion of each region is not accessible and a user portion is accessible only if the corresponding bit of the `ERACCESS` register is set. It is implementation dependent which bits of the `ERACCESS` register correspond to which ERI space regions.

5.5.2.4 CACHEADRDIS

This 8-bit register contains a bit corresponding to each 512MB region of memory. It resets to zero and may be left at that value permanently. If any bits of the register are set, the logic assumes that the data cache does not need to be powered up for addresses in that 512MB region. If the MPU lookup for an access produces a cacheable memory type, a `LoadStoreErrorCause` exception will be raised.

Setting one or more bits in this register can save power by not enabling the cache during accesses either to DataRAM or to devices or other non-cacheable memory regions. If a data cache is not configured or is disabled, the exception is still raised if the memory type is cacheable. Newer configurations not containing the `CACHEADRDIS` register do not require the help of software for this power reduction.

5.5.3 The Structure of the Memory Protection Unit Option TLB

The TLB is shared between instruction accesses and data accesses and contains both a Foreground map and a Background map. The Background map is fixed at configuration time, covers all of memory, and is not defined by the ISA. It is implemented simply as combinational gates and typically takes relatively few gates. It may be used, for example, to provide access to part of memory at reset time, to provide access to a complex I/O space, or even to be the entire memory map in configurations without Foreground Segments.

Memory Protection Unit Addressing below shows an example of foreground and background maps combining to form a set of resulting regions. In each address region, the pair of letters, `aa-jj` for the foreground map and `ss-zz` for the background map, indicate a set of access rights and memory type which is applied to the address region or segment indicated by the rectangle containing them. The numbers to the left of the foreground map indicate the entry number. When more than one entry number is indicated, the two entries have the same lowest address and only the highest numbered of them is ever consulted. The shading of the rectangles indicates whether the region is a background segment of a foreground segment as well as whether or not the foreground segment is enabled.

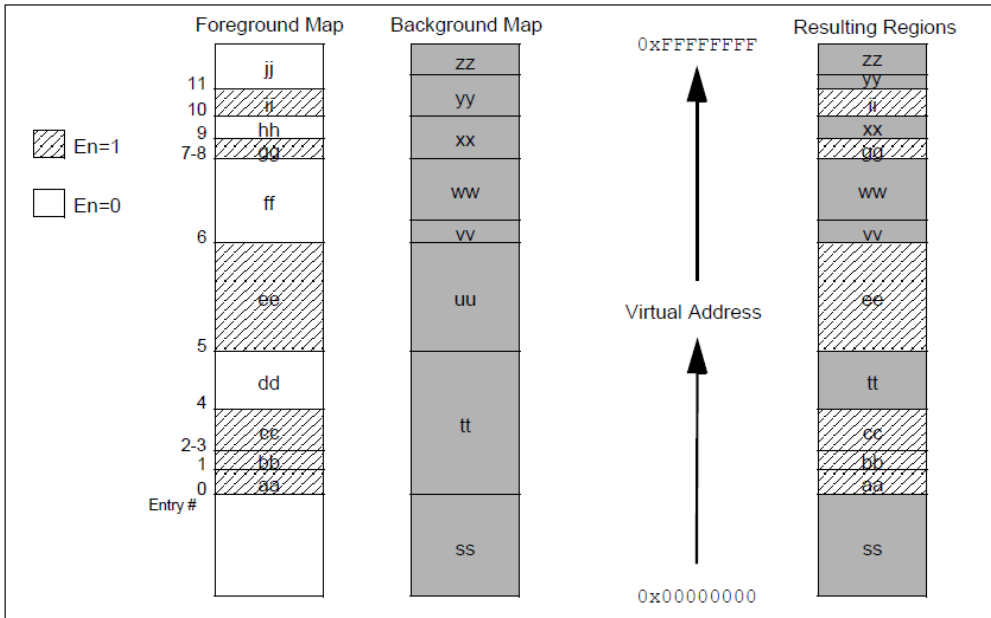


Figure 28: Memory Protection Unit Addressing

The runtime flexibility in the Memory Protection Unit Option is provided by Foreground Segments numbered from 0 to N-1, where $0 \leq N \leq 32$. Each Foreground Segment consists of a lowest address, an enable, an access rights field, and a memory type field. If the `MPULOCKABLE` parameter is False, the lowest address of each Foreground Segment must be no smaller than the lowest address of the preceding Foreground Segment in numerical order. Each enabled Foreground Segment manages protection for addresses which are greater than or equal to its own lowest address field and also less than the lowest address field of the next higher numbered Foreground Segment. Thus, if two sequential Foreground Segments contain identical lowest address fields, the lower numbered one is never used. If the `MPULOCKABLE` parameter is False and the order described is violated so that it is ambiguous which Foreground Segment should be used for a given access, then an exception will be raised on the instruction doing the access. Under the Exception Option 2, `InstTLBMultiHitCause` or `LoadStoreTLBMultiHitCause` will be raised.

If the `MPULOCKABLE` parameter is True, then the entries are not required to be in address order. If multiple enabled entries match, then the lowest numbered one will take priority and the higher numbered one(s) will be ignored. In this case, no exception is raised when multiple hits occur.

The enable bit causes the access rights and memory type fields of the Foreground Segment to be used if it is set. If no enabled entry matches, the access rights and memory type are determined from the Background map.

The chosen Foreground or Background entry has a way of providing access rights and memory type information, which will be described later (see [Memory Protection Unit Option Access Rights Field](#) on page 214 and [Memory Protection Unit Option Memory Type Field](#) on page 215).

The enables for all Foreground Segments are also available in the `MPUENB` Special Register so that several of them may be modified atomically.

5.5.4 Formats for Writing Memory Protection Unit Option TLB Entries

During normal operation when instructions and data are being accessed from memory, only lookups are being done in the TLBs. For maintenance of the TLBs, however, the entries in the TLBs are accessed by the instructions in [Memory Protection Unit Option Instruction Additions](#).

Writing the TLB with the `WPTLB` instruction requires the formats for the `as` and `at` registers shown in [Memory Protection Unit Option Addressing \(`as`\) Format for `WPTLB`](#) and [Memory Protection Unit Option Data \(`at`\) Format for `WPTLB`](#).

The format of the `as` register used for the `WPTLB` instruction is shown in [Memory Protection Unit Option Addressing \(`as`\) Format for `WPTLB`](#). The lowest bit contains the Enable bit, which will be written to the corresponding bit of the `MPUENB` register. The Lock bit may be set by software if the `MPULOCKABLE` parameter is True but is hardwired to zero if it is False. It may never be cleared by software but is only cleared by reset. When it is set, no bit of the MPU entry may be modified. In addition the Address field of the next higher numbered entry also may not be modified. The upper bits contain the upper 27 bits of the 32-bit lowest address of the segment being written. The size of this field is shown as 27 bits, which is the largest it is possible for the field to be, since `MINSEGMENTSIZ` cannot be smaller than 32-bytes. In configurations where `MINSEGMENTSIZ` is larger than 32 bytes, this field will be correspondingly smaller and left aligned.

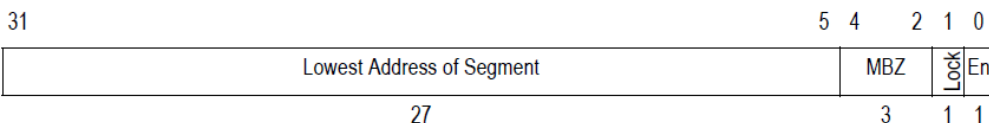


Figure 29: Memory Protection Unit Option Addressing (`as`) Format for `WPTLB`

The format of the `at` register used for the `WPTLB` instruction is shown in [Memory Protection Unit Option Data \(`at`\) Format for `WPTLB`](#). The lowest five bits contain the Segment number and may be the result of a probe instruction ([PPTLB Formats for Probing Memory Protection Unit Option TLB Entries](#) on page 213). The Acc Rights field contains the access rights for the segment as described in [Memory Protection Unit Option Access Rights Field](#) on page 214. The Memory Type field contains the memory type for the segment as described in [Memory Protection Unit Option Memory Type Field](#) on page 215.

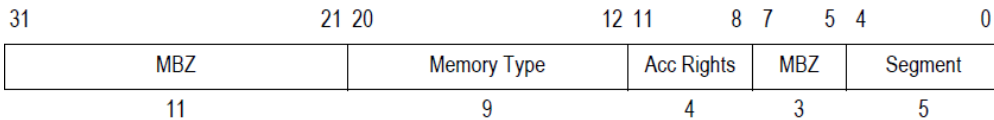


Figure 30: Memory Protection Unit Option Data (at) Format for WPTLB

MBZ means the bits must be zero on write. If the given segment number is `NFOREGROUNDSEGMENTS` or greater, the instruction will be a NOP. After modifying any TLB entry with a `WPTLB` instruction, no sync instruction is needed before use of the entry.

5.5.5 Formats for Reading Memory Protection Unit Option TLB Entries

Reading the TLB with the `RPTLB0` and `RPTLB1` instructions requires the formats for the `as` and `at` registers shown in [Memory Protection Unit Option Addressing \(as\) Format for RPTLB0 and RPTLB1](#) through [Memory Protection Unit Option Data \(at\) Format for RPTLB1](#). These figures show, in parallel, the formats for different ways of the cache and different conditions.

The format of the `as` register used for the `RPTLB0` and `RPTLB1` instructions is shown in [Memory Protection Unit Option Addressing \(as\) Format for RPTLB0 and RPTLB1](#). The low order five bits contain the segment to be accessed. They may be the result of the probe instruction ([PPTLB Formats for Probing Memory Protection Unit Option TLB Entries](#) on page 213).

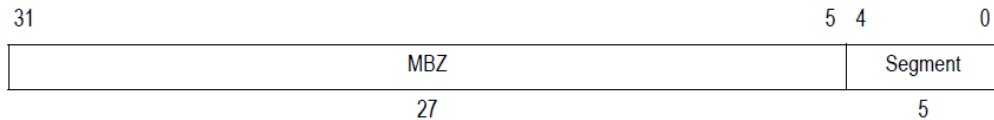


Figure 31: Memory Protection Unit Option Addressing (as) Format for RPTLB0 and RPTLB1

Because reading generates more information than can fit in one 32-bit register, there are two read instructions that return different values. The data resulting from the `RPTLB0` instruction is shown in [Memory Protection Unit Option Data \(at\) Format for RPTLB0](#). The low bit contains the Enable bit from the corresponding bit of the `MPUENB` register. The next bit contains the value of the Lock bit. The upper bits contain the upper 27 bits of the 32-bit lowest address of the segment. The size of this address field is shown as 27 bits, which is the largest it is possible for the field to be, since `MINSEGMENTSIZ` cannot be smaller than 32-bytes. In configurations where `MINSEGMENTSIZ` is larger than 32 bytes, this field will be correspondingly smaller and left aligned.

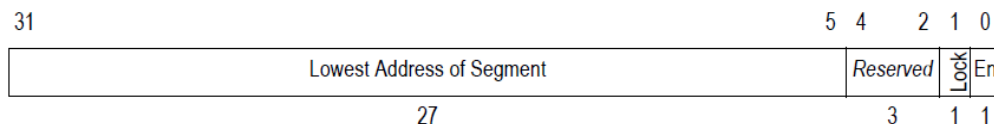


Figure 32: Memory Protection Unit Option Data (at) Format for RPTLB0

The data resulting from the `RPTLB1` instruction is shown in [Memory Protection Unit Option Data \(*at*\) Format for `RPTLB1`](#). The Acc Rights field contains the access rights for the segment as described in ([Memory Protection Unit Option Memory Type Field](#) on page 215). The Memory Type field contains the memory type for the segment as described in ([Memory Protection Unit Option Memory Type Field](#) on page 215).

If the given segment number is `NFOREGROUNDSEGMENTS` or greater, the instruction will read all zeros.

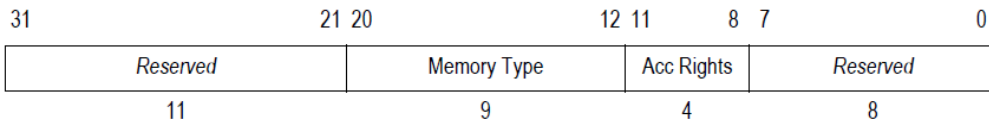


Figure 33: Memory Protection Unit Option Data (*at*) Format for `RPTLB1`

5.5.6 Formats for Probing Memory Protection Unit Option TLB Entries

Probing the TLB with the `PPTLB` instruction requires the formats for the `as` and `at` registers shown in [Memory Protection Unit Option Addressing \(*as*\) Format for `PxTLB`](#) and [Memory Protection Unit Option Data \(*at*\) Format for `PPTLB`](#). Unlike writing and reading the TLBs as explained in the previous two sections, the operation of probing a TLB begins without knowing the segment containing the sought after value. The probe instruction answers the question of what segment in this TLB, if any, would be used to translate an access with a particular address. The sought for address is given in the `as` register as shown in [Memory Protection Unit Option Addressing \(*as*\) Format for `PxTLB`](#).

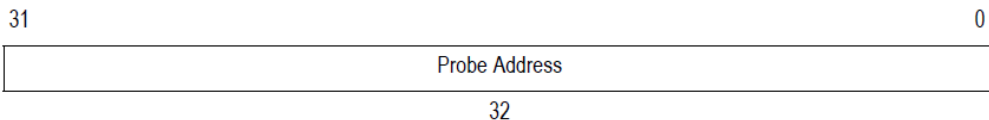


Figure 34: Memory Protection Unit Option Addressing (*as*) Format for `PxTLB`

The data resulting from the `PPTLB` instruction is shown in [Memory Protection Unit Option Data \(*at*\) Format for `PPTLB`](#). The Acc Rights field contains the access rights that will be used in protecting the address supplied in `as`. The Memory Type field contains the memory type that will be used in protecting the address supplied in `as`. If a Foreground Segment was used for the address in `as`, the V bit will be set, the B bit will be clear and the Segment field will contain the number of the segment used. If the Background region below Foreground Segment #0 was used, the V bit will be clear, the B bit will be set and the Segment field will be undefined. If the Background region was used other than below Foreground Segment #0, both the V and B bits will be clear and, if the `MPULOCKABLE` parameter is False, the Segment field will indicate the Foreground Segment that matched the address range but was not enabled.

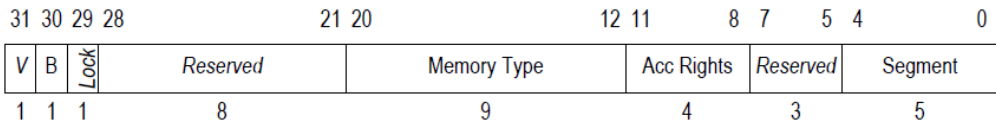


Figure 35: Memory Protection Unit Option Data (a_t) Format for PPTLB

5.5.7 Memory Protection Unit Option Access Rights Field

The meaning of values in the Access Rights field is shown below in [Memory Protection Unit Option Access Rights](#). The first column is the number in the four-bit field labeled Access Rights and the other columns show whether privileged and unprivileged accesses of type read, write, and execute are permitted. When an access is attempted which is not permitted or which is reserved, an exception will be raised. Under the Exception Option 2, the cause value will be `LoadProhibitedCause`, `StoreProhibitedCause`, or `InstFetchProhibitedCause` depending on whether the attempted access was read, write, or execute.

Table 100: Memory Protection Unit Option Access Rights

Field Value	Access when PS.RING==0 ¹			Access when PS.RING==1 ¹		
	Read	Write	Execute	Read	Write	Execute
0	—	—	—	—	—	—
1	<i>Reserved</i>					
2 ²	—	—	Yes	—	—	—
3 ²	—	—	—	—	—	Yes
4	Yes	—	—	—	—	—
5	Yes	—	Yes	—	—	—
6	Yes	Yes	—	—	—	—
7	Yes	Yes	Yes	—	—	—
8	—	Yes	—	—	Yes	—
9	Yes	Yes	—	Yes	Yes	Yes
10	Yes	Yes	—	Yes	—	—

Field Value	Access when PS . RING==0 ¹			Access when PS . RING==1 ¹		
	Read	Write	Execute	Read	Write	Execute
11	Yes	Yes	Yes	Yes	—	Yes
12	Yes	—	—	Yes	—	—
13	Yes	—	Yes	Yes	—	Yes
14	Yes	Yes	—	Yes	Yes	—
15	Yes	Yes	Yes	Yes	Yes	Yes

1. Operation with PS . RING==2 or PS . RING==3 is undefined.
2. The two Execute Only encodings are available only when the MPUEXECUTEONLY parameter is True. If the parameter is False, the encodings are Reserved.

Separately, if PS . RING is 1 and execution of a privileged instruction is attempted, an exception of type `PrivilegedCause` will be raised under the Exception Option 2 (see [Exception Causes](#)).

5.5.8 Memory Protection Unit Option Memory Type Field

The meaning of the values in the Memory Type field is shown below in [Memory Protection Unit Option Memory Type](#). The first column is the number in the field labeled Memory Type and the other columns show some of the principal characteristics of the memory type.

Shareable regions are regions where hardware ensures that multiple masters can correctly share the memory in that region. Non-shareable regions may only be accessed by a single master. Multiple threads on the same master may share a non-shareable region.

When an interrupt arrives during a non-interruptible device load, the processor will wait to receive the load value from the system, complete the load instruction, and then process the interrupt. In addition, non-interruptible device loads will not be speculated. These characteristics are required to properly read devices that have read side effects.

When an interrupt arrives during an interruptible device load, the load will complete on the bus, the value will be thrown away, and after the interrupt, the load will repeat. This capability is required for best interrupt latency. These loads may also be speculated.

Loads from memory (as opposed to device space) and all instruction fetches will freely be interrupted and retried and may be speculated as memory is assumed to have no side effects. All stores happen exactly once and are not speculated. Interrupt latency is not affected.

Put differently, stores and non-interruptible loads are "guarded" while instruction fetches and interruptible loads are not guarded.

Additional characteristics of the memory type are indicated by letters used in the field value column indicating either a 0 or a 1. The meaning of these letters is described below.

Table 101: Memory Protection Unit Option Memory Type

Field Value ¹	System Type	Shareable	Interruptible Load	Cached in Data or Instruction Cache
00_000_000B ²	Device	Yes	No	No
00_000_011B			Yes	
00_000_100B ²				
00_000_111B				
00_001_100B	Non-Cacheable Memory	No	Yes	No
01_rwc_1001		Yes	Yes	Yes ³
00_001_111B				No
00_001_0RWC	Cacheable Memory	No		Yes
01_rwc_0RWC		Yes	Yes	
00_011_IRWC			No	
11_rwc_IRWC		Yes	Yes ⁴	

1. Letters in values are explained in text. All values not listed are reserved and raise an InstFetchProhibitedCause, LoadProhibitedCause, or StoreProhibitedCause exception depending on access type under the Exception Option 2 .
2. Exclusive loads and stores are sent with a zero AxLOCK field.
3. Care must be exercised in using this value as data is requested from the system as non-cacheable but yet is cached in the processor.
4. Shareable Cacheable regions only cache data in the processor when full hardware coherence is both configured and enabled.

The following letters are used in *Memory Protection Unit Option Memory Type* above:

- **b**: If the system bus supports the concepts, the bus transaction resulting from this access will indicate non-Bufferable if the bit is clear and Bufferable if the bit is set.
- **c**: If the system bus supports the concepts, any bus transaction resulting from this access will indicate Write-through cacheable if the bit is clear and Writeback cacheable if the bit is set.
- **w**: If the system bus supports the concepts, any bus transaction resulting from this access will indicate no-Write-allocate if the bit is clear and Write-allocate if the bit is set.
- **r**: If the system bus supports the concepts, any bus transaction resulting from this access will indicate no-Read-allocate if the bit is clear and Read-allocate if the bit is set.
- **i**: If the system bus supports the concepts, any bus transaction resulting from this access will indicate Outer-shareable if the bit is clear and Inner-shareable if the bit is set.
- **c**: The processor will cache this in its internal data cache in Write-through mode if the bit is clear and in Write-back mode if the bit is set. If the processor is not capable of the indicated mode, it will not cache the line. If the System Type is Cacheable Memory and Shareable, then the processor cache only caches data if full hardware coherence is configured and enabled.
- **w**: The processor will use this bit as a hint that its internal data cache should not allocate on write if the bit is clear and should allocate on write if the bit is set. The processor is not required to follow this hint.
- **r**: The processor will use this bit as a hint that its internal instruction and data caches should not allocate on read if the bit is clear and should allocate on read if the bit is set. The processor is not required to follow this hint.

Regions marked as Shareable in *Memory Protection Unit Option Memory Type* are correctly shared with other masters in all systems. Those which do not have cacheable memory types perform each access externally and mark the transaction as non-cacheable so that sharing can occur in main memory or in a cache common to all masters. Those which do have cacheable memory types use cache coherence if it is configured and enabled.

If coherence is not configured or not enabled, the processor internal caches refuse to cache accesses to Shareable regions. The system can then make them shared. Such accesses will be marked cacheable and shareable on the system bus such that a coherent cache outside the processor may hold them for improved performance. An external cache for which coherence is not configured or not enabled may also refuse to cache regions labeled shareable so that they may be shared in main memory or in a cache common to all masters.

5.6 MMU Option

The MMU Option is a memory management unit created to run protected operating systems such as Linux on the Xtensa processor with demand paging hardware with a memory-based page table.

- Prerequisites: *Exception Option 2* on page 126

- Incompatible options: [Region Protection Option](#) on page 196, [Memory Protection Unit Option](#) on page 205, [Extended L32R Option](#) on page 86
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

This option is also built from the capabilities discussed in the introduction ([Overview of Memory Management Concepts](#) on page 184). It uses `RingCount = 4` and only Ring 0 may execute privileged instructions. The option sets `ASIDBits` to 8, which allows for lower TLB management overhead.

The instruction and data TLBs are programmed to have seven and ten ways, respectively (see [The Structure of the MMU Option TLBs](#) on page 224). Some of the ways set only by software or can be constants; others auto-refill from a page table in memory that contains 4-byte PTEs, each mapping a 4kB page with a 20-bit PPN, a 2-bit ring number, an attribute, and bits reserved for software. For a programmer's view of the MMU, refer to the *Xtensa Microprocessor Programmer's Guide*.

5.6.1 MMU Option Architectural Additions

[MMU Option Processor-Configuration Additions](#) through [MMU Option Instruction Additions](#) show this option's architectural additions.

Table 102: MMU Option Processor-Configuration Additions

Parameter	Description	Valid Values
EXTMEMATTRIBUTES	Extended memory attributes are enabled if true. The Extended memory attributes are oriented toward AMBA bus attributes.	False, True
ALTPAGESIZES	Alternate page sizes determines which page sizes are available in the implementation.	0, 1
NIREFILLENTRIES	Number of auto-refill entries in the ITLB (divided among 4 ways)	16, 32 (4, 8 entries per TLB way)
NDREFILLENTRIES	Number of auto-refill entries in the DTLB (divided among 4 ways)	16, 32 (4, 8 entries per TLB way)
IVARWAY56	Ways 5&6 of the ITLB can be variable for greater flexibility in mapping memory	Variable or Fixed ¹

Parameter	Description	Valid Values
DVARWAY56	Ways 5&6 of the DTLB can be variable for greater flexibility in mapping memory	Variable or Fixed ¹
IWAY4INDXCNT	Number of entries in ITLB way 4	4, 8
DWAY4INDXCNT	Number of entries in DTLB way 4	4, 8
IWAY5INDXCNT	Number of entries in ITLB way 5	4, 8
DWAY5INDXCNT	Number of entries in DTLB way 5	4, 8
DWAY7INDXCNT	Number of entries in DTLB way 7	1, 4
DWAY8INDXCNT	Number of entries in DTLB way 8	1, 4
DWAY9INDXCNT	Number of entries in DTLB way 9	1, 4
DUALMEMACCESS	Processor is capable of supporting two load/store accesses per instruction	False, True

1. Implementations may allow only Fixed, only Variable or a choice of either for this value.

Table 103: MMU Option Exception Additions

Exception	Description	EXCCAUSE Value
PrivilegedCause	Privileged instruction attempted with CRING ≠ 0	8
InstTLBMissCause	Instruction fetch finds no entry in ITLB	16
InstTLBMultiHitCause	Instruction fetch finds multiple entries in ITLB	17
InstFetchPrivilegeCause	Instruction fetch matching entry requires lower CRING	18
InstFetchProhibitedCause	Instruction fetch is not allowed in region	20
LoadStoreTLBMissCause	Load/store finds no entry in DTLB	24

Exception	Description	EXCCAUSE Value
LoadStoreTLBMultiHitCause	Load/store finds multiple entries in DTLB	25
LoadStorePrivilegeCause	Load/store matching entry requires lower CRING	26
LoadProhibitedCause	Load is not allowed in region	28
StoreProhibitedCause	Store is not allowed in region	29

Table 104: MMU Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
PS.RING	1	2	Privilege level (see <i>PS Register Fields</i>)	R/W	230
PTEVADDR	1	32	Page Table Virtual Address	R/W	83
RASID	1	32	Per-ring ASIDs	R/W	90
ITLBCFG	1	2/4	Instruction TLB configuration	R/W	91
DTLBCFG	1	2/4	Data TLB configuration	R/W	92
ERACCESS	1	16	External Register Access Control	R/W	95
ITLB Entries	Variable ²	variable	Instruction TLB entries	R/W	<i>MMU Option Instruction Additions</i>
DTLB Entries	Variable ²	variable	Data TLB entries	R/W	<i>MMU Option Instruction Additions</i>
VADDRSTATUS	1	2	Exists when DUALMEMACCESS=True. On an MMU exception,	R/W	84

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ⁷
			this contains a bit for each possible access indicating whether it had a valid access. Bit[0] corresponds to VADDR0 and Bit[1] corresponds to VADDR1 ³		
VADDR0	1	32	Exists when DUALMEMACCE SS=True. On an MMU exception, this contains the address of one of accesses. VADDR0 [11:0] is hardwired to zero. ³	R/W	85
VADDR1	1	32	Exists when DUALMEMACCE SS=True. On an MMU exception, this contains the address of the other access. VADDR1 [11:0] is hardwired to zero. ³	R/W	86

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` (*Processor Control Instructions*). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266. The TLB Entries are not Special Registers, but are accessed by the instructions in *MMU Option Instruction Additions*.
2. See [The Structure of the MMU Option TLBs](#) on page 224 for more information on TLB structure.
3. The three registers beginning with VADDR are added when there can be two memory accesses in the same instruction so that the MMU miss handler can ensure that both accesses will succeed together and avoid the possibility that each might repeatedly replace the other's translation.

Table 105: MMU Option Instruction Additions

Instruction ¹	Format	Definition
IDTLB	RRR on page 656	Invalidate data TLB entry
IITLB	RRR on page 656	Invalidate instruction TLB entry
PDTLB	RRR on page 656	Probe data TLB
PITLB	RRR on page 656	Probe instruction TLB
RDTLB0	RRR on page 656	Read data TLB virtual
RDTLB1	RRR on page 656	Read data TLB Translation
RITLB0	RRR on page 656	Read instruction TLB virtual
RITLB1	RRR on page 656	Read instruction TLB translation
WDTLB	RRR on page 656	Write data TLB
WITLB	RRR on page 656	Write instruction TLB
1. These instructions are fully described in Instruction Descriptions on page 321.		

5.6.2 MMU Option Register Formats

This section describes the address and data formats needed for reading and writing the instruction and data TLBs.

5.6.2.1 PTEVADDR

Because four ways of each TLB are configured as AutoRefill, the MMU Option supports hardware refill of the TLB from a page table ([MMU Option Auto-Refill TLB Ways and PTE Format](#) on page 231). The base virtual address of the current page table is specified in the PTEBase field of the PTEVADDR register. When read, PTEVADDR returns the PTEBase field in its upper bits as shown in [MMU Option PTEVADDR Register Format](#), EXCVADDR31..12 in the field labeled VPN below followed by two zero bits. When PTEVADDR is written, only the PTEBase field is modified. PTEVADDR is undefined after reset. [MMU Option PTEVADDR Register Format](#) shows the PTEVADDR register format.

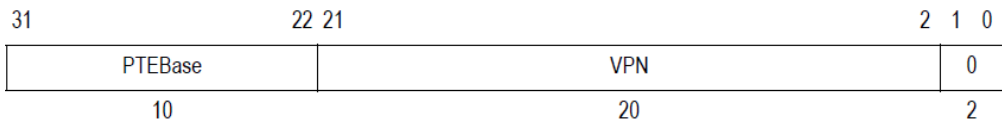


Figure 36: MMU Option PTEVADDR Register Format

5.6.2.2 RASID

The Ring ASID ($RASID$) register holds the current ASIDs for each ring. The register is divided into four 8-bit sections, one for each ASID. The Ring 0 ASID is hardwired to 1. The operation of the processor is undefined if any two of the *four* ASIDs are equal or if it contains an ASID of zero. $RASID$ is $32'h04030201$ after reset. *MMU Option RASID Register Format* shows the $RASID$ register format.

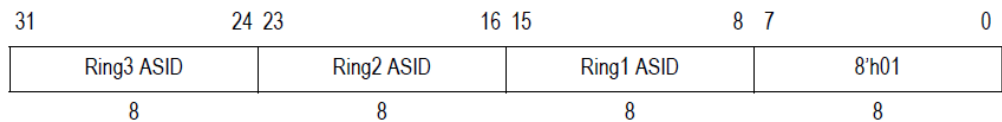


Figure 37: MMU Option RASID Register Format

5.6.2.3 ITLBCFG

Because some ways of the instruction TLB are configured with variable page sizes, the $ITLBCFG$ register specifies the page size for those ways. The bits of $ITLBCFG$ that change the available page sizes as described in *The Structure of the MMU Option TLBs* on page 224 are variable while all other bits are hardwired to zero. When the size of pages in a way is changed, the entire way should be invalidated or overwritten. The register is zero after reset.

5.6.2.4 DTLBCFG

Because some ways of the data TLB are configured with variable page sizes, the $DTLBCFG$ register specifies the page size for those ways. The bits of $DTLBCFG$ that change the available page sizes as described in *The Structure of the MMU Option TLBs* on page 224 are variable while all other bits are hardwired to zero. When the size of pages in a way is changed, the entire way should be invalidated or overwritten. The register is zero after reset.

5.6.2.5 ERACCESS

This register contains a single bit for each External Register Access region used by the RER and WER instructions. The regions are always accessible when $CRING = 0$ but when $CRING \neq 0$ a privileged portion of each region is not accessible and a user portion is accessible only if the corresponding bit of the $ERACCESS$ register is set. It is implementation dependent which bits of the $ERACCESS$ register correspond to which ERI space regions.

5.6.3 The Structure of the MMU Option TLBs

The instruction TLB is 7-way set-associative. Ways 0-3 are AutoRefill ways used for hardware refill of 4 kB page table entries from the page table when no matching TLB entry is found. Higher numbered ways are either fixed or loaded explicitly.

The data TLB is 10-way set-associative and has three additional ways. Again ways 0-3 are AutoRefill ways used for hardware refill of 4kB page table entries and higher numbered ways are fixed or loaded explicitly.

MMU Option Page Sizes and Entry Counts shows the available page sizes and entry counts for each way as determined by the configuration parameter `ALTPAGESIZES` along with other configuration parameters referenced in the table. The lower case 'x' in configuration parameters is replaced by 'I' for the `ITLB` and 'D' for the `DTLB`.

Table 106: MMU Option Page Sizes and Entry Counts

ALTPAGESIZES	Way		Page Size	Entry Count
0	0		4kB	$N \times \text{REFILLENTRIE} / 4$
	1		4kB	$N \times \text{REFILLENTRIE} / 4$
	2		4kB	$N \times \text{REFILLENTRIE} / 4$
	3		4kB	$N \times \text{REFILLENTRIE} / 4$
	4		From $x\text{TLBCFG}[17:16]$: 0 → 1MB 1 → 4MB 2 → 16MB 3 → 64MB	4
	5		From $(x\text{VARWAY56}, x\text{TLBCFG}[20])$: (False, 0) → 128MB - Fixed [†] (False, 1) → 128MB - Fixed [†] (True, 0) → 128MB (True, 1) → 256MB	From $(x\text{VARWAY56}, x\text{TLBCFG}[20])$: (False, 0) → 2 - Fixed [†] (False, 1) → 2 - Fixed [†] (True, 0) → 4

ALTPAGESIZES	Way		Page Size	Entry Count
				(True, 1) → 4
	6	From (xVARWAY56, xTLBCFG[24]): (False, 0) → 256MB - Fixed ² (False, 1) → 256MB - Fixed ² (True, 0) → 512MB (True, 1) → 256MB		From (xVARWAY56, xTLBCFG[24]): (False, 0) → 2 - Fixed ² (False, 1) → 2 - Fixed ² (True, 0) → 8 (True, 1) → 8
	7	4kB		1
	8	4kB		1
	9	4kB		1
1	0	4kB		8
	1	4kB		8
	2	4kB		8
	3	4kB		8
	4	From xTLBCFG[16]: 0 → 256kB 1 → 4MB		From xWAY4INDXCNT: 4 → 4 8 → 8
	5	From xTLBCFG[20]: 0 → 4MB 1 → 256kB		From xWAY5INDXCNT: 4 → 4 8 → 8
	6	From xTLBCFG[25:24]: 0 → 512MB 1 → 256MB 2 → 256kB 3 → 4MB		8

ALTPAGESIZES	Way		Page Size	Entry Count
	7		4kB	From DWAY7INDXCNT: 1→ 1 4→ 4
	8		4kB	From DWAY8INDXCNT: 1→ 1 4→ 4
	9		4kB	From DWAY9INDXCNT: 1→ 1 4→ 4
2-15			Reserved	
<ol style="list-style-type: none"> 1. These fixed entries map 128MB at 0xD000_0000 and 0xD800_0000 to 0x0000_0000 as cached (4'h7) and cache bypass (4'h3) respectively. Both use ASID=0x01. 2. These fixed entries map 256MB at 0xE000_0000 and 0xF000_0000 to 0xF000_0000 as cached (4'h7) and cache bypass (4'h3) respectively. 				

After reset, all ways except Way-6 and, if it is not variable, Way-5 are set to invalid by having their ASID values set to zero. If Way-6 is variable, it is set after reset to cover all of memory in eight 512MB chunks. The ASID values are all set to 0x01. The Attribute fields are set to 4'h3 (Bypass) for EXTMEMATTRIBUTES = False. The Memory Type is set to 0x00 (Unbuffered Device) and all memory has read, write, and execute privileges for EXTMEMATTRIBUTES = True.

5.6.4 The MMU Option Memory Map

The memory map is determined by the TLB configurations given in [The Structure of the MMU Option TLBs](#) on page 224. [MMU Option Address Map with IVARWAY56 and DVARWAY56 Fixed](#) shows a graphical representation of the constant translations in Way 5 and Way 6 when ALTPAGESIZES is 0 and IVARWAY56 and DVARWAY56 are Fixed, as well as the regions that are mapped by more flexible ways than these. Way 5 and Way 6 may be used to emulate this same arrangement when IVARWAY56 and DVARWAY56 are Variable.

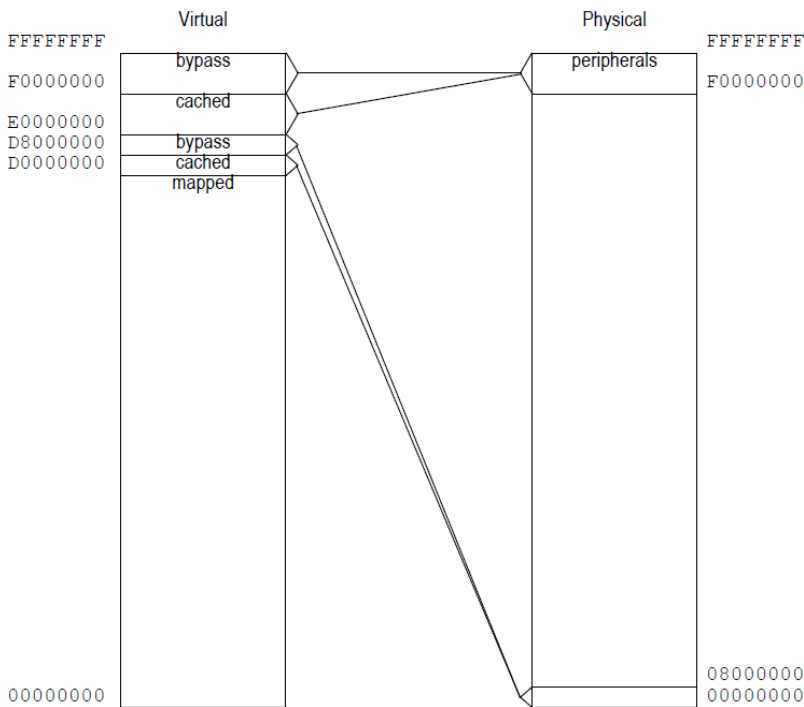


Figure 38: MMU Option Address Map with IVARWAY56 and DVARWAY56 Fixed

This configuration provides both bypass and cached access to peripherals. Bypass access is used for devices and cached access is used for ROMs, for example. It also provides bypass and cached access to the low 128 MB of memory. This allows system software to access its memory without competing with user code for other TLB entries. These are available after reset. The large page way (Way 4) and the auto-refill ways (Ways 0-3) may be used to map as much additional space as desired ([MMU Option Auto-Refill TLB Ways and PTE Format](#) on page 231). In the data TLB, Ways 7-9 may be used to map single pages so that they are always available.

5.6.5 Formats for Writing MMU Option TLB Entries

During normal operation when instructions and data are being accessed from memory, only lookups are being done in the TLBs. For maintenance of the TLBs, however, the entries in the TLBs are accessed by the instructions in [MMU Option Instruction Additions](#).

Writing the TLB with `WITLB` and `WDTLB` instructions requires the formats for the `as` and `at` registers shown in [MMU Option TLB Write Formats](#). The number of bits required for the VPN and PPN fields vary with the page size and bits below $\ln_2(\text{PageSize})$ are ignored in the field. The VPN field contains both index and tag bits for the entry assigned depending on the entry count in the way.

The ring in bits[5:4] of register `at` is used to choose one of the `ASID` values from the `RASID` register. It is not possible to write any `ASID` which is not currently in the `RASID` register.

For forward compatibility, the reserved bits in register `as` may be either zero or the result of a probe instruction. For forward compatibility, the reserved bits in register `at` must be zero. If `ALTPAGESIZES=0` and `xVARWAY56=Fixed`, the `WxTLB` instruction has no effect. If the Way Number is too large for the TLB being written, the result is undefined.

Table 107: MMU Option TLB Write Formats

EXTMEMATTRIBUTES	Register <code>as</code> (source)	Register <code>at</code> (source)
False	[31:12]: PPN - bits below $\ln 2(\text{PageSize})$ are ignored [11:4]: Reserved [3:0]: Way Number to be written	[31:12]: VPN - bits below $\ln 2(\text{PageSize})$ are ignored [11:6]: Reserved - MBZ [5:4]: Ring (chooses <code>ASID</code> from <code>RASID</code>) [3:0]: Attribute
True	[31:12]: VPN - bits below $\ln 2(\text{PageSize})$ are ignored [11:4]: Reserved [3:0]: Way Number to be written	[31:12]: PPN [11:8,3:2]: Memory Type [7:6]: Reserved - MBZ [5:4]: Ring (chooses <code>ASID</code> from <code>RASID</code>) [1,0]: W,X

After modifying any TLB entry with a `WITLB` instruction, an `ISYNC` must be executed before executing any instruction that depends on the modification. The ITLB entry currently being used for instruction fetch may not be changed.

After modifying any TLB entry with a `WDTLB` instruction, the operation of loads and stores that depend on that TLB entry are undefined until a `DSYNC` instruction is executed.

5.6.6 Formats for Reading MMU Option TLB Entries

Reading the TLB with the `RITLB0`, `RITLB1`, `RDTLB0`, and `RDTLB1` instructions requires the formats for the `as` and `at` registers shown in [MMU Option TLB Read Formats](#). The entry read depends only on the index portion of the VPN and so bits above and below those in register `as` are ignored. Note that these index bits are not returned by `RxTLB0`. Bits below $\ln 2(\text{PageSize})$ in PPN are returned as zeros.

For forward compatibility, the reserved bits in register `as` may be either zero or the result of a probe instruction. If the Way Number is too large for the TLB being written, the result is undefined.

Table 108: MMU Option TLB Read Formats

EXTMEM-ATTRIBUTES	Register <i>as</i> (source)	Register <i>at</i> (destination)
False	<p>[31:12]: VPN - bits below $\ln_2(\text{PageSize})$ and at or above $\ln_2(\text{PageSize} * \text{EntryCount})$ are ignored</p> <p>[11:4]: Reserved</p> <p>[3:0]: Way Number to be read</p>	<p>For $R_x\text{TLB}_0$:</p> <p>[31:12]: VPN - bits below $\ln_2(\text{PageSize} * \text{EntryCount})$ are zero</p> <p>[11:8]: Undefined</p> <p>[7:0]: ASID</p> <p>For $R_x\text{TLB}_1$:</p> <p>[31:12]: PPN - bits below $\ln_2(\text{PageSize})$ are zero</p> <p>[11:4]: Undefined</p> <p>[3:0]: Attribute</p>
True	<p>[31:12]: VPN - bits below $\ln_2(\text{PageSize})$ and at or above $\ln_2(\text{PageSize} * \text{EntryCount})$ are ignored</p> <p>[11:4]: <i>Reserved</i></p> <p>[3:0]: Way Number to be written</p>	<p>For $R_x\text{TLB}_0$:</p> <p>[31:12]: VPN - bits below $\ln_2(\text{PageSize} * \text{EntryCount})$ are zero</p> <p>[11:8]: Undefined</p> <p>[7:0]: ASID</p> <p>For $R_x\text{TLB}_1$:</p> <p>[31:12]: PPN - bits below $\ln_2(\text{PageSize})$ are zero</p> <p>[11:8,3:2]: Memory Type</p> <p>[7:4]: Undefined</p> <p>[1,0]: W,X</p>

5.6.7 Formats for Probing MMU Option TLB Entries

Probing the TLB with the `PITLB` and `PDTLB` instructions requires the formats for the `as` and `at` registers shown in [MMU Option Addressing \(*as*\) Format for \$P_x\text{TLB}\$](#) and [MMU Option Data \(*at*\) Format for \$PITLB\$](#) . Unlike writing and reading the TLBs as explained in the previous two sections, the operation of probing a TLB begins without knowing the way containing the sought after value. The formats do not, therefore, vary with the way being accessed. The probe instructions answer the question of what entry in this TLB, if any, would be used to translate an access with a particular address from a particular ring. The sought for address is given in the `as` register as shown in [MMU Option Addressing \(*as*\) Format for \$P_x\text{TLB}\$](#) and the ring is given by `PS.RING` (not `CRING`, so that while `PS.EXCM` is set, a probe may be done for a user program). If, for example, there is an entry that matches in address, but its `ASID` does not match any `ASID` in the `RASID` register, or an entry that matches in address, but the `ASID`

corresponds in the `RASID` register to a ring of lower number than the current `PS.RING`, the probe will not return a hit.

The format of the `as` register used for the `PITLB` and `PDTLB` instructions is shown in [MMU Option Addressing \(*as*\) Format for `PxTLB`](#). Any address may be used as input to the probe instructions.

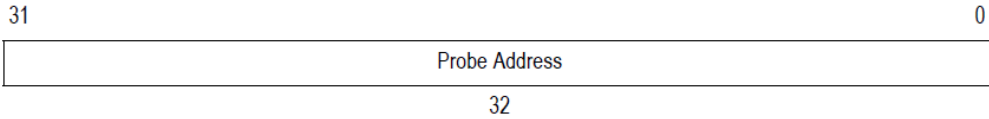


Figure 39: MMU Option Addressing (*as*) Format for `PxTLB`

The data resulting from the `PITLB` and `PDTLB` instructions is shown in [MMU Option Data \(*at*\) Format for `PITLB`](#) and [MMU Option Data \(*at*\) Format for `PDTLB`](#). The low three/four bits contain the Way (if any), which would be used to translate the address and the next bit up is set if there is a translation in the TLB, and clear if there is not. Some bits are undefined for forward compatibility but the result is such that, if `Hit=1`, it may be used as the `as` register for `WxTLB`, `RxTLB0`, `RxTLB1`, or `IxTLB`.

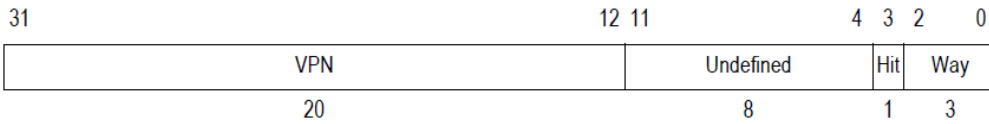


Figure 40: MMU Option Data (*at*) Format for `PITLB`

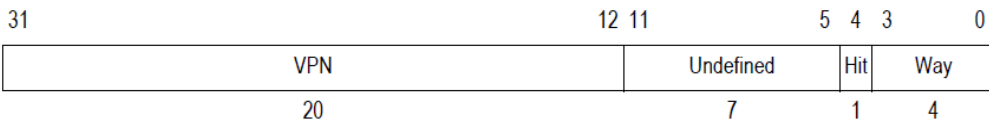


Figure 41: MMU Option Data (*at*) Format for `PDTLB`

5.6.8 Format for Invalidating MMU Option TLB Entries

Invalidating the TLB with the `IITLB` and `IDTLB` instructions requires the formats for the `as` register shown in [MMU Option TLB Invalidate Formats](#). The number of bits required for the VPN field varies with the page size and bits below $\ln_2(\text{Page Size})$ and bits at or above $\ln_2(\text{Page Size} * \text{Entry Count})$ are ignored in the field.

For forward compatibility, the reserved bits in register `as` may be either zero or the result of a probe instruction. If `ALTPAGESIZES=0` and `xVARWAY56=Fixed`, the `IxTLB` instruction has no effect. If the Way Number is too large for the TLB being written, the result is undefined.

Invalidation of an entry sets the corresponding `ASID` to zero so that it no longer responds when an address is looked up in the TLB.

Table 109: MMU Option TLB Invalidate Formats

EXTMEM-ATTRIBUTES	Register as (source)	Register at (unused)
Any	[31:12]: VPN - bits below $\ln_2(\text{PageSize})$ and at or above $\ln_2(\text{PageSize} * \text{EntryCount})$ are ignored [11:4]: Reserved [3:0]: Way Number to be invalidated	

After invalidating any TLB entry with an `IITLB` instruction, an `ISYNC` must be executed before executing any instruction that depends on the modification. The ITLB entry currently being used for instruction fetch may not be invalidated.

After invalidating any TLB entry with a `IDTLB` instruction, the operation of loads and stores that depend on that TLB entry are undefined until a `DSYNC` instruction is executed.

5.6.9 MMU Option Auto-Refill TLB Ways and PTE Format

When no TLB entry matches the ASIDs and the virtual address presented to the MMU, the MMU attempts to automatically load the appropriate page table entry (PTE) from the page table and write it into the TLB in one of the AutoRefill ways. This hardware-generated load from the page table itself requires virtual-to-physical address translation, which executes at Ring 0 so that it has access to the page table and uses the DTLB. An error of any sort during the automatic refill process will cause an `InstTLBMissCause` or a `LoadStoreTLBMissCause` exception to be raised so that system software can take appropriate action and possibly retry the access. This combination of hardware and software refill gives excellent performance while minimizing processor complexity. If the second translation succeeds, the PTE load is done through the DataCache, if one is configured, and the attributes for the page containing the PTE enable such a cache access. The PTE's `Ring` field is then used as an index into the `RASID` register, and the resulting ASID is written together with the rest of the PTE into the TLB.

Xtensa's TLB refill mechanism requires the page table for the current address space to reside in the current virtual address space. The `PTEBase` field of the `PTEVADDR` register gives the base address of the page table. On a TLB miss, the processor forms the virtual address of the PTE by concatenating the `PTEBase` portion of `PTEVADDR`, the Virtual Page Number (VPN) bits of the miss virtual address, and 2 zero bits. The bits used from `PTEVADDR` and from the virtual address are configuration dependent; the exact calculation for 4-byte PTEs is

$$\text{PTEVADDR}_{31..22} \parallel \text{vAddr}_{31..12} \parallel 2'b00$$

The format of the PTEs when `EXTMEMATTRIBUTES=False` is shown in [MMU Option Page Table Entry \(PTE\) Format when `EXTMEMATTRIBUTES=False`](#). The most significant bits hold the Physical Page Number (PPN), the translation of the virtual address corresponding to this entry. The `Sw` bits are available for software use in the page table (they are not stored in the TLB). The Ring field specifies the privilege level required to access this page; this is used to choose one of the four ASIDs from `RASID` when the TLB is written. The attribute field gives the access attributes for this page (see [MMU Option Memory Attributes when `EXTMEMATTRIBUTES=False`](#) on page 233).

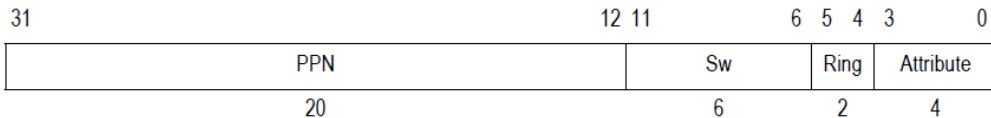


Figure 42: MMU Option Page Table Entry (PTE) Format when `EXTMEMATTRIBUTES=False`

The format of the PTEs when `EXTMEMATTRIBUTES=True` is shown in [MMU Option Page Table Entry \(PTE\) Format when `EXTMEMATTRIBUTES=True`](#). The most significant bits hold the Physical Page Number (PPN), the translation of the virtual address corresponding to this entry. The `sw` bits are available for software use in the page table (they are not stored in the TLB). The Ring field specifies the privilege level required to access this page; this is used to choose one of the four ASIDs from `RASID` when the TLB is written. The MT field gives the access attributes for this page (see [MMU Option Memory Type when `EXTMEMATTRIBUTES=True`](#) on page 236). The `W` bit indicates whether writes are allowed to the page and the `X` bit indicates whether execution is allowed from the page.

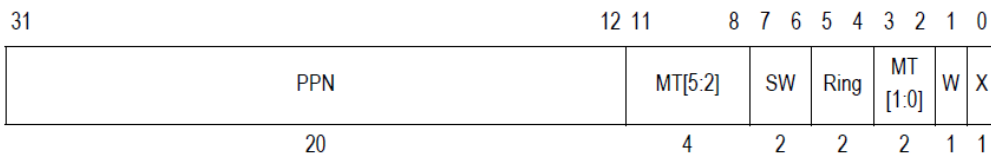


Figure 43: MMU Option Page Table Entry (PTE) Format when `EXTMEMATTRIBUTES=True`

The configuration described in [The MMU Option Memory Map](#) on page 226 (with `IVARWAY56/DVARWAY56` Fixed) provides a maximum of 3328 MB of dynamically mapped space (4 GB of total virtual address space with 768 MB of statically mapped space). The page table for this maximum size requires 851968 PTEs (3328MB/4 kB). The entire set of PTEs require 3328 kB of virtual address space (at 4 bytes per PTE). The PTEs themselves are at virtual addresses and, therefore, 832 of the PTEs in the table are for mapping the page table itself. These PTEs for mapping the page table will fit onto a single page, the mapping for which may be written into one of the single-entry ways (Ways 7-9) of the data TLB for guaranteed access.

For example, if `PTEVADDR` is set to `32'hCFC00000`, then the virtual address space between there and `32'hCFF3FFFF` is used as the page table. That page table is mapped by the 832 entries between `32'hCFF3F000` and `32'hCFF3FCFF`. The translation for the page at

32'hCFF3F000 is placed in one of the single-entry ways of the data TLB. (The accesses that might have used the remaining 192 PTE entries on that page would already have been translated by one of the constant ways.) Many of those 832 entries may be marked invalid and the physical address space required for the page table may be made very small.

In systems with large memories, the above maximum configuration may be improved in performance by mapping the entire page table into the constant way (Way 5). If PTEVADDR is set to 32'hD4000000, for example, the virtual address space between there and 32'hD433FFFF, which maps to the physical address space between 32'h04000000 and 32'h0433FFFF (between 64 MB and about 68 MB) is used for a flat page table mapping all of memory. Any TLB miss will now be handled by the hardware refill as the translation for the PTE will be handled by the constant way. The disadvantage is that over 3 MB of memory must be allocated to the page table.

In a small system, where all processes are limited to the first 8 MB of virtual space, PTEVADDR might be set to 32'hCFC00000 and two of the single entry ways set to map the page at 32'hCFC00000 and the page at 32'hCFC01000. One or both pages of PTEs could be used for translations and the hardware refill would always succeed for legal addresses.

5.6.10 MMU Option Memory Attributes when `EXTMEMATTRIBUTES=False`

Currently available hardware supports the memory attributes described in this section when `EXTMEMATTRIBUTES=False`. T1050 hardware supported somewhat different memory attributes, which are described in [MMU Option Memory Attributes](#). System software may use the subset of attributes (1, 3, 5, 7, 12, 13, and 14) which have not changed to support all Xtensa processors.

The memory attributes discussed in this section apply both to attribute values written in and read from the TLBs (see [Formats for Writing MMU Option TLB Entries](#) on page 227 and [Formats for Reading MMU Option TLB Entries](#) on page 228) and to attribute values stored in the PTE entries and written into the AutoRefill ways of the TLBs (see [MMU Option Auto-Refill TLB Ways and PTE Format](#) on page 231).

For a more detailed description of the memory access process and the place of these attributes in it, see [The Memory Access Process](#) on page 190.

[MMU Option Attribute Field Values](#) shows the meanings of the attributes for instruction fetch, data load, and data store. For a more detailed description of the memory access process and the place of these attributes in it, see [The Memory Access Process](#) on page 190.

The first column in [MMU Option Attribute Field Values](#) indicates the attribute from the TLB while the remaining columns indicate various effects on the access. The columns are described in the following bullets:

- **Attr** — the value of the 4-bit Attribute field of the TLB entry.
- **Rights** — whether the TLB entry may successfully translate a data load, a data store, or an instruction fetch.
 - The first character is an `x` if the entry is valid for a data load and a dash ("-") if not.

- The second character is a *w* if the entry is valid for a data store and a dash ("-") if not.
- The third character is an *x* if the entry is valid for an instruction fetch and a dash ("-") if not.

If the translation is not successful, an exception is raised.

Local memory accesses (including XLMI) consult only the Rights column.

- **Meaning for Cache Access** — the verbal description of the type of access made to the cache.
- **Access Cache** — indicates whether the cache provides the data.
 - The first character is an *h* if the cache provides the data when the tag indicates hit and a dash ("-") if it does not.
 - The second character is an *m* if the cache provides the data when the tag indicates a miss and a dash ("-") if it does not. This capability is used only for Isolate mode.
- **Fill Cache** — indicates whether an allocate and fill is done to the cache if the tag indicates a miss.
 - The first character is an *r* if the cache is filled on a data load and a dash ("-") if it is not.
 - The second character is a *w* if the cache is filled on a data store and a dash ("-") if it is not.
 - The third character is an *x* if the cache is filled on an instruction fetch and a dash ("-") if it is not.
- **Guard Load** — refers to the guarded attribute as described in [Access Characteristics Encoded in the Attributes](#). Stores are always guarded and instruction fetches are never guarded, but loads are guarded where there is a “yes” in this column. Local memory loads are not guarded.
- **Write Thru** — indicates whether a write is done through the PIF interface.
 - The first character is an *h* if a Write Thru occurs when the tag indicates hit and a dash ("-") if it does not.
 - The second character is an *m* if a Write Thru occurs when the tag indicates a miss and a dash ("-") if it does not.

Writes to local memories are never Write-Thru. In most implementations, a write-thru will only occur after any needed cache fill is complete.

Table 110: MMU Option Attribute Field Values

Attr	Rights	Meaning for Cache Access	Access Cache	Fill Cache	Guard Load	Write Thru
0	r--	Bypass cache	--	---	yes	--
1	r-x	Bypass cache	--	---	yes	--

Attr	Rights	Meaning for Cache Access	Access Cache	Fill Cache	Guard Load	Write Thru
2	rw-	Bypass cache	--	---	yes	hm
3	rwX	Bypass cache	--	---	yes	hm
4	r--	Cached, WrtBack alloc	h-	r--	-	--
5	r-x	Cached, WrtBack alloc	h-	r-x	-	--
6	rw-	Cached, WrtBack alloc	h-	rw- ¹	-	-- ¹
7	rwX	Cached, WrtBack alloc	h-	rwX ¹	-	-- ¹
8	r--	Cached, WrtThru	h-	r--	-	--
9	r-x	Cached, WrtThru	h-	r-x	-	--
10	rw-	Cached, WrtThru	h-	r--	-	hm
11	rwX	Cached, WrtThru	h-	r-x	-	hm
12	---	illegal ²	--	---	-	--
13	rw-	Cache Isolated ³	hm	---	-	--
14	---	illegal ²	--	---	-	--
15	---	Reserved ²	—	—	—	—

1. If the Data Cache is not configured as writeback, entries for Attributes 6 & 7 are replaced by the corresponding ones for Attributes 10 & 11
2. Raises exception. EXCCAUSE is set to InstFetchProhibitedCause, LoadProhibitedCause, or StoreProhibitedCause depending on access type
3. For test only, implementation dependent, uses data cache like local memories and ignores tag.

5.6.11 MMU Option Memory Type when *EXTMEMATTRIBUTES*=True

Hardware supports the Memory Types described in this section when *EXTMEMATTRIBUTES*=True. The Memory Type used here is similar in concept to the Memory Type described for the Memory Protection Unit in *Memory Protection Unit Option Memory Type Field* on page 215. Not all of those Memory Type values are supported and the encoding is narrowed to fit into 6 bits.

The meaning of the values in the Memory Type field is shown below in *MMU Option Memory Type*. The first column is the number in the field labeled Memory Type and the other columns show some of the principal characteristics of the memory type.

Shareable regions are regions where hardware ensures that multiple masters can correctly share the memory in that region. Non-shareable regions may only be accessed by a single master. Multiple threads on the same master may share a non-shareable region.

When an interrupt arrives during a non-interruptible device load, the processor will wait to receive the load value from the system, complete the load instruction, and then process the interrupt. In addition, non-interruptible device loads will not be speculated. These characteristics are required to properly read devices that have read side effects.

When an interrupt arrives during an interruptible device load, the load will complete on the bus, the value will be thrown away, and after the interrupt, the load will repeat. This capability is required for best interrupt latency. These loads may also be speculated.

Loads from memory (as opposed to device space) and all instruction fetches will freely be interrupted and retried and may be speculated as memory is assumed to have no side effects. All stores happen exactly once and are not speculated. Interrupt latency is not affected.

Put differently, stores and non-interruptible loads are "guarded" while instruction fetches and interruptible loads are not guarded.

Additional characteristics of the memory type are indicated by letters used in the field value column indicating either a 0 or a 1. The meaning of these letters is described below.

Table 111: MMU Option Memory Type

Field Value ¹	System Type	Shareable	Interruptible Load	Cached in Data or Instruction Cache
0_00_00B ²	Device	Yes	No	No
0_00_01B ²			Yes	
0_00_10B	Non-Cacheable Memory	No		
0_00_11B		Yes		

Field Value ¹	System Type	Shareable	Interruptible Load	Cached in Data or Instruction Cache
0_01_RWC	Cacheable Memory	No		
0_1I_RWC		Yes		
1_wc_RWC		No		Yes
<ol style="list-style-type: none"> Letters in values are explained in text. Exclusive loads and stores are sent with a zero AxLOCK field. 				

The following letters are used in *MMU Option Memory Type* above:

- **b**: If the system bus supports the concepts, the bus transaction resulting from this access will indicate non-Bufferable if the bit is clear and Bufferable if the bit is set.
- **c**: If the system bus supports the concepts, any bus transaction resulting from this access will indicate Write-through cacheable if the bit is clear and Writeback cacheable if the bit is set.
- **w**: If the system bus supports the concepts, any bus transaction resulting from this access will indicate no-Write-allocate if the bit is clear and Write-allocate if the bit is set.
- **r**: If the system bus supports the concepts, any bus transaction resulting from this access will indicate no-Read-allocate if the bit is clear and Read-allocate if the bit is set.
- **i**: If the system bus supports the concepts, any bus transaction resulting from this access will indicate Outer-shareable if the bit is clear and Inner-shareable if the bit is set.
- **c**: The processor will cache this in its internal data cache in Write-through mode if the bit is clear and in Write-back mode if the bit is set. If the processor is not capable of the indicated mode, it will not cache the line. If the System Type is Cacheable Memory and Shareable, then the processor cache only caches data if full hardware coherence is configured and enabled.
- **w**: The processor will use this bit as a hint that its internal data cache should not allocate on write if the bit is clear and should allocate on write if the bit is set. The processor is not required to follow this hint.

Regions marked as Shareable in *MMU Option Memory Type* are correctly shared with other masters in all systems. Those which do not have cacheable memory types perform each access externally and mark the transaction as non-cacheable so that sharing can occur in main memory or in a cache common to all masters. Those which do have cacheable memory types use cache coherence if it is configured and enabled.

If coherence is not configured or not enabled, the processor internal caches refuse to cache accesses to Shareable regions. The system can then make them shared. Such accesses will be marked cacheable and shareable on the system bus such that a coherent cache outside the processor may hold them for improved performance. An external cache for which

coherence is not configured or not enabled may also refuse to cache regions labeled shareable so that they may be shared in main memory or in a cache common to all masters.

5.6.12 MMU Option Operation Semantics

The following functions are used in the operation sections of the individual instruction definitions:

```

function ltranslate(vAddr, ring)
    ltranslate ← (pAddr, attributes, cause)
endfunction ltranslate

function ASID(ring)
    ASID ← RASIDring*8+ASIDBits-1..ring*8
endfunction ASID

function InstPageBits(wi)
    sizecodebits ← ceil(log2(InstTLB[wi].PageSizeCount))
    sizecode ← IPAGESIZEwi*4+sizecodebits-1..wi*4
    InstPageBits ← InstTLB[wi].PageBits[sizecode]
endfunction InstPageBits

function SplitInstTLBEntrySpec(spec)
    wih ← ceil(log2(InstTLBWayCount)) ← 1
    wi ← specwih..0
    eil ← InstPageBits(wi)
    eih ← eil ← log2(InstTLB[wi].IndexCount)
    ei ← speceih..eil
    vpn ← specInstTLBAddrBits-1..eih+1
    SplitInstTLBEntrySpec ← (vpn, ei, wi)
endfunction SplitInstTLBEntrySpec

function ProbeInstTLB (vAddr)
    match ← 0
    vpn ← undefined
    ei ← undefined
    wi ← undefined
    for i in 0..InstTLBWayCount-1 do
        if then
            match ← match + 1
            vpn ← x
            ei ← x
            wi ← i
        endif
    endfor
    ProbeInstTLB ← (match, vpn, ei, wi)
endfunction ProbeInstTLB

```

6. Options for Other Purposes

Topics:

- *Windowed Register Option*
- *Miscellaneous Special Registers Option*
- *Thread Pointer Option*
- *Processor ID Option*
- *Debug Option*

This section contains options that do not fit easily into the previous sections. The Windowed Register Option provides the hardware for a memory efficient ABI. The Miscellaneous Special Registers Option provides additional scratch registers. The Processor ID Option provides the ability for software to determine on which processor it is running. The Debug Option provides hardware to assist in debugging processors.

6.1 Windowed Register Option

The Windowed Register Option replaces the simple 16-entry AR register file with a larger register file from which a window of 16 entries is visible at any given time. The window is rotated on subroutine entry and exit, automatically saving and restoring some registers. When the window is rotated far enough to require registers to be saved to or restored from the program stack, an exception is raised to move some of the register values between the register file and the program stack. The option reduces code size and increases performance of programs by eliminating register saves and restores at procedure entry and exit, and by reducing argument-shuffling at calls. It allows more local variables to live permanently in registers, reducing the need for stack-frame maintenance in non-leaf routines.

Xtensa ISA register windows are different from register windows in other instruction sets. Xtensa register increments are 4, 8, and 12 on a per-call basis, not a fixed increment as in other instruction sets. Also, Xtensa processors have no global address registers. The caller specifies the increment amount, while the callee performs the actual increment by the `ENTRY` instruction. The compiler uses an increment sufficient to hide the registers that are live at the point of the call (which the compiler can pack into the fewest possible at the low end of the register-number space). The number of physical registers is 32 or 64, which makes this a more economical configuration. Sixteen registers are visible at one time. Assuming that the average number of live registers at the point of call is 6.5 (return address, stack pointer, and 4.5 local variables), and that the last routine uses 12 registers at its peak, this allows nine call levels to live in 64 registers ($8 \times 6.5 + 12 = 64$). As an example, an average of 6.5 live registers might represent 50% of the calls using an increment of 4, 38% using an increment of 8, and 12% using an increment of 12.

- Prerequisites: [Exception Option 2](#) on page 126.
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

The rotation of the 16-entry visible window within the larger register file is controlled by the WindowBase Special Register added by the option. The rotation always occurs in units of four registers, causing the number of bits in WindowBase to be $\log_2(\text{NAREG}/4)$. Rotation at the time of a call can instantly save some registers and provide new registers for the called routine. Each saved register has a reserved location on the stack, to which it may be saved if the call stack extends enough farther to need to re-use the physical registers. The WindowStart Special Register, which is also added by the option and consists of $\text{NAREG}/4$ bits, indicates which four register units are currently cached in the physical register file instead of residing in their stack locations. An attempt to use registers live with values from a parent routine raises an Overflow Exception which saves those values and frees the registers for use. A return to a calling routine whose registers have been previously saved to the stack raises an Underflow Exception which restores those values. Programs without wide swings in the depth of the call stack save and restore values only occasionally.

6.1.1 Windowed Register Option Architectural Additions

Windowed Register Option Constant Additions (Exception Causes) through *Windowed Register Option Instruction Additions* show this option's architectural additions.

Table 112: Windowed Register Option Constant Additions (Exception Causes)

Exception Cause	Description	Constant Value
AllocaCause	MOVSP instruction, if the caller's registers are not present in the register file (see <i>Exception Causes</i> on page 138)	6'b000101 (decimal 5)

Table 113: Windowed Register Option Processor-Configuration Additions

Parameter	Description	Valid Values
WindowOverflow4	Window overflow exception vector for 4-register stack frame	32-bit address ¹
WindowUnderflow4	Window underflow exception vector for 4-register stack frame	32-bit address ¹
WindowOverflow8	Window overflow exception vector for 8-register stack frame	32-bit address ¹
WindowUnderflow8	Window underflow exception vector for 8-register stack frame	32-bit address ¹
WindowOverflow12	Window overflow exception vector for 12- register stack frame	32-bit address ¹
WindowUnderflow12	Window underflow exception vector for 12- register stack frame	32-bit address ¹
NAREG	Number of address registers	32 or 64
<p>1. Some implementations have restrictions on the alignment and relative location of the WindowOverflowN and WindowUnderflowN vectors. See “<code>procedure WindowCheck (wr, ws, wt)</code>” in <i>Window Overflow Check</i> on page 245 for how these are used.</p>		

Table 114: Windowed Register Option Processor-State Additions and Changes

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
AR	NAREG	32	Address registers (general registers)	R/W	—
WindowBase	1	$\log_2(\text{NAREG}/4)$	Base of current address-register window	R/W	72
WindowStart	1	NAREG/4	Call-window start bits	R/W	73
PS.CALLINC	1	2	Miscellaneous processor state, window increment from call (see PS Register Fields on page 135)	R/W	230
PS.OWB	1	4	Miscellaneous processor state, old window base (see PS Register Fields on page 135)	R/W	230
PS.WOE	1	1	Miscellaneous processor state, window overflow enable (see PS Register Fields on page 135)	R/W	230

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` ([Processor Control Instructions](#)). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.

Table 115: Windowed Register Option Instruction Additions

Instruction ¹	Format	Definition
MOVSP	RRR on page 656	Atomic check window and move

Instruction ¹	Format	Definition
CALL4, CALL8, CALL12	CALLX on page 657	Call subroutine, PC-relative. These instructions communicate the number of registers to hide using PS .CALLINC in addition to the operation of CALL0.
CALLX4, CALLX8, CALLX12	CALLX on page 657	Call subroutine, address in register. These instructions communicate the number of registers to hide using PS .CALLINC in addition to the operation of CALLX0.
ENTRY	BRI12 on page 658	Subroutine entry—rotate registers, adjust stack pointer. This instruction should not be used in a routine called by CALL0 or CALLX0.
RETW	CALLX on page 657	Subroutine return—unrotate registers, jump to return address. Used to return from a routine called by CALL4, CALL8, CALL12, CALLX4, CALLX8, or CALLX12.
RETW.N ²	RRRN on page 658	Same as RETW in a 16-bit encoding
ROTW	RRR on page 656	Rotate window by a constant. ROTW is intended for use in exception handlers and context switch.
L32E	RRI4 on page 656	Load 32 bits for window exception
S32E	RRI4 on page 656	Store 32 bits for window exception
REWO	RRR on page 656	Return from window overflow exception
REWU	RRR on page 656	Return from window underflow exception
<p>1. These instructions are fully described in Instruction Descriptions on page 321.</p> <p>2. Exists only if the Code Density Option described in Code Density Option on page 82 is configured.</p>		

6.1.2 Managing Physical Registers

The `WindowBase` Special Register gives the position of the current window into the physical register file. In the instruction descriptions, `AR[i]` is a short-hand for a reference to the physical register file `AddressRegister` defined as follows:

```
AddressRegister[((2'b00i3..2) + WindowBase) | i1..0]
```

The `WindowStart` Special Register gives the state of physical registers (unused or part of a window). `WindowStart` is used both to detect overflow and underflow on register use and procedure return, as well as to determine the number of registers to be saved in a given stack frame when handling exceptions and switching contexts. There is one bit in `WindowStart` for each four physical registers. This bit is set if those four registers are `AR[0]` to `AR[3]` for some call. `WindowStart` bits are set by `ENTRY` and cleared by `RETW`.

The `WindowBase` and `WindowStart` registers are undefined after processor reset, and should be initialized by the reset exception vector code.

Conceptual Register Window Read through *Fastest Register Window Read* show three functionally identical implementations of windowed registers. *Conceptual Register Window Read* shows the concept of how the registers are addressed. *Faster Register Window Read* shows logic with the same functional result but with little or no penalty paid in timing for the addition of the `WindowBase` value. *Fastest Register Window Read* shows a third version of the logic with the same functional result but with no timing loss at all caused by the addition of the `WindowBase` value.

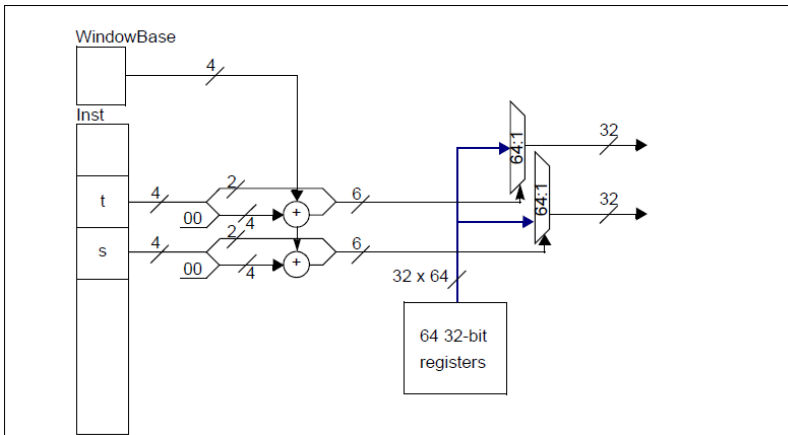


Figure 44: Conceptual Register Window Read

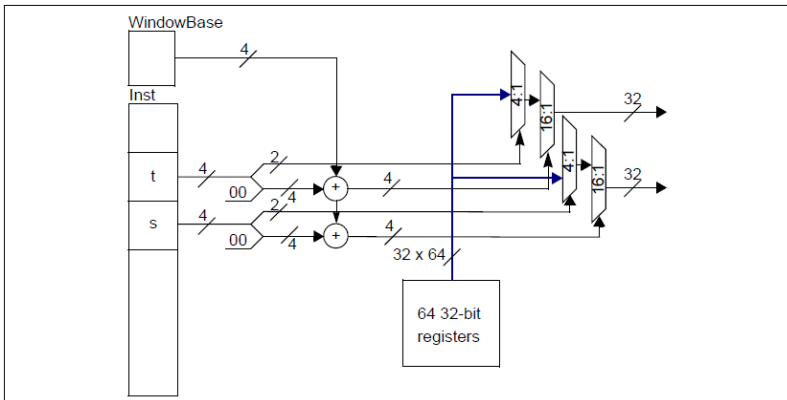


Figure 45: Faster Register Window Read

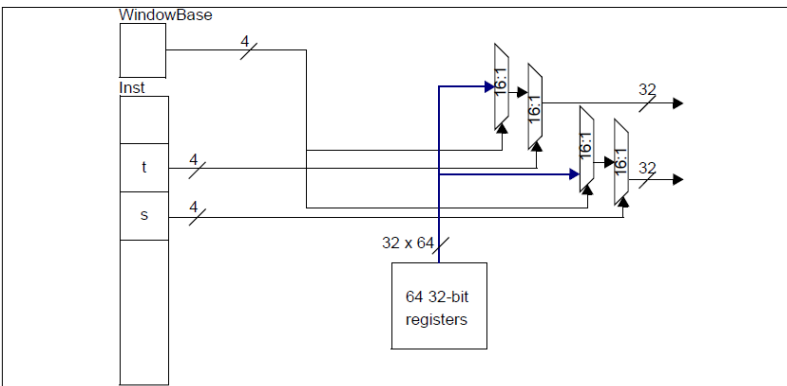


Figure 46: Fastest Register Window Read

6.1.3 Window Overflow Check

The `ENTRY` instruction moves the register window, but does not guarantee that all the registers in the current window are available for use. Instead, the processor waits for the first reference to an occupied physical register before triggering a window overflow. This prevents unnecessary overflows, because many routines do not use all 16 of their virtual registers.

Register Window Near Overflow shows the state of the register file just prior to a reference that causes an overflow. The `WS(n)` notation shows which `WindowStart` bits are set in this example, and gives the distance to the next bit set (that is, the number of registers stored for the corresponding stack frame). In the figure, “`rmax`” indicates the maximum register that the current procedure uses and “`Base`” is an abbreviation for `WindowBase`. Note that registers are considered in groups of four here.

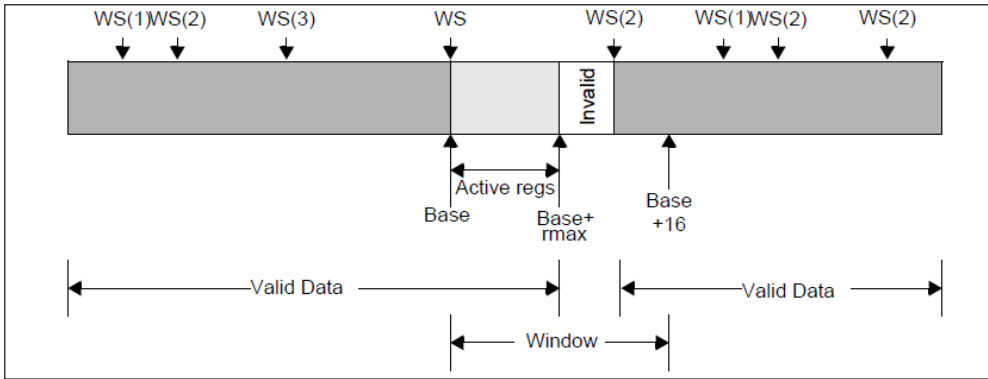


Figure 47: Register Window Near Overflow

The check for overflow is done as follows:

```
WindowCheck ( if ref(AR[r]) then r3..2 else 2'b00,
              if ref(AR[s]) then s3..2 else 2'b00,
              if ref(AR[t]) then t3..2 else 2'b00)
```

where `ref()` is 1 if the register is used by the instruction, and 0 otherwise, and `WindowCheck` is defined as follows:

```
procedure WindowCheck (wr, ws, wt)
  n ← if (wr ≠ 2'b00 or ws ≠ 2'b00 or wt ≠ 2'b00)
      and WindowStartWindowBase+1 then 2'b01
  else if (wr1 or ws1 or wt1)
      and WindowStartWindowBase+2 then 2'b10
  else if (wr = 2'b11 or ws = 2'b11 or wt = 2'b11)
      and WindowStartWindowBase+3 then 2'b11
  else 2'b00
  if CWOE = 1 and n ≠ 2'b00 then
    PS.OWB ← WindowBase
    m ← WindowBase + (2'b001n)
    PS.EXCM ← 1
    EPC[1] ← PC
    nextPC ← if WindowStartm+1 then WindowOverflow4
             else if WindowStartm+2 then WindowOverflow8
             else WindowOverflow12
    WindowBase ← m
  endif
endprocedure WindowCheck
```

A single instruction may raise multiple window overflow exceptions. For example, suppose that registers 4..7 of the current window still contain a previous call frame's values (`WindowStartWindowBase+1` is set), and 8..15 are part of the subroutine called by that frame (`WindowStartWindowBase+2` is also set), and an instruction references register 10. The processor will raise an exception to spill registers 4..7 and then return to retry the

instruction, which will then raise another exception to spill registers 8..15. On return from this overflow handler, the reference will finally succeed.

6.1.4 Call, Entry, and Return Mechanism

The register window mechanics of the {CALL, CALLX}{4,8,12}, ENTRY, and {RETW, RETW.N} instructions are:

```
CALLn/CALLXn
    WindowCheck (2'b00, 2'b00, n)
    PS.CALLINC ← n
    tmp ← nextPC
    nextPC ← computation according to CALL type
    AR[n!2'b00] ← n | (tmp)29..0

ENTRY s, imm12
    AR[PS.CALLINC!s1..0] ← AR[s] - (017!imm12!03)
    WindowBase ← WindowBase + (02!PS.CALLINC)
    WindowStartWindowBase ← 1
```

In the definition of ENTRY above, the AR read and the AR write refer to different registers.

```
RETW/RETN
    n ← AR[0]31..30
    nextPC ← PC31..30 | AR[0]29..0
    owb ← WindowBase
    m ← if WindowStartWindowBase-4'b0001 then 2'b01
        elsif WindowStartWindowBase-4'b0010 then 2'b10
        elsif WindowStartWindowBase-4'b0011 then 2'b11
        else 2'b00
    if n = 2'b00 | (m ≠ 2'b00 & m ≠ n) | PS.WOE=0 | PS.EXCM=1 then
        -- undefined operation
        -- may raise illegal instruction exception
    else
        WindowBase ← WindowBase - (02!n)
        if WindowStartWindowBase ≠ 0 then
            WindowStartowb ← 0
        else
            -- Underflow exception
            PS.EXCM ← 1
            EPC[1] ← PC
            PS.OWB ← owb
            nextPC ← if n ← 2'b01 then WindowUnderflow4
                else if n ← 2'b10 then WindowUnderflow8
                else WindowUnderflow12
        endif
    endif
endif
```

The RETW opcode assignment is such that the *s* and *t* fields are both zero, so that the hardware may use either AR[*s*] or AR[*t*] in place of AR[0] above. Underflow is detected by the caller's window's WindowStart bit being clear (that is, not valid). [Register Window Just Before Underflow](#) shows the register file just before a RETW that raises an underflow exception. window overflow and window underflow exceptions leave PS.UM unchanged.

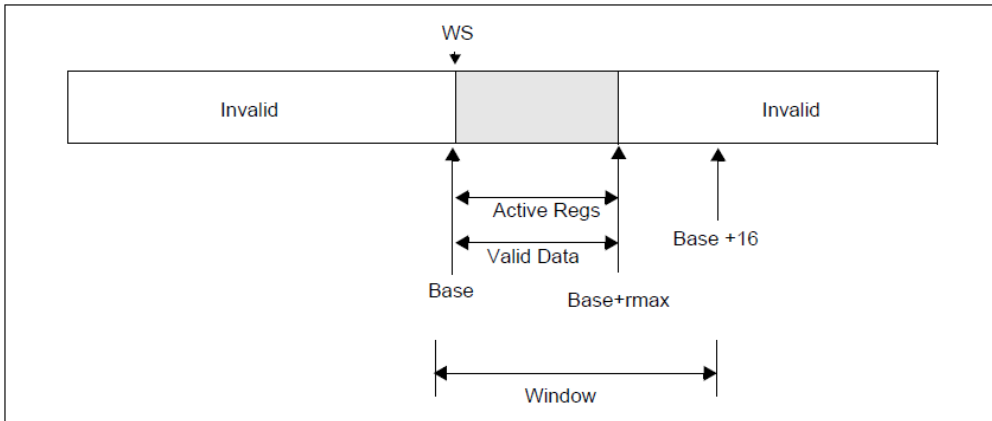


Figure 48: Register Window Just Before Underflow

6.1.5 Windowed Procedure-Call Protocol

While the procedure-call protocol is a matter for the compiler and ABI, the Xtensa ISA, and particularly the Windowed Register Option was designed with the following goals in mind:

- Provide highly efficient call/return (measured in both code size and execution time)
- Support per-call register window increments
- Use a single stack for both register save/restore and local variables
- Support variable frame sizes (for example, `alloca`)
- Support programming language exception handling (for example, `setjmp/longjmp`, `catch/throw`, and so forth)
- Support debuggers
- Require minimal special ISA features (special registers and so forth)

[Windowed Register Usage](#) shows the register usage in the Windowed Register Option. Refer to [The Windowed Register and CALL0 ABIs](#) on page 684 for a more complete description of the Windowed Register ABI.

Table 116: Windowed Register Usage

Callee Register	Register Name	Usage
0	a0	Return address
1	a1/sp	Stack pointer
2..7	a2..a7	In, out, inout, and return values

Calls to routines that use only `a2..a3` as parameters may use the `CALL4`, `CALL8`, or `CALL12` instructions to save 4, 8, or 12 live registers. Calls to routines that use `a2..a7` for parameters may use only `CALL8` or `CALL12`. The following assembly language illustrates the call protocol.

```
// In procedure g, the call
//   z = f(x, y)
// would compile into
    mov     a6, x    // a6 is f's a2 (x)
    mov     a7, y    // a7 is f's a3 (y)
    call4   f        // put return address in f's a0,
                    // goto f
    mov     z, a6    // a6 is f's a2 (return value)
// The function
//   int f(int a, int *b) { return a + *b; }
// would compile into
f:   entry   sp, framesize // allocate stack frame, rotate regs
      // on entry, a0/ return address, a1/ stack pointer,
      // a2/ a, a3/ *b
      l32i   a3, a3, 0 // *b
      add    a2, a2, a3 // *b + a
      retw
```

The “highly efficient call/return” goal requires that there not be separate stack and frame pointer registers in cases where they would differ by a constant (that is, no `alloca` is used). There are simply not enough registers to waste. For routines that do call `alloca`, the compiler will copy the initial stack pointer to another register and use that for addressing all locals.

The variable allocation,

```
p1 = alloca(n1);
```

will be implemented as

```
movi    t4, -16           // for alignment to 16-byte boundary
sub     t5, sp, n1        // reserve stack space
and     t4, t5, t4        // ...
movsp   sp, t4           // atomically set sp
addi    p1, sp, -16+botsize // save pointer
```

The `botsize` in the last statement allows the compiler to maintain a block of words at the bottom of the stack (for example, this block might be for memory arguments to routines). The `-16` is a constant of the call protocol; it puts 16 bytes of the bottom area below the stack pointer (since they are infrequently referenced), leaving the limited range of the ISA’s load/store offsets available for more frequently referenced locals.

[Stack Frame Before `alloca\(\)`](#) and [Stack Frame After First `alloca\(\)`](#) show the stack frame before and after `alloca`.

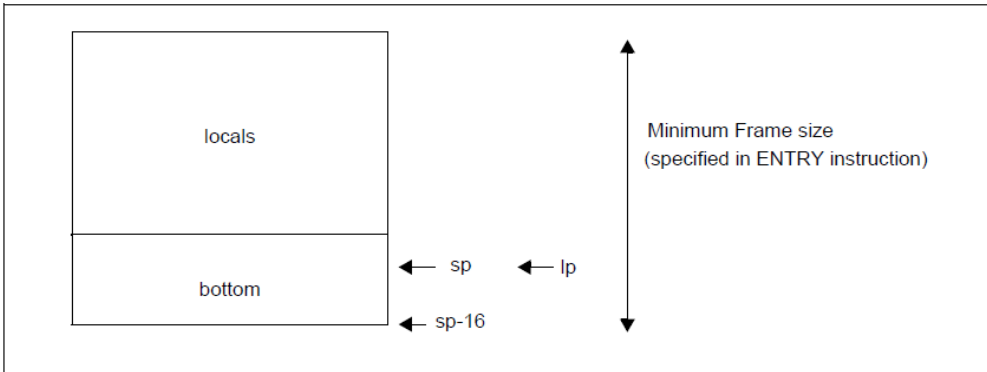


Figure 49: Stack Frame Before `alloca()`

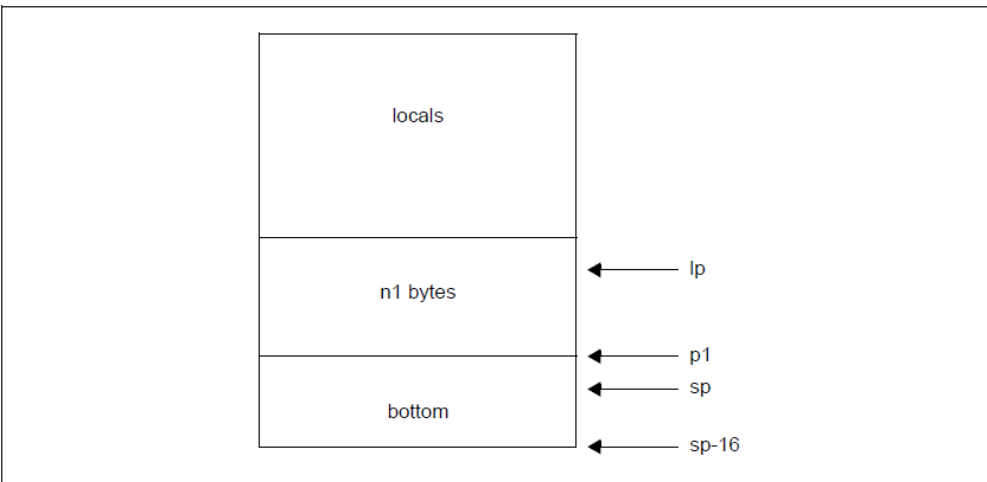


Figure 50: Stack Frame After First `alloca()`

Stack Frame Layout shows the stacking of frames when the stack grows downward, as on most other systems. The window save area for a frame is addressed with negative offsets from the next stack frame's `sp`. Four registers are saved in the base save area. If more than four registers are saved, they are stored at the top of the stack frame, in the extra save area, which can be found with negative offsets from the previous stack frame's `sp`. This unusual split allows for simple backtrace while providing for a variable sized save area.

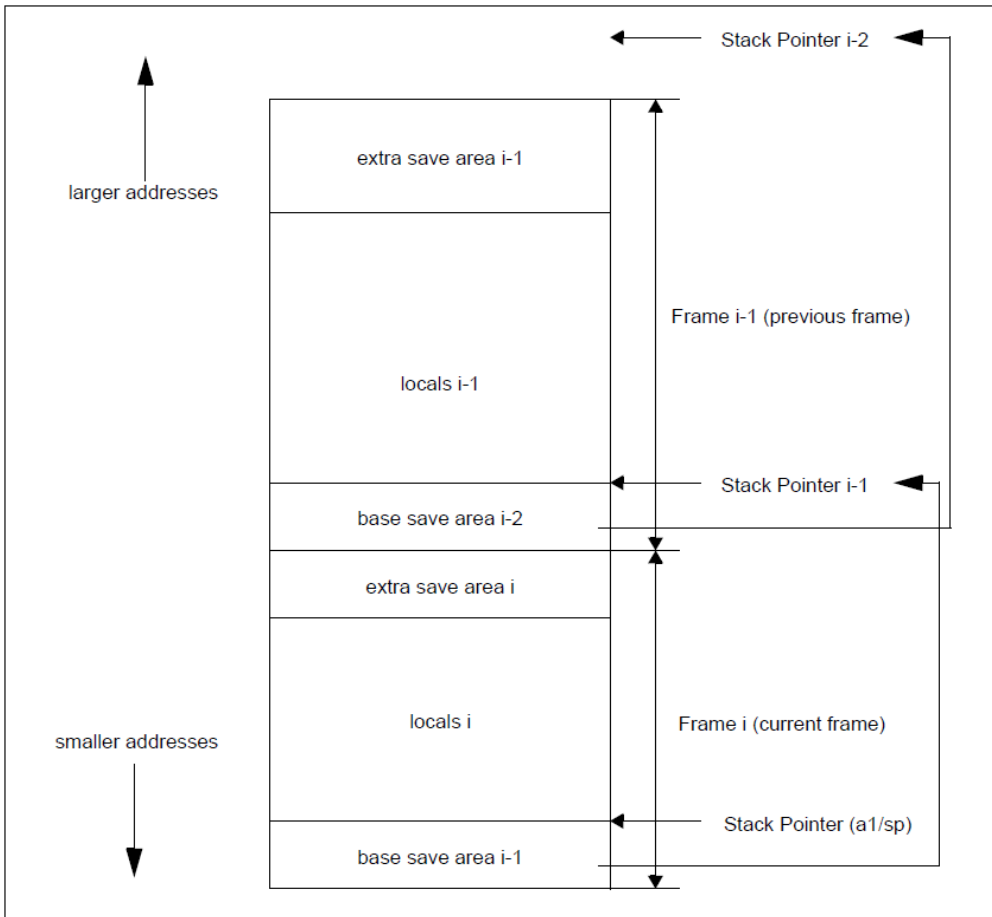


Figure 51: Stack Frame Layout

Several of the goals listed require that call stacks be backward-traceable. That is, from the state of `call[i]`, it must be possible to determine the state of `call[i-1]`. It is best if the state of `call[i]` can be summarized in a single pointer (at least when the registers have been saved), in which case this requirement is best described as: There must be a means of determining the pointer for `call[i-1]` from the pointer of `call[i]`. For managing register-window overflow or underflow, this method should also be very efficient; it should not, for example, involve routine-specific information or other table lookup (for example, frame size or stack offsets).

The Xtensa ISA represents the state of `call[i]` with its stack pointer (not the frame pointer, as that is routine-specific and would cost too much to lookup). This can be made to work even with `alloca`. Therefore it must be possible to read the stack pointer for `call[i-1]` at a fixed offset from the stack pointer (not the frame pointer) for `call[i]`. Thus, the stack pointer

for `call[i-1]` is stored in the area labeled “base save area i-1” in *Stack Frame Before `alloca()`*.

For efficiency, the `call[i-1]` stack pointer is only stored into `call[i]`'s frame when `call[i-1]`'s registers are stored into the stack on overflow. This is sufficient for register window underflow handling. Other back-tracing operations should begin by storing registers of all call frames back into the stack.

Because the `call[i-1]` stack pointer is referenced infrequently, it is stored at a negative offset from the stack pointer. This leaves the ISA's limited positive offsets available for more frequent uses. Thus, the stack always reaches to 16 bytes below the contents of the stack pointer. Interrupts and such must respect this 16-byte reserved space below the stack pointer. Because the minimum number of registers to save is four, the processor stores four of `call[i-1]`'s registers, `a0..a3`, in this space; the rest (if any) are saved in `call[i-1]`'s own frame.

The register-window call instructions only store the least-significant 30 bits of the return address. Register-window return instructions leave the two most-significant bits of the PC unchanged. Therefore, subroutines called using register window instructions must be placed in the same 1 GB address region as the call.

6.1.6 Window Overflow and Underflow to and from the Program Stack

Register-window underflow occurs when a return instruction decrements to a window that has been spilled (indicated by its `WindowStart` bit being cleared). The processor saves the current PC in `EPC[1]` and transfers to one of three underflow handlers based on the register window decrement. When the MMU Option is configured, it is necessary for the handlers to access the stack with the same privilege level as the code that raised the exception. Two special instructions, `L32E` and `S32E`, are therefore added by the Windowed Register Option for this purpose. In addition, these instructions use negative offsets in the formation of the virtual address, which saves several instructions in the handlers. The exception handlers could be as simple as the following:

```
WindowOverflow4: // inside call[i] referencing a register that
                // contains data from call[j]
                // On entry here: window rotated to call[j] start point; the
                // registers to be saved are a0-a3; a4-a15 must be preserved
                // a5 is call[j+1]'s stack pointer
s32e  a0, a5, -16 // save a0 to call[j+1]'s frame
s32e  a1, a5, -12 // save a1 to call[j+1]'s frame
s32e  a2, a5, -8  // save a2 to call[j+1]'s frame
s32e  a3, a5, -4  // save a3 to call[j+1]'s frame
rfwo // rotates back to call[i] position

WindowUnderflow4: // returning from call[i+1] to call[i] where
                 // call[i]'s registers must be reloaded
                // On entry here: a0-a3 are to be reloaded with
                // call[i].reg[0..3] but initially contain garbage.
                // a4-a15 are call[i+1].reg[0..11],
                // (in particular, a5 is call[i+1]'s stack pointer)
```

```

// and must be preserved
l32e a0, a5, -16 // restore a0 from call[i+1]'s frame
l32e a1, a5, -12 // restore a1 from call[i+1]'s frame
l32e a2, a5, -8 // restore a2 from call[i+1]'s frame
l32e a3, a5, -4 // restore a3 from call[i+1]'s frame
rfwu

```

WindowOverflow8:

```

// On entry here: window rotated to call[j]; the registers to be
// saved are a0-a7; a8-a15 must be preserved
// a9 is call[j+1]'s stack pointer
s32e a0, a9, -16 // save a0 to call[j+1]'s frame
l32e a0, a1, -12 // a0 <- call[j-1]'s sp
s32e a1, a9, -12 // save a1 to call[j+1]'s frame
s32e a2, a9, -8 // save a2 to call[j+1]'s frame
s32e a3, a9, -4 // save a3 to call[j+1]'s frame
s32e a4, a0, -32 // save a4 to call[j]'s frame
s32e a5, a0, -28 // save a5 to call[j]'s frame
s32e a6, a0, -24 // save a6 to call[j]'s frame
s32e a7, a0, -20 // save a7 to call[j]'s frame
rfwo // rotates back to call[i] position

```

WindowUnderflow8:

```

// On entry here: a0-a7 are call[i].reg[0..7] and initially
// contain garbage, a8-a15 are call[i+1].reg[0..7],
// (in particular, a9 is call[i+1]'s stack pointer)
// and must be preserved
l32e a0, a9, -16 // restore a0 from call[i+1]'s frame
l32e a1, a9, -12 // restore a1 from call[i+1]'s frame
l32e a2, a9, -8 // restore a2 from call[i+1]'s frame
l32e a7, a1, -12 // a7 <- call[i-1]'s sp
l32e a3, a9, -4 // restore a3 from call[i+1]'s frame
l32e a4, a7, -32 // restore a4 from call[i]'s frame
l32e a5, a7, -28 // restore a5 from call[i]'s frame
l32e a6, a7, -24 // restore a6 from call[i]'s frame
l32e a7, a7, -20 // restore a7 from call[i]'s frame
rfwu

```

WindowOverflow12:

```

// On entry here: window rotated to call[j]; the registers to be
// saved are a0-a11; a12-a15 must be preserved
// a13 is call[j+1]'s stack pointer
s32e a0, a13, -16 // save a0 to call[j+1]'s frame
l32e a0, a1, -12 // a0 <- call[j-1]'s sp
s32e a1, a13, -12 // save a1 to call[j+1]'s frame
s32e a2, a13, -8 // save a2 to call[j+1]'s frame
s32e a3, a13, -4 // save a3 to call[j+1]'s frame
s32e a4, a0, -48 // save a4 to end of call[j]'s frame
s32e a5, a0, -44 // save a5 to end of call[j]'s frame
s32e a6, a0, -40 // save a6 to end of call[j]'s frame
s32e a7, a0, -36 // save a7 to end of call[j]'s frame
s32e a8, a0, -32 // save a8 to end of call[j]'s frame
s32e a9, a0, -28 // save a9 to end of call[j]'s frame
s32e a10, a0, -24 // save a10 to end of call[j]'s frame
s32e a11, a0, -20 // save a11 to end of call[j]'s frame
rfwo // rotates back to call[i] position

```

WindowUnderflow12:

```

// On entry here: a0-a11 are call[i].reg[0..11] and initially
// contain garbage, a12-a15 are call[i+1].reg[0..3],
// (in particular, a13 is call[i+1]'s stack pointer)
// and must be preserved
l32e a0, a13, -16 // restore a0 from call[i+1]'s frame
l32e a1, a13, -12 // restore a1 from call[i+1]'s frame

```

```

132e a2, a13, -8 // restore a2 from call[i+1]'s frame
132e a11, a1, -12 // a11 <- call[i-1]'s sp
132e a3, a13, -4 // restore a3 from call[i+1]'s frame
132e a4, a11, -48 // restore a4 from end of call[i]'s frame
132e a5, a11, -44 // restore a5 from end of call[i]'s frame
132e a6, a11, -40 // restore a6 from end of call[i]'s frame
132e a7, a11, -36 // restore a7 from end of call[i]'s frame
132e a8, a11, -32 // restore a8 from end of call[i]'s frame
132e a9, a11, -28 // restore a9 from end of call[i]'s frame
132e a10, a11, -24 // restore a10 from end of call[i]'s frame
132e a11, a11, -20 // restore a11 from end of call[i]'s frame
rfwu

```

6.2 Miscellaneous Special Registers Option

The Miscellaneous Special Registers Option provides zero to four scratch registers within the processor readable and writable by `RSR`, `WSR`, and `XSR`. These registers are privileged. They may be useful for some application-specific exception and interrupt processing tasks in the kernel. The `MISC` registers are undefined after reset.

- Prerequisites: None
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

6.2.1 Miscellaneous Special Registers Option Architectural Additions

[Miscellaneous Special Registers Option Processor-Configuration Additions](#) and [Miscellaneous Special Registers Option Processor-State Additions](#) show this option's architectural additions.

Table 117: Miscellaneous Special Registers Option Processor-Configuration Additions

Parameter	Description	Valid Values
NMISC	Number of miscellaneous 32-bit Special Registers	0..4

Table 118: Miscellaneous Special Registers Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ⁷
MISC	NMISC	32	Miscellaneous privileged register	R/W	244-247

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
<p>1. Special Registers are accessed with <code>RSR</code>, <code>WSR</code>, and <code>XSR</code> (<i>Processor Control Instructions</i>). Processor state is listed in <i>Table 127: Alphabetical List of Processor State</i> on page 266.</p>					

6.3 Thread Pointer Option

The Thread Pointer Option provides an additional register to facilitate implementation of Thread Local Storage by operating systems and tools. The register is readable and writable by `RUR` and `WUR`. The register is unprivileged and is undefined after reset.

- Prerequisites: None
- Incompatible options: None
- For more information on option compatibility, see *Purpose of Options* on page 74 and a specific *Xtensa Microprocessor Data Book*.

6.3.1 Thread Pointer Option Architectural Additions

Thread Pointer Option Processor-State Additions shows this option's architectural additions.

Table 119: Thread Pointer Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Register Number ¹
THREADPTR	1	32	Thread pointer	R/W	User 231
<p>1. Special Registers are accessed with <code>RSR</code>, <code>WSR</code>, and <code>XSR</code> (<i>Processor Control Instructions</i> on page 46). Processor state is listed in <i>Table 127: Alphabetical List of Processor State</i> on page 266.</p>					

6.4 Processor ID Option

In some applications there are multiple Xtensa processors executing from the same instruction memory, and there is a need to distinguish one processor from another. This option allows the system logic to provide each processor an identity by reading the `PRID` register. The `PRID` value for each processor is typically in the range 0..`NPROCESSORS-1`, but this is not required. The `PRID` register is constant after reset.

- Prerequisites: None
- Incompatible options: None

- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

6.4.1 Processor ID Option Architectural Additions

[Processor ID Option Processor-State Additions](#) shows this option’s architectural additions.

Table 120: Processor ID Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ⁷
PRID	1	32 ²	Processor Id	R	235

1. Registers with a Special Register assignment are read with the `RSR` instruction. See [Table 127: Alphabetical List of Processor State](#) on page 266.
2. Some implementations may support only the low 16 bits of the `PRID` register.

6.5 Debug Option

The Debug Option implements instruction-counting and breakpoint exceptions for debugging by software or external hardware. The option uses an interrupt level previously defined in the High-Priority Interrupt Option. In some implementations, some debug interrupts may not be masked by `PS.INTLEVEL` (see the *Xtensa Debug Guide*). The Debug Option is useful when configuring a new (not previously debugged) Xtensa processor configuration or for running previously untested software on a processor.

- Prerequisites: [High-Priority Interrupt Option](#) on page 157
- Incompatible options: None
- For more information on option compatibility, see [Purpose of Options](#) on page 74 and a specific *Xtensa Microprocessor Data Book*.

Some of the features listed below are added only when the OCD Option (see the *Xtensa Debug Guide*) is configured in addition to the Debug Option. Those features are included here, under the Debug Option, so that their architectural aspects are documented, but marked as “available only with OCD Option.”

6.5.1 Debug Option Architectural Additions

[Debug Option Processor-Configuration Additions](#) through [Debug Option Instruction Additions](#) show this option’s architectural additions.

Table 121: Debug Option Processor-Configuration Additions

Parameter	Description	Valid Values
DEBUGLEVEL	Debug interrupt level	2..NLEVEL ^{1,2}
NIBREAK	Number of instruction breakpoints (break registers)	0..2
NDBREAK	Number of data breakpoints (break registers)	0..2
SZICOUNT	Number of bits in the ICOUNT register	2, 32

1. NLEVEL is specified in the High-Priority Interrupt Option, *High-Priority Interrupt Option Processor-Configuration Additions*.
2. DEBUGLEVEL must be greater than EXCMLEVEL (see *High-Priority Interrupt Option Processor-Configuration Additions*)

Table 122: Debug Option Processor-State Additions

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
ICOUNT	1	2,32	Instruction count	R/W	236
ICOUNTLEVEL	1	4	Instruction-count level	R/W	237
IBREAKA	NIBREAK	32	Instruction-break address	R/W	128-129
IBREAKENABLE	1	NIBREAK	Instruction-break enable bits	R/W	96
DBREAKA	NDBREAK	32	Data-break address	R/W	144-145
DBREAKC	NDBREAK	8 ²	Data break control	R/W	160-161
DEBUGCAUSE	1	10	Cause of last debug exception	R	233

Register Mnemonic	Quantity	Width (bits)	Register Name	R/W	Special Register Number ¹
DDR ³	1 ³	32	Debug data register	R/W	104

1. Special Registers are accessed with `RSR`, `WSR`, and `XSR` (*Processor Control Instructions*). Processor state is listed in [Table 127: Alphabetical List of Processor State](#) on page 266.
2. See [DBREAKC\[i\] Format](#) for the DBREAKC register format.
3. The DDR register may have separate physical registers for in and out directions in some implementations. The register is only available with the OCD Option, for which the Debug Option is a prerequisite.

Table 123: Debug Option Instruction Additions

Instruction ¹	Format	Definition
BREAK	RRR on page 656	Breakpoint
BREAK.N ²	RRR on page 656	Narrow breakpoint
LDDR32.P	RRR on page 656	Load DDR Register from Memory
SDDR32.P	RRR on page 656	Store DDR Register to Memory

1. These instructions are fully described in [Instruction Descriptions](#) on page 321.
2. Exists only if the Code Density Option described in [Code Density Option](#) on page 82 is configured.

6.5.2 Debug Cause Register

The `DEBUGCAUSE` register contains a coded value giving the reason(s) that the processor took the debug exception. It is implementation-specific whether all applicable bits are set or whether lower-priority conditions are undetected in the presence of higher-priority conditions.

For the priority of the bits in the `DEBUGCAUSE` register, see [Exception Priority under the Exception Option 2](#) on page 145.

[DEBUGCAUSE Register](#) below shows the bits in the `DEBUGCAUSE` register, and [DEBUGCAUSE Fields](#) describes more fully the meaning of each bit.

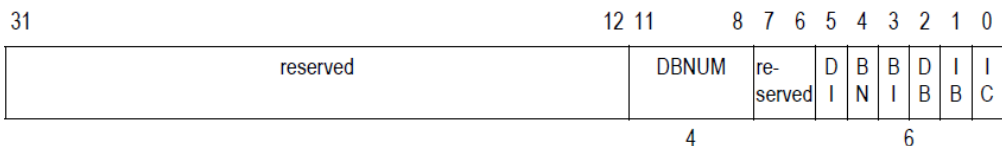


Figure 52: DEBUGCAUSE Register

Table 124: DEBUGCAUSE Fields

Bit	Field	Meaning
0	IC	ICOUNT exception
1	IB	IBREAK exception
2	DB	DBREAK exception
3	BI	BREAK instruction
4	BN	BREAK.N instruction
5	DI	Debug interrupt ¹
11-8	DBNUM	Which of the DBREAK registers matched (added in RA-2004.1 release)
1. The debug interrupt is only available with the OCD Option.		

The `DEBUGCAUSE` register is undefined after processor reset and when `CINTLEVEL < DEBUGLEVEL`.

6.5.3 Using Breakpoints

`BREAK` and `BREAK.N` are 24-bit and 16-bit instructions that simply raise a `DEBUGLEVEL` exception with `DEBUGCAUSE` bit 3 or 4 set, respectively, when executed. Software can replace an instruction with a breakpoint instruction to transfer control to a debug monitor when execution reaches the replaced instruction.

The `BREAK` and `BREAK.N` instructions cannot be used on ROM code, and so the ISA provides a configurable number of instruction-address breakpoint registers. When the processor is about to complete the execution of the instruction fetched from virtual address `IBREAKA[i]`, and `IBREAKENABLEi` is set, it raises an exception instead. It is up to the software to compare the PC to the various `IBREAKA/IBREAKENABLE` pairs to determine which comparison caused the exception.

The processor also provides a configurable number of data-address breakpoint registers. Each breakpoint specifies a naturally aligned power of two-sized block of bytes between one byte and 128 bytes in the processor's address space and whether the break should occur on a load or a store or both. Not all implementations are able to cover a block as large as 128 bytes. If an implementation is not able to cover the full 128 bytes, the upper `MASK` bits in `DBREAKC` will appear hardwired to zero. The lowest address of the covered block of bytes is placed in one of the `DBREAKA` registers. The size of the covered block of bytes is placed in the low bits of the corresponding `DBREAKC` register while the upper two bits of the `DBREAKC` register contain an indication of which access types should raise the exception. The settings for each possible block size are shown in *DBREAK Fields*. The 'x' values under `DBREAKA[i]6..0` allow any naturally aligned address to be specified for that size. The result of other combinations of `DBREAKC` and `DBREAKA` is not defined.

Table 125: DBREAK Fields

Desired DBREAK Size	<code>DBREAKC[i]_{6..0}</code> ¹	<code>DBREAKA[i]_{6..0}</code>
1 Byte	7'b1111111	7'bxxxxxxx
2 Bytes	7'b1111110	7'bxxxxx0
4 Bytes	7'b1111100	7'bxxxx00
8 Bytes	7'b1111000	7'bxxxx000
16 Bytes	7'b1110000	7'bxxx0000
32 Bytes	7'b1100000	7'bxx00000
64 Bytes	7'b1000000	7'bx000000
128 Bytes	7'b0000000	7'b0000000
<p>1. Some upper bits of this field may be hardwired to zero. Any such bits function as if they were hardwired to one.</p>		

When any of the bytes accessed by a load or store matches any of the bytes of the block specified by one of the `DBREAK[i]` register pairs, the processor raises an exception instead of executing the load or store. Specifically, "match" is defined as:

```
(if load then DBREAKC[i]30 else DBREAKC[i]31) and
(DBREAKA[i] >= (125#DBREAKC[i]6..0 and vAddr)) and
(DBREAKA[i] <= (125#DBREAKC[i]6..0 and (vAddr+sz-1)))
```

where `sz` is the number of bytes in the memory access. That is, both the first and last byte of the memory access are masked by $(1^{25} \parallel \text{DBREAKC}[i]_{6..0})$. This operation aligns both byte addresses to the `DBREAK` size indicated by `DBREAKC[i]` as in [DBREAK Fields](#). If the first or last masked address or any address between them matches `DBREAKA[i]` then a match exists. Note that bits in `DBREAKA[i]_{6..0}` corresponding to clear bits in `DBREAKC[i]_{6..0}` should also be clear.

For the `DBREAK` exception, the `DBNUM` field of the `DEBUGCAUSE` register records, as a four bit encoded number, which of the possible `DBREAK[i]` registers raised the exception. If more than one `DBREAK[i]` matches, one of the ones that matched is recorded in `DBNUM`.

The processor clears `IBREAKENABLE` on processor reset; the `IBREAKA`, `DBREAKA`, and `DBREAKC` registers are undefined after reset.

6.5.4 Debug Exceptions

Typically `DEBUGLEVEL` is set to `NLEVEL` (highest priority for maskable interrupts) to allow debugging of other exception handlers. `DEBUGLEVEL` may, in certain cases be set to a lower level than `NLEVEL`.

The relation between the current interrupt level (`CINTLEVEL`, [PS Register Fields](#)) and the specified debug interrupt level (`DEBUGLEVEL`, [Debug Option Processor-Configuration Additions](#)) determine whether debug interrupts can be taken. All debug exceptions (`ICOUNT`, `IBREAK`, `DBREAK`, `BREAK`, `BREAK.N`) are disabled when `CINTLEVEL` \geq `DEBUGLEVEL`. In this case, the `BREAK` and `BREAK.N` instructions perform no operation.

6.5.5 Instruction Counting

The `ICOUNT` register counts instruction completions when `CINTLEVEL` is less than `ICOUNTLEVEL`. Instructions that raise an exception (including the `ICOUNT` exception) do not increment `ICOUNT`. When `ICOUNT` would increment to 0, it instead generates an `ICOUNT` exception. (See [The checkcount Procedure](#) on page 263 for the formal specification.) Because `ICOUNT` has priority ahead of other exceptions (see [Exception Priority under the Exception Option 2](#) on page 145), it is taken even if another exception would have kept the instruction from completing and, therefore, `ICOUNT` from incrementing.

When `ICOUNTLEVEL` is 1, for example, `ICOUNT` stops counting when an interrupt or exception occurs and starts again at the return. Neither the instruction not executed nor the return increment `ICOUNT`, but the re-execution of the instruction does. By this mechanism, the count of instructions can be made the same whether or not the interrupt or exception is taken. When incrementing is turned on or off by `RSIL.PS`, `WSR.PS`, or `XSR.PS` instructions, the state of `CINTLEVEL` and `ICOUNTLEVEL` before the instruction begins determines whether or not the increment is done, as well as whether or not the exception is raised.

Instruction counting may be used to implement single or multi-stepping. For repeatable programs, it can also be used to determine the instruction count of the point of failure, and

allow the program to be re-run up to some point before the point of failure so that the failure can be directly observed with tracing or stepping.

The purpose of the `ICOUNTLEVEL` register is to allow various levels of exception and interrupt processing to be visible or invisible for debugging. An `ICOUNTLEVEL` setting of 1 causes single-stepping to ignore exceptions and interrupts, whereas setting it to `DEBUGLEVEL` allows the programmer to debug exception and interrupt handlers. The `ICOUNTLEVEL` register should only be modified while `PS.INTLEVEL` or `PS.EXCM` is high enough that both before and after the change, `ICOUNT` is not incrementing.

This discussion applies for `SZICOUNT=32`. If `SZICOUNT=2`, then the upper bits appear as all ones for all purposes of reading with `RSR` and for comparing. In that case, `WSR.ICOUNT` affects only the lower two bits. The result is that the feature is really only useful for single stepping because it cannot count very far. But in other respects it behaves in the same fashion.

`ICOUNTLEVEL` is undefined after reset. The `ICOUNT` register should be read or written only when `CINTLEVEL` is greater than or equal to `ICOUNTLEVEL`, where the `ICOUNT` register is not incrementing (see [INTSET - Special Register #226 \(write\)](#)).

6.5.6 Debug Registers

Like all special registers, the `IBREAKA`, `IBREAKENABLE`, `DBREAKA`, `DBREAKC`, and `ICOUNT` registers are read and written using the `RSR`, `WSR`, and `XSR` instructions. [DBREAKC\[i\] Format](#) shows the format of the `DBREAKC` registers and [DBREAKC\[i\] Register Fields](#) shows the `DBREAKC[i]` register fields.

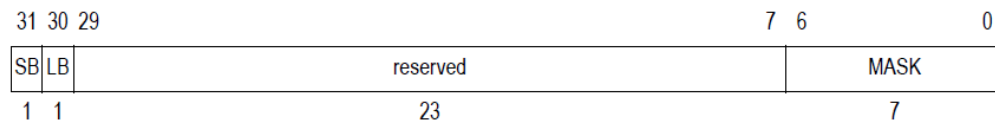


Figure 53: `DBREAKC[i]` Format

Table 126: `DBREAKC[i]` Register Fields

Field	Width (bits)	Definition
MASK	7	Mask specifying which bits of <code>vAddr</code> to compare to <code>DBREAKA[i]</code> See Using Breakpoints on page 259 for details.
LB	1	Load data address match enable 0 → no exception on load data address match 1 → exception on load data address match

Field	Width (bits)	Definition
SB	1	Store data address match enable 0 → no exception on store data address match 1 → exception on store data address match
reserved		Reserved for future use Writing a non-zero value to one of these fields results in undefined processor behavior.

6.5.7 Debug Interrupts

The debug data register (DDR) allows communication between a debug supervisor executing on the processor and a debugger executing on a remote host. To stop an executing program being debugged, the external debugger may use a debug interrupt. Debug interrupts share the same vector as other debug exceptions (InterruptVector[DEBUGLEVEL]), but are distinguished by the setting of the DI bit of the DEBUGCAUSE register. Both the DDR register and the debug interrupt are only available with the OCD option (see the *Xtensa Debug Guide*).

The INTENABLE register (see [Interrupt Option](#) on page 151) does not contain a bit for the debug interrupt.

6.5.8 The `checkIcount` Procedure

The definition of `checkIcount`, used in [Little-Endian Fetch Semantics](#) on page 52 and [Big-Endian Fetch Semantics](#) on page 53, is:

```

procedure checkIcount ()
  if CINTLEVEL < ICOUNTLEVEL then
    if ICOUNT ≠ -1 then
      ICOUNT ← ICOUNT + 1
    elseif CINTLEVEL < DEBUGLEVEL then
      -- Exception
      DEBUGCAUSE ← 1
      EPC[DEBUGLEVEL] ← PC
      EPS[DEBUGLEVEL] ← PS
      PC ← InterruptVector[DEBUGLEVEL]
      PS.EXCM ← 1
      PS.INTLEVEL ← DEBUGLEVEL
    endif
  endif
endprocedure checkIcount

```


7. Processor State

Topics:

- [Processor State Alphabetical List](#)
- [General Registers](#)
- [Program Counter](#)
- [Special Registers](#)
- [User Registers](#)
- [TLB Entries](#)
- [Additional Register Files](#)
- [Caches and Local Memories](#)

The architectural state of an Xtensa machine consists of its `AR` register file, a `PC`, Special Registers, User Registers, TLB entries, and additional register files (added by options and designer's TIE). The Windowed Register Option causes an increase in the physical size of the `AR` register file but does not change the number of registers visible by instructions at any given time. To a lesser extent, caches and local memories can be considered in some ways to be architectural state. The subsections of this chapter cover each of these categories of state in turn.

The Floating-Point Coprocessor Option adds the `FR` register file and two User Registers called `FCR` and `FSR`. The Region Protection Option and the MMU Option add ITLB Entries and DTLB Entries. Other options add only Special Registers. Designer's TIE may add User Registers, and additional register files. Only the `AR` register file, the `PC`, and `SAR` are in all Xtensa processors.

7.1 Processor State Alphabetical List

[Alphabetical List of Processor State](#) contains an alphabetical list of all Cadence-defined registers that make up Xtensa processor state, including the registers added by all architectural options. The Special Register number column of most entries contains a Special Register number, which can be looked up in [Special Registers](#) on page 272 for more information. The last column contains a reference where more information can be found in the pages following the table.

Table 127: Alphabetical List of Processor State

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
ACCHI	Accumulator high bits	MAC16 Option on page 91	17	ACCHI – Special Register #17
ACCLO	Accumulator low bits	MMU Option on page 217	16	ACCLO – Special Register #16
APB0CFG	APB0 Configuration Register	APB Option	117	APB0CFG – Special Register #117
AR	Address registers (general registers)	Core Architecture on page 77	—	General Registers on page 271
ATOMCTL	Atomic Operation Control	Conditional Store Option on page 118	99	ATOMCTL – Special Register #99
BR	Boolean registers / register file	Boolean Option on page 97	4	BR – Special Register #4
CACHEADDRDIS	Disable Data Cache by Address Region	Memory Protection Unit Option on page 205	98	CACHEADDRDIS – Special Register #90
CCOMPARE0..2	Cycle number to interrupt	Timer Interrupt Option on page 161	240-242	CCOMPARE0..2 – Special Register #240-242
CCOUNT	Cycle count	Timer Interrupt Option on page 161	234	CCOUNT – Special Register #234
CPENABLE	Coprocessor enable bits	Coprocessor Context Option on page 149	224	CPENABLE – Special Register #224

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
DBREAKA0..2	Data break address	Debug Option on page 256	144-145	DBREAKA0..1 - Special Register #144-145
DBREAKC0..2	Data break control	Debug Option on page 256	160-161	DBREAKC0..1 - Special Register #160-161
DEBUGCAUSE	Cause of last debug exception	Debug Option on page 256	233	DEBUGCAUSE - Special Register #233
DDR	Debug data register	Debug Option on page 256	104	DDR - Special Register #104
DEPC	Double exception PC	Exception Option 2 on page 126	192	DEPC - Special Register #192
DTLB Entries	Data TLB entries	Region Protection Option on page 196	—	TLB Entries on page 317
DTLBCFG	Data TLB configuration	MMU Option on page 217	92	DTLBCFG - Special Register #92
EPC1	Level-1 exception PC	Exception Option 2 on page 126	177	EPC1 - Special Register #177
EPC2..7	High level exception PC	High-Priority Interrupt Option on page 157	178-183	EPC2..7 - Special Register #178-183
EPS2..7	High level exception PS	High-Priority Interrupt Option on page 157	194-199	EPS2..7 - Special Register #194-199
ERACCESS	External Register Access Control	Memory Protection Unit Option on page 205	95	ERACCESS - Special Register #95
EXCCAUSE	Cause of last exception	Exception Option 2 on page 126	232	EXCCAUSE - Special Register #232

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
EXCSAVE1	Level-1 exception save location	Exception Option 2 on page 126	209	EXCSAVE1 – Special Register #209
EXCSAVE2..7	High level exception save location	High-Priority Interrupt Option on page 157	210-215	EXCSAVE2..7– Special Register #210-215
EXCVADDR	Exception virtual address	Exception Option 2 on page 126	238	EXCVADDR – Special Register #238
FCR	Floating point control register	Floating-Point Coprocessor Option on page 99	—	FCR – User Register #232
FR	Floating point registers	Floating-Point Coprocessor Option on page 99	—	Additional Register Files on page 318
FSR	Floating point status register	Floating-Point Coprocessor Option on page 99	—	FSR – User Register #233
IBREAKA0..2	Instruction break address	Debug Option on page 256	128-129	IBREAKA0..1 – Special Register #128-129
IBREAKC0..2	Instruction break control		192-193	
IBREAKENABLE	Instruction break enable bits	Debug Option on page 256	96	IBREAKENABLE – Special Register #96
ICOUNT	Instruction count	Debug Option on page 256	236	ICOUNT – Special Register #236
ICOUNTLEVEL	Instruction count level	Debug Option on page 256	237	ICOUNTLEVEL – Special Register #237
INTCLEAR	Clear requests in INTERRUPT	Interrupt Option on page 151	227	INTCLEAR – Special Register #227

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
INTENABLE	Interrupt enable bits	Interrupt Option on page 151	228	INTENABLE – Special Register #228
INTERRUPT	Interrupt request bits	Interrupt Option on page 151	226	INTERRUPT – Special Register #226 (read)
INTSET	Set Requests in INTERRUPT	Interrupt Option on page 151	226	INTSET – Special Register #226 (write)
ITLB Entries	Instruction TLB entries	Region Protection Option on page 196	—	TLB Entries on page 317
ITLBCFG	Instruction TLB configuration	MMU Option on page 217	91	ITLBCFG – Special Register #91
LBEG	Loop-begin address	Loop Option on page 84	0	LBEG – Special Register #0
LCOUNT	Loop count	Loop Option on page 84	2	LCOUNT – Special Register #2
LEND	Loop-end address	Loop Option on page 84	1	LEND – Special Register #1
LITBASE	Literal base	Extended L32R Option on page 86	5	LITBASE – Special Register #5
M0..3	MAC16 data registers/register file	MAC16 Option on page 91	32-35	M0..3 – Special Register #32-35
MECR	Memory error check register	Memory ECC/Parity Option on page 168	110	MECR – Special Register #110
MEMCTL	L1 Memory Controls	Core Architecture on page 77	97	MEMCTL – Special Register #97
MEPC	Memory error PC register	Memory ECC/Parity Option on page 168	106	MEPC – Special Register #106
MEPS	Memory error PS register	Memory ECC/Parity Option on page 168	107	MEPS – Special Register #107

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
MESAVE	Memory error save register	Memory ECC/Parity Option on page 168	108	MESAVE – Special Register #108
MESR	Memory error status register	Memory ECC/Parity Option on page 168	109	MESR – Special Register #109
MEVADDR	Memory error virtual addr register	Memory ECC/Parity Option on page 168	111	MEVADDR – Special Register #111
MISC0..3	Misc register 0-3	Miscellaneous Special Registers Option on page 254	244-247	MISC0..3 – Special Register #244-247
MPUCFG	Number of MPU Foreground Segments	Memory Protection Unit Option on page 205	92	MPUCFG – Special Register #92
MPUENB	Enables for each Foreground Segment	Memory Protection Unit Option on page 205	90	MPUENB – Special Register #90
MR	MAC16 Data registers/register file	MMU Option on page 217	32-35	M0..3 – Special Register #32-35
OPMODE	Operation Mode - Impl defined	Core Architecture on page 77	119	OPMODE – Special Register #119
PC	Program counter	Core Architecture on page 77	—	Program Counter on page 272
PRID	Processor Id	Processor ID Option on page 255	235	PRID – Special Register #235
PS	Processor status	See PS Register Fields	230	PS – Special Register #230
PTEVADDR	Page table virtual address	MMU Option on page 217	83	PTEVADDR – Special Register #83
RASID	Ring ASID values	MMU Option on page 217	90	RASID – Special Register #90
SAR	Shift-amount register	Core Architecture on page 77	3	SAR – Special Register #3

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
SCOMPARE1	Expected data value for S32C1I	Multiprocessor Synchronization Option on page 115	12	SCOMPARE1 - Special Register #12
THREADPTR	Thread pointer	Thread Pointer Option on page 255	—	THREADPTR - User Register #231
VECBASE	Vector Base	Relocatable Vector Option on page 147	231	VECBASE - Special Register #231
WindowBase	Base of current AR window	Windowed Register Option on page 240	72	WindowBase - Special Register #72
WindowStart	Call-window start bits	Windowed Register Option on page 240	73	WindowStart - Special Register #73
<p>1. Used in RSR, WSR, and XSR instructions.</p> <p>2. FCR & FSR are User Registers where most are system registers. These names are used in RUR and WUR instructions.</p>				

7.2 General Registers

Many Xtensa instructions operate on the general registers in the `AR` register file. The instructions view sixteen such registers at any given time and usually have a 4-bit specifier field in the instruction for each register they access.

These general registers are named address registers (`AR`) to distinguish them from the many different types of data registers that can be added to the instruction set ([Additional Register Files](#) on page 318). Although the `AR` registers can be used to hold data as well, they are involved with both the instruction set and the execution pipeline in such a way as to make them ideally suited to contain addresses and the information used to compute addresses. They are ideally suited to computing branch conditions and targets as well, and as such fill the role of general registers in the Xtensa instruction set.

When the Windowed Register Option is enabled, there are actually more than sixteen registers in the `AR` register file. The windowed register ABI, described in [The Windowed Register and CALL0 ABIs](#) on page 684, can be used in combination with the Windowed Register Option to make use of the additional registers and avoid many of the register saves and restores that would normally be associated with calls and returns. This improves both the speed and the code density of Xtensa processors.

Reads from and writes to the `AR` register file are always interlocked by hardware. No synchronization instructions are ever required by them.

The contents of the `AR` register file are undefined after reset.

7.3 Program Counter

The program counter (`PC`) holds the address of the next instruction to execute. It is updated by instructions as they execute. Non-branch instructions simply increment it by their length. Branch instructions, when taken, load it with a new value. Call and return instructions exist, which move values between the `PC` and general register `AR[0]`. Options such as the Loop Option change the `PC` in other useful ways.

Changes to and uses of the `PC` are always interlocked by hardware. No synchronization instructions are ever required by them.

7.4 Special Registers

Special Registers hold the majority of the state added to the processor by the Options listed in [Architectural Options](#) on page 73. [Numerical List of Special Registers](#) shows the Special Registers in numerical order with references to a more detailed description. Special Registers not listed in [Numerical List of Special Registers](#) are reserved for future use.

Table 128: Numerical List of Special Registers

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
LBEG	Loop-begin address	Loop Option on page 84	0	LBEG - Special Register #0
LEND	Loop-end address	Loop Option on page 84	1	LEND - Special Register #1
LCOUNT	Loop count	Loop Option on page 84	2	LCOUNT - Special Register #2
SAR	Shift-amount register	Core Architecture on page 45	3	SAR - Special Register #3
BR	Boolean registers / register file	Boolean Option on page 97	4	BR - Special Register #4
LITBASE	Literal base	Extended L32R Option on page 86	5	LITBASE - Special Register #5

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
SCOMPARE1	Expected data value for S32C1I	Conditional Store Option on page 118	12	SCOMPARE1 - Special Register #12
ACCLO	Accumulator low bits	MAC16 Option on page 91	16	ACCLO - Special Register #16
ACCHI	Accumulator high bits	MAC16 Option on page 91	17	ACCHI - Special Register #17
M0..3 / MR	MAC16 data registers / register file	MAC16 Option on page 91	32-35	M0..3 - Special Register #32-35
WindowBase	Base of current AR window	Windowed Register Option on page 240	72	WindowBase - Special Register #72
WindowStart	Call-window start bits	Windowed Register Option on page 240	73	WindowStart - Special Register #73
PTEVADDR	Page table virtual address	MMU Option on page 217	83	PTEVADDR - Special Register #83
RASID	Ring ASID values	MMU Option on page 217	90	RASID - Special Register #90
MPUENB	Enables for each Foreground Segment	Memory Protection Unit Option on page 205	90	MPUENB - Special Register #90
ITLBCFG	Instruction TLB configuration	MMU Option on page 217	91	ITLBCFG - Special Register #91
DTLBCFG	Data TLB configuration	MMU Option on page 217	92	DTLBCFG - Special Register #92
MPUCFG	Number of MPU Foreground Segments	Memory Protection Unit Option on page 205	92	MPUCFG - Special Register #92
ERACCESS	External Register Access Control	Core Architecture on page 45	95	ERACCESS - Special Register #95
CACHEADDRDIS	Disable Data Cache by Address Region	Memory Protection Unit Option on page 205	98	CACHEADDRDIS - Special Register #90

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
IBREAKENABLE	Instruction break enable bits	Debug Option on page 256	96	IBREAKENABLE – Special Register #96
MEMCTL	L1 Memory Controls	Core Architecture on page 45	97	MEMCTL – Special Register #97
ATOMCTL	Atomic Operation Control	Conditional Store Option on page 118 or Exclusive Access Option on page 123	99	ATOMCTL – Special Register #99
DDR	Debug data register	Debug Option on page 256	104	DDR – Special Register #104
MEPC	Memory error PC register	Memory ECC/Parity Option on page 168	106	MEPC – Special Register #106
MEPS	Memory error PS register	Memory ECC/Parity Option on page 168	107	MEPS – Special Register #107
MESAVE	Memory error save register	Memory ECC/Parity Option on page 168	108	MESAVE – Special Register #108
MESR	Memory error status register	Memory ECC/Parity Option on page 168	109	MESR – Special Register #109
MECR	Memory error check register	Memory ECC/Parity Option on page 168	110	MECR – Special Register #110
MEVADDR	Memory error virtual addr register	Memory ECC/Parity Option on page 168	111	MEVADDR – Special Register #111
IBREAKA0..1	Instruction break address	Debug Option on page 256	128-129	IBREAKA0..1 – Special Register #128-129
DBREAKA0..1	Data break address	Debug Option on page 256	144-145	DBREAKA0..1 – Special Register #144-145
DBREAKC0..1	Data break control	Debug Option on page 256	160-161	DBREAKC0..1 – Special Register #160-161

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
EPC1	Level-1 exception PC	Exception Option 2 on page 126	177	EPC1 – Special Register #177
EPC2..7	High level exception PC	High-Priority Interrupt Option on page 157	178-183	EPC2..7 – Special Register #178-183
IBREAKC0..2	Instruction break control		192-193	
DEPC	Double exception PC	Exception Option 2 on page 126	192	DEPC – Special Register #192
EPS2..7	High level exception PS	High-Priority Interrupt Option on page 157	194-199	EPS2..7 – Special Register #194-199
EXCSAVE1	Level-1 exception save location	Exception Option 2 on page 126	209	EXCSAVE1 – Special Register #209
EXCSAVE2..7	High level exception save location	High-Priority Interrupt Option on page 157	210-215	EXCSAVE2..7 – Special Register #210-215
CPENABLE	Coprocessor enable bits	Coprocessor Context Option on page 149	224	CPENABLE – Special Register #224
INTERRUPT	Interrupt request bits	Interrupt Option on page 151	226	INTERRUPT – Special Register #226 (read)
INTSET	Set requests in INTERRUPT	Interrupt Option on page 151	226	INTSET – Special Register #226 (write)
INTCLEAR	Clear requests in INTERRUPT	Interrupt Option on page 151	227	INTCLEAR – Special Register #227
INTENABLE	Interrupt enable bits	Interrupt Option on page 151	228	INTENABLE – Special Register #228
PS	Processor status	See PS Register Fields	230	PS – Special Register #230

Name ¹	Description	Required Configuration Option	Special Register Number	More Detail
VECBASE	Vector Base	Relocatable Vector Option on page 147	231	VECBASE - Special Register #231
EXCCAUSE	Cause of last exception	Exception Option 2 on page 126	232	EXCCAUSE - Special Register #232
DEBUGCAUSE	Cause of last debug exception	Debug Option on page 256	233	DEBUGCAUSE - Special Register #233
CCOUNT	Cycle count	Timer Interrupt Option on page 161	234	CCOUNT - Special Register #234
PRID	Processor Id	Processor ID Option on page 255	235	PRID - Special Register #235
ICOUNT	Instruction count	Debug Option on page 256	236	ICOUNT - Special Register #236
ICOUNTLEVEL	Instruction count level	Debug Option on page 256	237	ICOUNTLEVEL - Special Register #237
EXCVADDR	Exception virtual address	Exception Option 2 on page 126	238	EXCVADDR - Special Register #238
CCOMPARE0..2	Cycle number to generate interrupt	Timer Interrupt Option on page 161	240-242	CCOMPARE0..2 - Special Register #240-242
MISC0..3	Misc register 0-3	Miscellaneous Special Registers Option on page 254	244-247	MISC0..3 - Special Register #244-247
1. Used in RSR, WSR, and XSR instructions.				

[Reading and Writing Special Registers](#) on page 277 describes the process of reading and writing these special registers, while the sections that follow describe groups of specific Special Registers in more detail. A table is included for each special register, which includes

information specific to that special register. The gray shaded rows describe the information that is contained in the unshaded rows immediately below them.

The first row shows the Special Register number, the Name (which is used in the `RSR.*`, `WSR.*`, and `XSR.*` instruction names), a short description, and the value immediately after reset.

The second row shows the Option that creates the Special Register, the count or number of such special registers, the number of bits in the special register, whether access to the register is privileged (requires `CRING=0`) or not, and whether `XSR.*` is a legal instruction or not. The Option that creates the Special Register is described in [Architectural Options](#) on page 73 including more information on each Special Register.

The third row shows the function of the `WSR.*` and `RSR.*` instructions for this Special Register. The function of the `XSR.*` instruction is the combination of the `RSR.*` and the `WSR.*` instructions.

The fourth row shows the other instructions that affect or are affected by this Special Register.

The last row of each Special Register's table shows what SYNC instructions are required when using this Special Register. If no SYNC instructions are ever required, the row is left out. On the left is an instruction or other action that changes the value of the Special Register. On the right is an instruction or other action that makes use of the value of the Special Register. If a SYNC instruction is required for this pair of operations to work as they should, it is listed in the middle. Wherever a `DSYNC` is required an `ISYNC`, `RSYNC`, or `ESYNC` can also be used. Wherever an `ESYNC` is required an `ISYNC` or `RSYNC` can also be used. Wherever an `RSYNC` is required an `ISYNC` can also be used. Note that the 16-bit versions (`*.N`) of 24-bit instructions are not listed separately but always have exactly the same requirements. Versions T1050 and before required additional SYNC instructions in some cases as described in [Reduction of SYNC Instruction Requirements](#).

Because of the importance of its subfields, the `PS` Special Register is a special case. Its subfields are listed in the same format as special registers. The synchronizations needed simply because the register has been written are listed under the entire register, while the synchronizations needed because the value of a subfield has been changed are listed under the subfield.

7.4.1 Reading and Writing Special Registers

The `RSR.*`, `WSR.*`, and `XSR.*` instructions access the special registers. The accesses to the Special Registers act as separate instructions in many ways. For the full instruction name, replace the `.*` in the instructions with the name as given in the Special Register Tables in this section.

Each `RSR.*` instruction moves a value from a Special Register to a general (`AR`) register. Each `WSR.*` instruction moves a value from a general (`AR`) register to a Special Register. Each `XSR.*` instruction exchanges the values in a general (`AR`) register and a Special

Register. Some Special Registers do not allow this exchange. The Special Register tables in this section show which do and do not allow this exchange. The exchange takes place with the two reads taking place first, and then the two writes. In some cases, the write of a Special Register can affect other behavior of the processor. In general, this behavior change does not occur until after the instruction (including `XSR.*`) has completed execution.

Some of the Special Registers have interactions with other instructions or with hardware execution. These interactions are also listed in the Special Register tables in this section. Because modification of many Special Registers is an unusual occurrence, synchronization instructions are used to ensure that their values have propagated everywhere before certain other actions are allowed to take place. Some of the interlocks would be costly in performance or in gates if done in hardware, and the synchronization instructions can be the most efficient solution.

7.4.2 LOOP Special Registers

The Loop Option adds the three registers shown in [LBEG - Special Register #0](#) through [LCOUNT - Special Register #2](#) for controlling zero overhead loops. When the PC reaches `LEND`, it executes at `LBEG` instead and decrements `LCOUNT`. When `LCOUNT` reaches zero, the loop back does not occur.

Table 129: LBEG - Special Register #0

SR#	Name	Description			Reset Value
0	LBEG	Loop begin - address of beginning of zero overhead loop			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
Loop Option on page 84		1	32	No	Yes
WSR Function			RSR Function		
LBEG ← AR[t]			AR[t] ← LBEG		
Other Changes to the Register			Other Effects of the Register		
LOOP/LOOPGTZ/LOOPNEZ			Branch at end of zero overhead loop		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR LBEG ⇒ ISYNC ⇒ Potential branch caused by attempt to execute LEND					

Table 130: LEND - Special Register #1

SR#	Name	Description			Reset Value
1	LEND	Loop end - address of instruction after zero overhead loop			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Loop Option</i> on page 84		1	32	No	Yes
WSR Function			RSR Function		
LEND ← AR[t]			AR[t] ← LEND		
Other Changes to the Register			Other Effects of the Register		
LOOP/LOOPGTZ/LOOPNEZ			Branch at end of zero overhead loop		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR LEND ⇒ ISYNC ⇒ Potential branch caused by attempt to execute LEND					

Table 131: LCOUNT - Special Register #2

SR#	Name	Description			Reset Value
2	LCOUNT	Loop count remaining			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Loop Option</i> on page 84		1	32	No	Yes
WSR Function			RSR Function		
LCOUNT ← AR[t]			AR[t] ← LCOUNT		
Other Changes to the Register			Other Effects of the Register		
LOOP/LOOPGTZ/LOOPNEZ			Branch at end of zero overhead loop		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR LCOUNT ⇒ ESYNC ⇒ RSR/XSR LCOUNT					
WSR/XSR LCOUNT ⇒ ISYNC ⇒ Potential branch caused by attempt to execute LEND					

SR#	Name	Description	Reset Value
WSR/XSR LCOUNT to zero \Rightarrow ISYNC \Rightarrow WSR/XSR PS.EXCM with zero (for protection)			

7.4.3 MAC16 Special Registers

The MAC16 Option adds the six registers described in [ACCL0 – Special Register #16](#) through [M0..3 – Special Register #32-35](#).

Table 132: ACCL0 – Special Register #16

SR#	Name	Description			Reset Value
16	ACCL0	Accumulator - low bits			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
MAC16 Option on page 91		1	32	No	Yes
WSR Function			RSR Function		
ACC _{31..0} \leftarrow AR[t]			AR[t] \leftarrow ACC _{31..0}		
Other Changes to the Register			Other Effects of the Register		
MUL.* / MULA.* / MULS.* / UMUL.*			MULA.* / MULS.*		

Table 133: ACCHI – Special Register #17

SR#	Name	Description			Reset Value
17	ACCHI	Accumulator - high bits			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
MAC16 Option on page 91		1	8	No	Yes
WSR Function			RSR Function		
ACC _{39..32} \leftarrow AR[t] _{7..0} Undefined if AR[t] _{31..8} \neq AR[t] _{7²⁴}			AR[t] \leftarrow ACC _{39²⁴..32} ACC _{39..32}		
Other Changes to the Register			Other Effects of the Register		

SR#	Name	Description	Reset Value
		MUL.* / MULA.* / MULS.* / UMUL.*	MULA.* / MULS.*

Table 134: m0..3 – Special Register #32-35

SR#	Name	Description	Reset Value
32-35	m0..3 / MR ¹	MAC16 data registers / register file ¹	Undefined
Option	Count	Bits	Privileged?
<i>MAC16 Option</i> on page 91	4	32	No
WSR Function²		RSR Function²	
M[sr _{1..0}] ← AR[t]		AR[t] ← M[sr _{1..0}]	
Other Changes to the Register		Other Effects of the Register	
LDDEC/LDINC/MULA*.LDDEC/MULA*.LDINC		MUL.*D*/MULA.*D*/MULS.*D*	
<ol style="list-style-type: none"> 1. These registers are known as MR[0..3] in hardware and as m0..3 in the software. 2. sr_{1..0} refers to the low two bits of the sr field in the RSR, WSR, or XSR instruction. 			

7.4.4 Other Unprivileged Special Registers

The SAR Special Register is included in the Xtensa Core Architecture, while the BR, LITBASE, and SCOMPARE1 Special Registers are added by the options shown along with other information about them in [SAR – Special Register #3](#) through [SCOMPARE1 – Special Register #12](#).

Table 135: SAR – Special Register #3

SR#	Name	Description	Reset Value
3	SAR	Shift amount register	Undefined
Option	Count	Bits	Privileged?
Core Architecture (see Core Architecture on page 77)	1	6	No
WSR Function		RSR Function	

SR#	Name	Description	Reset Value
		$SAR \leftarrow AR[t]_{5..0}$ Undefined if $AR[t]_{31..6} \neq 0^{26}$	$AR[t] \leftarrow 0^{26} \parallel SAR$
Other Changes to the Register		Other Effects of the Register	
SSL/SSR/SSAI/SSA8B/SSA8L		SLL/SRL/SRA/SRC	

Table 136: BR - Special Register #4

SR#	Name	Description	Reset Value		
4	BR / b0..15 ¹	Boolean register / register file ¹	Undefined		
Option		Count	Bits	Privileged?	XSR Legal?
<i>Boolean Option</i> on page 97		1	16	No	Yes
WSR Function			RSR Function		
$BR \leftarrow AR[t]_{15..0}$ Undefined if $AR[t]_{31..16} \neq 0^{16}$			$AR[t] \leftarrow 0^{16} \parallel BR$		
Other Changes to the Register			Other Effects of the Register		
ALL4/ALL8/ANDB/ANDBC/ANY4/ANY8/ ORB/ORBC/XORB/OEQ.S/OLE.S/OLT.S/ UEQ.S/ULE.S/ULT.S/UN.S/User TIE			ALL4/ALL8/ANDB/ANDBC/ANY4/ANY8/ ORB/ORBC/XORB/ BF/BT/MOVF/MOVF.S/MOVT/MOVT.S		
1. This register is known as Special Register BR or as individual Boolean bits b0..15.					

Table 137: LITBASE - Special Register #5

SR#	Name	Description	Reset Value		
5	LITBASE	Literal base register	bit-0 clear ¹		
Option		Count	Bits	Privileged?	XSR Legal?
<i>Extended L32R Option</i> on page 86		1	21	No	Yes

SR#	Name	Description	Reset Value
WSR Function		RSR Function	
LITBASE \leftarrow AR[t] _{31..12} 0 ¹¹ AR[t] ₀ Undefined if AR[t] _{11..1} \neq 0 ¹¹		AR[t] \leftarrow LITBASE _{31..12} 0 ¹¹ LITBASE ₀	
Other Changes to the Register		Other Effects of the Register	
		L32R	
1. After reset bit-0 is clear but the remainder of the register is undefined.			

Table 138: SCOMPARE1 - Special Register #12

SR#	Name	Description	Reset Value
12	SCOMPARE1	Comparison register for the S32C1I instruction	Undefined
Option	Count	Bits	Privileged?
<i>Conditional Store Option</i> on page 118	1	32	No
WSR Function	RSR Function		XSR Legal?
SCOMPARE1 \leftarrow AR[t]	AR[t] \leftarrow SCOMPARE1		Yes
Other Changes to the Register		Other Effects of the Register	
		S32C1I	

7.4.5 Processor Status Special Register

The Processor Status Special Register is made up of multiple fields with different purposes within the processor. They are combined into one register to simplify the saving and restoring of state for exceptions, interrupts, and context switches. [PS - Special Register #230](#) describes the register as a whole, while [PS.INTLEVEL - Special Register #230 \(part\)](#) through [PS.WOE - Special Register #230 \(part\)](#) describe the individual pieces of the register in a similar format.

The synchronization section of [PS - Special Register #230](#) gives requirements that must be met whenever the PS register is written regardless of whether any of its bits are changed. The synchronization sections of [PS.INTLEVEL - Special Register #230 \(part\)](#) through [PS.WOE -](#)

[Special Register #230 \(part\)](#) give requirements that must be met only if that portion of the PS register is being modified.

Table 139: PS - Special Register #230

SR#	Name	Description			Reset Value
230	PS	Miscellaneous processor state			0x10 or 0x1F ¹
Option		Count	Bits	Privileged?	XSR Legal?
Exception Option 2 on page 126		1	15	Yes	Yes
WSR Function			RSR Function		
$PS \leftarrow 0^{13} \parallel AR[t]_{18..16} \parallel 0^4 \parallel AR[t]_{11..0}$ PS.RING should be changed only when CEXCM=1 before the instruction making the change.			AR[t] ← PS		
Other Changes to the Register			Other Effects of the Register		
CALL[X]4-12/RFE/RFD0/RFDD/RFW0/RFWU/RFI RSIL/WAITI/SETCTXTI/interrupts/exceptions			CALL[X]4-12/ENTRY/RETW/interrupts/loop-back Privileged-instructions/ld-st-instructions/exceptions		
Instruction ⇒ xSYNC ⇒ Instruction					
See following entries for subfields of PS. Write to PS.X means a write to PS that changes subfield X.					
1. PS is 5'h1F after reset if the.Interrupt Option is configured but reads as 5'h10 if it is not.					

Table 140: PS.INTLEVEL - Special Register #230 (part)

SR#	Name	Description			Reset Value
230 Part	PS.INTLEVEL	Interrupt level mask part of PS (PS - Special Register #230)			0x0 or 0xF ¹
Option		Count	Bits	Privileged?	XSR Legal?
Interrupt Option on page 151		1	4	Yes	Yes
WSR Function			RSR Function		
(see PS - Special Register #230)			(see PS - Special Register #230)		

SR#	Name	Description	Reset Value
Other Changes to the Register		Other Effects of the Register	
RFI/RFDD/RFDO/RSIL/WAITI/ Hi-level-interrupts/debug-exceptions/NMI		RSIL/interrupts/debug-exceptions	
Instruction ⇒ xSYNC ⇒ Instruction			
<p>Write to PS . INTLEVEL is a write to PS that changes subfield INTLEVEL.</p> <p>WSR/XSR PS . INTLEVEL ⇒ RSYNC ⇒ Change in accepting interrupts</p> <p>If PS . EXCM and PS . INTLEVEL are both changed in the same WSR . PS or XSR . PS instruction in such a way that a particular interrupt is forbidden both before and after the instruction, there will be no cycle during the instruction where the interrupt may be taken. Thus PS . EXCM may be cleared and PS . INTLEVEL raised (or PS . EXCM set and PS . INTLEVEL lowered) in the same instruction and no gap is opened between them.</p> <p>WSR/XSR PS . INTLEVEL ⇒ DSYNC ⇒ Change in taking debug exception (interrupt level)</p> <p>RFI/RFDD/RFDO/RSIL/WAITI ⇒ (none) ⇒ RSIL or change in accepting interrupts/debug-exceptions</p> <p>Hi-level-interrupts/debug-excep/NMI ⇒ (none) ⇒ RSIL or change in accepting interrupts/debug-exceptions</p>			
<p>1. PS . INTLEVEL is 4'hF after reset if the Interrupt Option is configured but reads as 4'h0 if it is not.</p>			

Table 141: PS . EXCM – Special Register #230 (part)

SR#	Name	Description	Reset Value	
230 Part	PS . EXCM	Exception mask part of PS (<i>PS – Special Register #230</i>)	0x1	
Option	Count	Bits	Privileged?	XSR Legal?
<i>Exception Option 2</i> on page 126	1	1	Yes	Yes
WSR Function		RSR Function		
(see <i>PS – Special Register #230</i>)		(see <i>PS – Special Register #230</i>)		
Other Changes to the Register		Other Effects of the Register		
RFI/RFDD/RFDO/RFE/RFWO/RFWU interrupts/exceptions		CALL[X]4-12/ENTRY/RETW/interrupts/loop-back lfetch/privileged-instr/ld-st-instructions/exceptions		


SR#	Name	Description	Reset Value
Instruction ⇒ xSYNC ⇒ Instruction			
Write to PS . EXCM is a write to PS that changes subfield EXCM.			
WSR/XSR PS . EXCM ⇒ ISYNC ⇒ Changes in instruction fetch privilege			
WSR/XSR PS . EXCM ⇒ RSYNC ⇒ Change in accepting Interrupts			
If PS . EXCM and PS . INTLEVEL are both changed in the same WSR . PS or XSR . PS instruction in such a way that a particular interrupt is forbidden both before and after the instruction, there will be no cycle during the instruction where the interrupt may be taken. Thus PS . EXCM may be cleared and PS . INTLEVEL raised (or PS . EXCM set and PS . INTLEVEL lowered) in the same instruction without a gap in interrupt masking.			
WSR/XSR PS . EXCM to one ⇒ (none) ⇒ Restore non-zero LCOUNT value			
WSR/XSR LCOUNT to zero ⇒ ISYNC ⇒ WSR/XSR PS . EXCM with zero (for protection)			
WSR/XSR PS . EXCM ⇒ ESYNC ⇒ CALL [X] 4-12/ENTRY/RETW			
 Note: In the Windowed Register Option, any instruction with an AR register operand can cause overflow exceptions.			
WSR/XSR PS . EXCM ⇒ DSYNC ⇒ Changes in data fetch privilege			
WSR/XSR PS . EXCM ⇒ (none) ⇒ Double exception vector or not			
RFI/RFDD/RFDO/RFE ⇒ (none) ⇒ Anything			
RFWO/RFWU ⇒ (none) ⇒ Anything			
Interrupts/exceptions ⇒ (none) ⇒ Anything.			

Table 142: PS . UM - Special Register #230 (part)

SR#	Name	Description	Reset Value	
230 Part	PS . UM	User vector mode part of PS (<i>PS - Special Register #230</i>)	0x0	
Option	Count	Bits	Privileged?	XSR Legal?
<i>Exception Option 2</i> on page 126	1	1	Yes	Yes
WSR Function		RSR Function		
(see <i>PS - Special Register #230</i>)		(see <i>PS - Special Register #230</i>)		
Other Changes to the Register		Other Effects of the Register		
RFI/RFDD/RFDO		RSIL/level-1-interrupts		


SR#	Name	Description	Reset Value
		general-exceptionsdebug-exceptions	
Instruction ⇒ xSYNC ⇒ Instruction			
Write to PS .UM is a write to PS that changes subfield UM.			
WSR/XSR PS .UM ⇒ RSYNC ⇒ Level-1-interrupts/general-exceptions/debug-exceptions			
 Note: In the Windowed Register Option, any instruction with an AR register operand can cause overflow exceptions.			

Table 143: PS .RING – Special Register #230 (part)

SR#	Name	Description	Reset Value
230 Part	PS .RING	Ring part of PS (<i>PS – Special Register #230</i>)	0x0
Option	Count	Bits	Privileged?
<i>MMU Option</i> on page 217	1	2	Yes
WSR Function		RSR Function	
(see <i>PS – Special Register #230</i>)		(see <i>PS – Special Register #230</i>)	
Other Changes to the Register		Other Effects of the Register	
RFI/RFDD/RFDO		Hi-level-interrupts/debug-exception/ Privileged-instructions/ld-st-instructions	
Instruction ⇒ xSYNC ⇒ Instruction			
Write to PS .RING is a write to PS that changes subfield RING.			
WSR/XSR PS .RING ⇒ ISYNC ⇒ Changes in instruction fetch privilege			
WSR/XSR PS .RING ⇒ DSYNC ⇒ Changes in data fetch privilege			

Table 144: PS .OWB – Special Register #230 (part)

SR#	Name	Description	Reset Value
230 Part	PS .OWB	Old window base part of PS (<i>PS – Special Register #230</i>)	0x0


SR#	Name	Description			Reset Value
Option		Count	Bits	Privileged?	XSR Legal?
Windowed Register Option on page 240		1	4	Yes	Yes
WSR Function			RSR Function		
(see PS – Special Register #230)			(see PS – Special Register #230)		
Other Changes to the Register			Other Effects of the Register		
RFI/RFDD/RFDO/overflow-or-underflow-exception			RFWO/RFWU/RSIL/hi-level-interrupt/debug-exception		

Table 145: PS.CALLINC – Special Register #230 (part)

SR#	Name	Description			Reset Value
230 Part	PS.CALLINC	Call increment part of PS (PS – Special Register #230)			0x0
Option		Count	Bits	Privileged?	XSR Legal?
Windowed Register Option on page 240		1	2	Yes	Yes
WSR Function			RSR Function		
(see PS – Special Register #230)			(see PS – Special Register #230)		
Other Changes to the Register			Other Effects of the Register		
CALL[X]4-12/RFI/RFDD/RFDO			ENTRY/RSIL/hi-level-interrupt/debug-exception		

Table 146: PS.WOE – Special Register #230 (part)

SR#	Name	Description			Reset Value
230 Part	PS.WOE	Window overflow enable part of PS (PS – Special Register #230)			0x0
Option		Count	Bits	Privileged?	XSR Legal?
Windowed Register Option on page 240		1	1	Yes	Yes

SR#	Name	Description	Reset Value
WSR Function		RSR Function	
(see PS - Special Register #230)		(see PS - Special Register #230)	
Other Changes to the Register		Other Effects of the Register	
RFI/RFDD/RFDO		CALL4-12/CALLX4-12/ENTRY/RETW/RSIL/ Hi-level-interrupt/debug-exception/overflow-exception	
Instruction ⇒ xSYNC ⇒ Instruction			
Write to PS.WOE is a write to PS that changes subfield WOE. WSR/XSR PS.WOE ⇒ RSYNC ⇒ CALL4-12/CALLX4-12/ENTRY/RETW WSR/XSR PS.WOE ⇒ RSYNC ⇒ Overflow-exception			
 Note: In the Windowed Register Option, any instruction with an AR register operand can cause overflow exceptions.			

7.4.6 Windowed Register Option Special Registers

The Windowed Register Option Special registers are described in [WindowBase - Special Register #72](#) and [WindowStart - Special Register #73](#).

Table 147: WindowBase - Special Register #72

SR#	Name	Description	Reset Value
72	WindowBase	Base of current AR register window	Undefined
Option	Count	Bits	Privileged?
Windowed Register Option on page 240	1	log2(NAREG/4)	Yes
WSR Function		RSR Function	
WindowBase ← AR[t] _{X-1..0} Undefined if AR[t] _{31..X} ≠ 0 ^{32-X} X = log2(NAREG/4)		AR[t] ← 0 ^{32-X} WindowBase X = log2(NAREG/4)	
Other Changes to the Register		Other Effects of the Register	

SR#	Name	Description	Reset Value
		ENTRY/MOVSP/RETW/RFW*/ROTW Overflow/underflow-exception	Any instruction which accesses the AR register file
Instruction ⇒ xSYNC ⇒ Instruction			
WSR/XSR WINDOWBASE ⇒ RSYNC ⇒ Any use or def of an ARregister			

Table 148: WindowStart - Special Register #73

SR#	Name	Description	Reset Value		
73	WindowStart	Call-window start bits	Undefined		
Option		Count	Bits	Privileged?	XSR Legal?
	<i>Windowed Register Option</i> on page 240	1	NAREG/4	Yes	Yes (future?)
WSR Function			RSR Function		
WindowStart \leftarrow AR[t] _{NAREG/4-1..0} Undefined if AR[t] _{31..NAREG/4} \neq 0 ^{32-NAREG/4}			AR[t] \leftarrow 0 ^{32-NAREG/4} WindowStart		
Other Changes to the Register			Other Effects of the Register		
ENTRY/MOVSP/RETW/RFWO/RFWU			Any instruction which accesses the AR register file		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR WINDOWSTART ⇒ RSYNC ⇒ Any use of an AR register when CWOE=1					
WSR/XSR WINDOWSTART ⇒ RSYNC ⇒ Any def of an AR register when CWOE=1					

7.4.7 Memory Management Special Registers

The Special Registers for managing memory are described in [PTEVADDR - Special Register #83](#) through [DTLBCFG - Special Register #92](#).

Table 149: PTEVADDR - Special Register #83

SR#	Name	Description			Reset Value
83	PTEVADDR	Virtual address for page table lookups			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>MMU Option</i> on page 217		1	32	Yes	Yes
WSR Function			RSR Function		
$\text{PTEVADDR}_{\text{VABITS}-1..X} \leftarrow \text{AR}[t]_{\text{VABITS}-1..X}$ $X = \text{VABITS} + \log_2(\text{PTEbytes}) - \min(\text{PTEPageSizes})$			$\text{AR}[t] \leftarrow \text{PTEVADDR}_{\text{VABITS}-1..Y} 0^Y$ $Y = \log_2(\text{PTEbytes})$		
Other Changes to the Register			Other Effects of the Register		
			Any instruction/data address translation		
Instruction \Rightarrow xSYNC \Rightarrow Instruction					
WSR/XSR PTEVADDR \Rightarrow ISYNC \Rightarrow Any instruction access that might miss the ITLB					
WSR/XSR PTEVADDR \Rightarrow DSYNC \Rightarrow Any load/store access that might miss the DTLB					

Table 150: RASID - Special Register #90

SR#	Name	Description			Reset Value
90	RASID	Current ASID values for each protection ring			0x04030201
Option		Count	Bits	Privileged?	XSR Legal?
<i>MMU Option</i> on page 217		1	32	Yes	Yes
WSR Function			RSR Function		
$\text{RASID} \leftarrow \text{AR}[t]_{31..8} \ 0^7 \ 1^1$			$\text{AR}[t] \leftarrow \text{RASID}$		
Other Changes to the Register			Other Effects of the Register		
			Any instruction/data address translation		
Instruction \Rightarrow xSYNC \Rightarrow Instruction					

SR#	Name	Description	Reset Value
WSR/XSR RASID ⇒ ISYNC ⇒ Instruction address translation that depends on the change			
WSR/XSR RASID ⇒ DSYNC ⇒ Data address translation that depends on the change			

Table 151: MPUENB - Special Register #90

SR#	Name	Description			Reset Value
90	MPUENB	Enables for each Foreground Segment			0x00000000
Option		Count	Bits	Privileged?	XSR Legal?
<i>Memory Protection Unit Option</i> on page 205		1	Variable ¹	Yes	Yes
WSR Function			RSR Function		
MPUENB ← AR[t] _{(NFOREGROUNDSEGMENTS-1)..0}			AR[t] ← MPUENB		
Other Changes to the Register			Other Effects of the Register		
			Any instruction/data address translation		
Instruction ⇒ xSYNC ⇒ Instruction					
1. The width is the configuration parameter NFOREGROUNDSEGMENTS.					

Table 152: ITLBCFG - Special Register #91

SR#	Name	Description			Reset Value
91	ITLBCFG	Instruction TLB configuration			0x00000000
Option		Count	Bits	Privileged?	XSR Legal?
<i>MMU Option</i> on page 217		1	32	Yes	Yes
WSR Function			RSR Function		
ITLBCFG AR[t]			AR[t] ← ITLBCFG		

SR#	Name	Description	Reset Value
Affected ways should be invalidated after change.			
Other Changes to the Register		Other Effects of the Register	
		Any instruction address translation	
Instruction ⇒ xSYNC ⇒ Instruction			
WSR/XSR I TLBCFG ⇒ ISYNC ⇒ Instruction address translation that depends on the change			

Table 153: DTLBCFG - Special Register #92

SR#	Name	Description	Reset Value		
92	DTLBCFG	Data TLB configuration	0x00000000		
Option		Count	Bits	Privileged?	XSR Legal?
MMU Option on page 217		1	32	Yes	Yes
WSR Function			RSR Function		
DTLBCFG ← AR[t] Affected ways should be invalidated after change.			AR[t] ← DTLBCFG		
Other Changes to the Register			Other Effects of the Register		
			Any data address translation		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR DTLBCFG ⇒ DSYNC ⇒ Any data address translation that depends on the change					

Table 154: MPUCFG - Special Register #92

SR#	Name	Description	Reset Value		
92	MPUCFG	Number of MPU Foreground Segments	NFOREGROUND-SEGMENTS		
Option		Count	Bits	Privileged?	XSR Legal?

SR#	Name	Description			Reset Value
	Memory Protection Unit Option on page 205	1	7	Yes	No
WSR Function			RSR Function		
If secure mode configured, MPUCFG[8] ← MPUCFG[8] AR[t][8] else no action on write			AR[t] ← MPUCFG		
Other Changes to the Register			Other Effects of the Register		
Instruction ⇒ xSYNC ⇒ Instruction					

Table 155: CACHEADDRDIS - Special Register #98

SR#	Name	Description			Reset Value
98	CACHEADDRDIS	Disable Data Cache by Address Region			0x00000000
Option		Count	Bits	Privileged?	XSR Legal?
Memory Protection Unit Option on page 205		1	8	Yes	Yes
WSR Function			RSR Function		
CACHEADDRDIS ← AR[t]			AR[t] ← CACHEADDRDIS		
Other Changes to the Register			Other Effects of the Register		
			Any data access to cache		
Instruction ⇒ xSYNC ⇒ Instruction					

7.4.8 Exception Option 2 Support Special Registers

The Special Registers that provide information for the handling of an exception are described in [EXCCAUSE - Special Register #232](#) through [DEBUGCAUSE - Special Register #233](#).

Table 156: EXCCAUSE - Special Register #232

SR#	Name	Description			Reset Value
232	EXCCAUSE	Exception cause register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
Exception Option 2 on page 126		1	6	Yes	Yes
WSR Function			RSR Function		
EXCCAUSE ← AR[t] _{5..0} Undefined if AR[t] _{31..6} ≠ 0 ²⁶			AR[t] ← 0 ²⁶ EXCCAUSE		
Other Changes to the Register			Other Effects of the Register		
Exception or interrupt					

Table 157: EXCVADDR - Special Register #238

SR#	Name	Description			Reset Value
238	EXCVADDR	Exception virtual address register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
Exception Option 2 on page 126		1	32	Yes	Yes
WSR Function			RSR Function		
EXCVADDR ← AR[t]			AR[t] ← EXCVADDR AR[t] is undefined if CEXCM = 0		
Other Changes to the Register			Other Effects of the Register		
Some exceptions (see Exception Causes), (see MMU Option Auto-Refill TLB Ways and PTE Format on page 231)					

Table 158: VECBASE – Special Register #231

SR#	Name	Description			Reset Value
231	VECBASE	Vector Base			User Defined ¹
Option		Count	Bits	Privileged?	XSR Legal?
<i>Relocatable Vector Option</i> on page 147		1	32-n ²	Yes	Yes
WSR Function ²			RSR Function ²		
VECBASE ← AR[t]			AR[t] ← VECBASE		
Other Changes to the Register			Other Effects of the Register		
			Exception Vector Locations		
<p>1. The reset value of VECBASE is set by the user as part of the configuration.</p> <p>2. Implementations usually place some alignment requirement on VECBASE, and may not implement some number of low-order bits of VECBASE. In such cases, WSR will set only the implemented bits and RSR will return zeros for unimplemented bits.</p>					

Table 159: MESR – Special Register #109

SR#	Name	Description			Reset Value
109	MESR	Memory error status register			32'hXXXX0C00
Option		Count	Bits	Privileged?	XSR Legal?
<i>Memory ECC/Parity Option</i> on page 168		1	32	Yes	Yes
WSR Function			RSR Function		
MESR ← AR[t]			AR[t] ← MESR		
Other Changes to the Register			Other Effects of the Register		
Memoryerror-exception, memory error without exception			Controls memory error logic		
Instruction ⇒ xSYNC ⇒ Instruction					

SR#	Name	Description	Reset Value
WSR/XSR MESR ⇒ ISYNC ⇒ Change in error behavior on instruction memories			
WSR/XSR MESR ⇒ DSYNC ⇒ Change in error behavior on data memories			

Table 160: MECCR - Special Register #110

SR#	Name	Description			Reset Value
110	MECCR	Memory error check register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Memory ECC/Parity Option</i> on page 168		1	22	Yes	Yes
WSR Function			RSR Function		
MECCR ← AR[t]			AR[t] ← MECCR		
Other Changes to the Register			Other Effects of the Register		
Memory error-exception, memory error without exception, Loads when MESR[9] is set.			Stores when MESR[9] is set.		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR MECCR ⇒ ISYNC ⇒ Check bit write to instruction memories					
WSR/XSR MECCR ⇒ DSYNC ⇒ Check bit write to data memories					

Table 161: MEVADDR - Special Register #111

SR#	Name	Description			Reset Value
111	MEVADDR	Memory error virtual address register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Memory ECC/Parity Option</i> on page 168		1	32	Yes	Yes
WSR Function			RSR Function		
MEVADDR ← AR[t]			AR[t] ← MEVADDR		

SR#	Name	Description	Reset Value
Other Changes to the Register		Other Effects of the Register	
Memoryerror-exception, memory error without exception			

Table 162: DEBUGCAUSE - Special Register #233

SR#	Name	Description	Reset Value		
233	DEBUGCAUSE	Debug cause register	Undefined		
Option		Count	Bits	Privileged?	XSR Legal?
Debug Option on page 256		1	12	Yes	No
WSR Function			RSR Function		
Reserved			$AR[t] \leftarrow 0^{20} \parallel \text{DEBUGCAUSE}$		
Other Changes to the Register			Other Effects of the Register		
Debug exception or interrupt					

7.4.9 Exception Option 2 State Special Registers

The Special Registers that save the PC and PS values and an initial register value for each of the levels are described in [EPC1 - Special Register #177](#) through [DEPC - Special Register #192](#).

Table 163: EPC1 - Special Register #177

SR#	Name	Description	Reset Value		
177	EPC1	Exception PC [1]	Undefined		
Option		Count	Bits	Privileged?	XSR Legal?
Exception Option 2 on page 126		1	32	Yes	Yes
WSR Function			RSR Function		
$EPC[1] \leftarrow AR[t]$			$AR[t] \leftarrow EPC[1]$		

SR#	Name	Description	Reset Value
Other Changes to the Register		Other Effects of the Register	
General-exception/overflow-or-underflow-exception		RFE/RFUE/RFWO/RFWU	

Table 164: EPC2..7 - Special Register #178-183

SR#	Name	Description	Reset Value
178-183	EPC2..7	Exception PC [2..7]	Undefined
Option	Count	Bits	Privileged?
<i>High-Priority Interrupt Option</i> on page 157	NLEVEL +NNMI-1	32	Yes
WSR Function¹		RSR Function¹	
EPC[sr _{3..0}] ← AR[t]		AR[t] ← EPC[sr _{3..0}] AR[t] is undefined if sr _{3..0} > NLEVEL+NNMI	
Other Changes to the Register		Other Effects of the Register	
Level[sr _{3..0}] - Interrupt/debug-exception/NMI		RFI[sr _{3..0}]/RFDO/RFDD	
1. sr _{3..0} refers to the low four bits of the sr field in the RSR, WSR, or XSR instruction.			

Table 165: DEPC - Special Register #192

SR#	Name	Description	Reset Value
192	DEPC	Double exception PC	Undefined
Option	Count	Bits	Privileged?
<i>Exception Option 2</i> on page 126	1	32	Yes
WSR Function		RSR Function	
DEPC ← AR[t]		AR[t] ← DEPC	
Other Changes to the Register		Other Effects of the Register	

SR#	Name	Description	Reset Value
Double exception		RFDE	

Table 166: MEPC - Special Register #106

SR#	Name	Description	Reset Value	
106	MEPC	Memory error PC register	Undefined	
Option	Count	Bits	Privileged?	XSR Legal?
<i>Memory ECC/Parity Option</i> on page 168	1	32	Yes	Yes
WSR Function		RSR Function		
MEPC ← AR[t]		AR[t] ← MEPC AR[t] is undefined unless MESR[0] is set.		
Other Changes to the Register		Other Effects of the Register		
Memoryerror-exception		RFME		

Table 167: EPS2..7 - Special Register #194-199

SR#	Name	Description	Reset Value	
194-199	EPS2..7	Exception processor status register [2..7]	Undefined	
Option	Count	Bits	Privileged?	XSR Legal?
<i>High-Priority Interrupt Option</i> on page 157	NLEVEL +NNMI-1	32	Yes	Yes
WSR Function ¹		RSR Function ¹		
EPS[sr _{3..0}] ← AR[t]		AR[t] ← EPS[sr _{3..0}] AR[t] is undefined if sr _{3..0} > NLEVEL+NNMI		
Other Changes to the Register		Other Effects of the Register		
Level[sr _{3..0}]-Interrupt/debug-exception/NMI		RFI[sr _{3..0}]/RFDO/RFDD		

SR#	Name	Description	Reset Value
1. $sr_{3..0}$ refers to the low four bits of the sr field in the RSR, WSR, or XSR instruction.			

Table 168: MEPS - Special Register #107

SR#	Name	Description			Reset Value
107	MEPS	Memory error PS register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
Memory ECC/Parity Option on page 168		1	32	Yes	Yes
WSR Function			RSR Function		
MEPS \leftarrow AR[t]			AR[t] \leftarrow MEPS AR[t] is undefined unless MESR[0] is set.		
Other Changes to the Register			Other Effects of the Register		
Memoryerror-exception			RFME		

Table 169: EXCSAVE1 - Special Register #209

SR#	Name	Description			Reset Value
209	EXCSAVE1	Exception save register[1]			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
Exception Option 2 on page 126		1	32	Yes	Yes
WSR Function			RSR Function		
EXCSAVE[1] \leftarrow AR[t]			AR[t] \leftarrow EXCSAVE[1]		
Other Changes to the Register			Other Effects of the Register		

Table 170: EXCSAVE2..7- Special Register #210-215

SR#	Name	Description			Reset Value
210-215	EXCSAVE2..7	Exception save register[2..7]			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>High-Priority Interrupt Option</i> on page 157		NLEVEL +NNMI-1	32	Yes	Yes
WSR Function ¹			RSR Function ¹		
EXCSAVE[sr _{3..0}] ← AR[t]			AR[t] ← EXCSAVE[sr _{3..0}] AR[t] is undefined if sr _{3..0} > NLEVEL+NNMI		
Other Changes to the Register			Other Effects of the Register		
1. sr _{3..0} refers to the low four bits of the sr field in the RSR, WSR, or XSR instruction.					

Table 171: MESAVE- Special Register #108

SR#	Name	Description			Reset Value
108	MESAVE	Memory error save register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Memory ECC/Parity Option</i> on page 168		1	32	Yes	Yes
WSR Function			RSR Function		
MESAVE ← AR[t]			AR[t] ← MESAVE		
Other Changes to the Register			Other Effects of the Register		

7.4.10 Interrupt Option Special Registers

The Special Registers that manage interrupt handling are described in [INTERRUPT – Special Register #226 \(read\)](#) through [INTENABLE – Special Register #228](#).

Table 172: INTERRUPT – Special Register #226 (read)

SR#	Name	Description			Reset Value
226	INTERRUPT	Interrupt pending register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
Interrupt Option on page 151		1	NINTERRUPT	Yes	No
WSR Function			RSR Function		
see INTSET – Special Register #226 (write) and INTCLEAR – Special Register #227			$AR[t] \leftarrow 0^{32-NINTERRUPT} \parallel INTERRUPT$		
Other Changes to the Register			Other Effects of the Register		
Assertion/deassertion of interrupt signals/ WSR.CCOMPAREn			Pipeline takes interrupt		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR INTSET ⇒ ESYNC ⇒ RSR INTERRUPT WSR INTCLEAR ⇒ ESYNC ⇒ RSR INTERRUPT					

Table 173: INTSET – Special Register #226 (write)

SR#	Name	Description			Reset Value
226	INTSET	Interrupt set register			No separate state
Option		Count	Bits	Privileged?	XSR Legal?
Interrupt Option on page 151		1	NINTERRUPT	Yes	No
WSR Function			RSR Function		
$INTERRUPT \leftarrow INTERRUPT \text{ or } AR[t]_{x-1..0}$			see INTERRUPT – Special Register #226 (read)		

SR#	Name	Description	Reset Value
		Undefined if $AR[t]_{31..X} \neq 0^{32-X}$ $X = NINTERRUPT$ Only software interrupt bits can be set.	
Other Changes to the Register		Other Effects of the Register	
(State is INTERRUPT)		(State is INTERRUPT)	
Instruction \Rightarrow xSYNC \Rightarrow Instruction			
WSR INTSET \Rightarrow ESYNC \Rightarrow RSR INTERRUPT WSR INTSET \Rightarrow RSYNC \Rightarrow Instruction which must execute after the software interrupt			

Table 174: INTCLEAR - Special Register #227

SR#	Name	Description	Reset Value		
227	INTCLEAR	Interrupt clear register	No separate state		
Option		Count	Bits	Privileged?	XSR Legal?
Interrupt Option on page 151		1	NINTERRUPT	Yes	No
WSR Function			RSR Function		
INTERRUPT \leftarrow INTERRUPT and not $AR[t]_{X-1..0}$ Undefined if $AR[t]_{31..X} \neq 0^{32-X}$ $X = NINTERRUPT$ Bits in $AR[t]_{X-1..0}$ may be set without causing harm. Only bits which can be cleared by this write are affected.			$AR[t] \leftarrow$ undefined ³²		
Other Changes to the Register			Other Effects of the Register		
(State is INTERRUPT)			(State is INTERRUPT)		
Instruction \Rightarrow xSYNC \Rightarrow Instruction					
WSR INTCLEAR \Rightarrow ESYNC \Rightarrow RSR INTERRUPT WSR INTCLEAR \Rightarrow RSYNC \Rightarrow Instruction which must execute after the cleared interrupt					

Table 175: INTENABLE - Special Register #228

SR#	Name	Description			Reset Value
228	INTENABLE	Interrupt enable register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Interrupt Option</i> on page 151		1	NINTERRUPT	Yes	Yes
WSR Function			RSR Function		
$INTENABLE \leftarrow AR[t]_{NINTERRUPT-1..0}$ Undefined if $AR[t]_{31..X} \neq 0^{32-X}$ $X = NINTERRUPT$			$AR[t] \leftarrow 0^{32-NINTERRUPT} INTENABLE$		
Other Changes to the Register			Other Effects of the Register		
			Pipeline takes interrupt		
Instruction \Rightarrow xSYNC \Rightarrow Instruction					
WSR/XSR INTENABLE \Rightarrow ESYNC \Rightarrow RSR/XSR INTENABLE					
WSR/XSR INTENABLE \Rightarrow RSYNC \Rightarrow Any instruction which must wait for INTENABLE changes					

7.4.11 Timing Special Registers

The Special Registers that manage instruction counting and cycle counting, including timer interrupts, are described in *ICOUNT - Special Register #236* through *CCOMPARE0..2 - Special Register #240-242*.

Table 176: ICOUNT - Special Register #236

SR#	Name	Description			Reset Value
236	ICOUNT	Instruction count register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Debug Option</i> on page 256		1	2 or 32	Yes	Yes
WSR Function			RSR Function		
$ICOUNT \leftarrow AR[t]$			$AR[t] \leftarrow ICOUNT$		

SR#	Name	Description	Reset Value
Write when $CINTLEVEL \geq ICOUNTLEVEL$		Defined only when $CINTLEVEL \geq ICOUNTLEVEL$	
Other Changes to the Register		Other Effects of the Register	
Increment on appropriate cycles		Debug exception	
Instruction \Rightarrow xSYNC \Rightarrow Instruction			
WSR/XSR ICOUNT \Rightarrow ESYNC \Rightarrow RSR/XSR ICOUNT			
WSR/XSR ICOUNT \Rightarrow ISYNC \Rightarrow Ending $CINTLEVEL \geq ICOUNTLEVEL$			

Table 177: ICOUNTLEVEL - Special Register #237

SR#	Name	Description	Reset Value
237	ICOUNTLEVEL	Instruction count level register	Undefined
Option	Count	Bits	Privileged?
<i>Debug Option</i> on page 256	1	4	Yes
WSR Function		RSR Function	
$ICOUNTLEVEL \leftarrow AR[t]_{3..0}$ Undefined if $AR[t]_{31..4} \neq 0^{28}$ Write when $CINTLEVEL \geq \text{old } ICOUNTLEVEL$ Write when $CINTLEVEL \geq \text{new } ICOUNTLEVEL$		$AR[t] \leftarrow 0^{28} \ ICOUNTLEVEL$	
Other Changes to the Register		Other Effects of the Register	
		Debug exception	
Instruction \Rightarrow xSYNC \Rightarrow Instruction			
WSR/XSR ICOUNTLEVEL \Rightarrow ISYNC \Rightarrow Ending $CINTLEVEL \geq \text{old } ICOUNTLEVEL$			
WSR/XSR ICOUNTLEVEL \leftarrow ISYNC \leftarrow Ending $CINTLEVEL \geq \text{new } ICOUNTLEVEL$			

Table 178: CCOUNT - Special Register #234

SR#	Name	Description			Reset Value
234	CCOUNT	Cycle count register			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Timer Interrupt Option</i> on page 161		1	32	Yes	Yes
WSR Function			RSR Function		
CCOUNT ← AR[t] Precise cycle of write is not defined Not usually written during normal operation.			AR[t] ← CCOUNT Precise cycle of read is not defined.		
Other Changes to the Register			Other Effects of the Register		
Increment each cycle			Generates Timer Interrupt		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR CCOUNT ⇒ ESYNC ⇒ RSR/XSR CCOUNT					

Table 179: CCOMPARE0..2 - Special Register #240-242

SR#	Name	Description			Reset Value
240-242	CCOMPARE0..2	Cycle count compare registers			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Timer Interrupt Option</i> on page 161		NCCOMPARE	32	Yes	Yes
WSR Function¹			RSR Function¹		
CCOMPARE[sr _{1..0}] ← AR[t] INTERRUPT _i ← 0; i is position of timer interrupt			AR[t] ← CCOMPARE[sr _{1..0}] AR[t] is undefined if sr _{1..0} ≥ NCOMPARE		
Other Changes to the Register			Other Effects of the Register		
			Timer Interrupt		

SR#	Name	Description	Reset Value
Instruction ⇒ xSYNC ⇒ Instruction			
WSR/XSR CCOMPARE0..2 ⇒ ESYNC ⇒ RSR/XSR CCOUNT (to ensure CCOUNT < CCOMPARE _n)			
WSR/XSR CCOMPARE0..2 ⇒ RSYNC ⇒ Any instruction which must execute after the update			
1. sr _{1..0} refers to the low two bits of the sr field in the RSR, WSR, or XSR instruction.			

7.4.12 Breakpoint Special Registers

The Special Registers that manage the handling of breakpoint exceptions are described in [IBREAKENABLE - Special Register #96](#) through [DBREAKA0..1 - Special Register #144-145](#).

Table 180: IBREAKENABLE - Special Register #96

SR#	Name	Description	Reset Value		
96	IBREAKENABLE	Instruction breakpoint enable register	0 ^{NIBREAK}		
Option		Count	Bits	Privileged?	XSR Legal?
<i>Debug Option</i> on page 256		1	NIBREAK	Yes	Yes
WSR Function			RSR Function		
IBREAKENABLE ← AR[t] _{NIBREAK-1..0} Undefined if AR[t] _{31..NIBREAK} ≠ 0 ^{32-NIB}			AR[t] ← 0 ^{32-NIBREAK} IBREAKENABLE		
Other Changes to the Register			Other Effects of the Register		
			Any instruction fetch		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR IBREAKENABLE ⇒ ISYNC ⇒ Any instruction access that might raise a breakpoint					

Table 181: IBREAKA0..1 - Special Register #128-129

SR#	Name	Description	Reset Value
128-129	IBREAKA0..1	Instruction breakpoint address registers	Undefined

SR#	Name	Description			Reset Value
Option		Count	Bits	Privileged?	XSR Legal?
<i>Debug Option</i> on page 256		NIBREAK	32	Yes	Yes
WSR Function ¹			RSR Function ¹		
IBREAKA[sr _{3..0}] ← AR[t] Operation is undefined if sr _{3..0} ≥ NIBREAK			AR[t] ← IBREAKA[sr _{3..0}] AR[t] is undefined if sr _{3..0} ≥ NIBREAK		
Other Changes to the Register			Other Effects of the Register		
			Any instruction fetch		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR IBREAKA0..1 ⇒ ISYNC ⇒ Any instruction access which might raise that breakpoint					
1. sr _{3..0} refers to the low four bits of the sr field in the RSR, WSR, or XSR instruction.					

Table 182: DBREAKC0..1 - Special Register #160-161

SR#	Name	Description			Reset Value
160-161	DBREAKC0..1	Data breakpoint control registers			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Debug Option</i> on page 256		NDBREAK	32	Yes	Yes
WSR Function ¹			RSR Function ¹		
DBREAKC[sr _{3..0}] ← AR[t] Operation is undefined if sr _{3..0} ≥ NDBREAK			AR[t] ← DBREAKC[sr _{3..0}] AR[t] is undefined if sr _{3..0} ≥ NDBREAK		
Other Changes to the Register			Other Effects of the Register		
			Any data access		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR DBREAKC0..1 ⇒ DSYNC ⇒ Any load/store access which might raise that breakpoint					

SR#	Name	Description	Reset Value
1. $sr_{3..0}$ refers to the low four bits of the sr field in the RSR, WSR, or XSR instruction.			

Table 183: DBREAKA0..1 - Special Register #144-145

SR#	Name	Description			Reset Value
144-145	DBREAKA0..1	Data breakpoint address registers			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
<i>Debug Option</i> on page 256		NDBREAK	32	Yes	Yes
WSR Function ¹			RSR Function ¹		
DBREAKA[$sr_{3..0}$] \leftarrow AR[t] Operation is undefined if $sr_{3..0} \geq$ NDBREAK			AR[t] \leftarrow DBREAKA[$sr_{3..0}$] AR[t] is undefined if $sr_{3..0} \geq$ NDBREAK		
Other Changes to the Register			Other Effects of the Register		
			Any data access		
Instruction \Rightarrow xSYNC \Rightarrow Instruction					
WSR/XSR DBREAKA0..1 \Rightarrow DSYNC \Rightarrow Any load/store access which might raise that breakpoint					
1. $sr_{3..0}$ refers to the low four bits of the sr field in the RSR, WSR, or XSR instruction.					

7.4.13 Other Privileged Special Registers

The Special Registers for other purposes are described in *PRID - Special Register #235* through *ATOMCTL - Special Register #99*.

Table 184: PRID - Special Register #235

SR#	Name	Description			Reset Value
235	PRID	Processor identification register			Pins
Option		Count	Bits	Privileged?	XSR Legal?
<i>Processor ID Option</i> on page 255		1	32	Yes	No

SR#	Name	Description	Reset Value
WSR Function		RSR Function	
Reserved		AR[t] ← PRID	
Other Changes to the Register		Other Effects of the Register	
Trailing edge of RESET			

Table 185: MMID - Special Register #89

SR#	Name	Description	Reset Value		
89	MMID	Memory map identification register	Undefined		
Option		Count	Bits	Privileged?	XSR Legal?
<i>Trace Port Option</i>		1	32	Yes	No
WSR Function		RSR Function			
ID written to Trace Port		Reserved			
Other Changes to the Register		Other Effects of the Register			

Table 186: DDR - Special Register #104

SR#	Name	Description	Reset Value		
104	DDR	Debug data register	Undefined		
Option		Count	Bits	Privileged?	XSR Legal?
<i>Debug Option</i> on page 256 ¹		1	32	Yes	Yes
WSR Function		RSR Function			
DDR ← AR[t] ²		AR[t] ← DDR ²			
Other Changes to the Register		Other Effects of the Register			

SR#	Name	Description	Reset Value
Instruction ⇒ xSYNC ⇒ Instruction			
WSR/XSR DDR ⇒ ESYNC ⇒ RSR/XSR DDR			
<ol style="list-style-type: none"> 1. The DDR register is actually created by the OCD Option but is listed with the Debug Option, which is a prerequisite for the OCD Option. 2. In some implementations the DDR state is different for reads and writes; WSR.DDR followed by RSR.DDR may not return the original value. 			

Table 187: CPENABLE - Special Register #224

SR#	Name	Description	Reset Value		
224	CPENABLE	Coprocessor enable register	Undefined		
Option		Count	Bits	Privileged?	XSR Legal?
Coprocessor Context Option on page 149		1	1-8	Yes	Yes
WSR Function			RSR Function		
CPENABLE ← AR[t] _{7..0} Undefined if AR[t] _{31..8} ≠ 0 ²⁴			AR[t] ← 0 ²⁴ CPENABLE (Bits corresponding to unused coprocessors are not defined on read.)		
Other Changes to the Register			Other Effects of the Register		
			Every coprocessor instruction		
Instruction ⇒ xSYNC ⇒ Instruction					
WSR/XSR CPENABLE ⇒ RSYNC ⇒ Coprocessor Instruction which is expected to see new value					

Table 188: ERACCESS - Special Register #95

SR#	Name	Description	Reset Value
95	ERACCESS	External Register Access Control	0x00000000

SR#	Name	Description			Reset Value
Option		Count	Bits	Privileged?	XSR Legal?
Core Architecture (see Core Architecture on page 77)		1	1-16	Yes	Yes
WSR Function			RSR Function		
ERACCESS \leftarrow AR[t] _{15..0} Undefined if AR[t] _{31..16} \neq 0 ¹⁶			AR[t] \leftarrow 0 ¹⁶ ERACCESS (Bits corresponding to unused external register sets are not defined on read.)		
Other Changes to the Register			Other Effects of the Register		
			Every RER or WER Instruction		

Table 189: MISC0..3 - Special Register #244-247

SR#	Name	Description			Reset Value
244-247	MISC0..3	Miscellaneous Special Registers Option on page 254			Undefined
Option		Count	Bits	Privileged?	XSR Legal?
Miscellaneous Special Registers Option		NMISC	32	Yes	Yes
WSR Function ¹			RSR Function ¹		
MISC[sr _{1..0}] \leftarrow AR[t]			AR[t] \leftarrow MISC[sr _{1..0}] AR[t] is undefined if sr _{1..0} \geq NMISC		
Other Changes to the Register			Other Effects of the Register		
1. sr _{3..0} refers to the low four bits of the sr field in the RSR, WSR, or XSR instruction.					

Table 190: ATOMCTL - Special Register #99

SR#	Name	Description	Reset Value
99	ATOMCTL	Atomic Operation Control	0x28

SR#	Name	Description			Reset Value
Option		Count	Bits	Privileged?	XSR Legal?
Conditional Store Option on page 118 or Exclusive Access Option on page 123		1	7	Yes	Yes
WSR Function			RSR Function		
ATOMCTL AR[t] _{8..0} Undefined ¹ if AR[t] _{31..9} ≠ 0 ²³ or AR[t] _{7..6} ≠ 0 ²			AR[t] ← 0 ²³ ATOMCTL[8:0]		
Other Changes to the Register			Other Effects of the Register		
GETEX and S32EX modify ATOMCTL[8]			Affects S32C1I, L32EX, S32EX, and GETEX		
Instruction → xSYNC ⇒ Instruction					
<p>1. Bit[8] must also be zero unless the Exclusive Access Option is configured. Bits[5:0] must also be zero unless either the Conditional Store Option or the Region Protection Option is configured.</p>					

Table 191: MEMCTL - Special Register #97

SR#	Name	Description			Reset Value
97	MEMCTL	L1 Memory Controls			0x0 or 0x1 ¹
Option		Count	Bits	Privileged?	XSR Legal?
Core Architecture (see Core Architecture on page 77)		1	22	Yes	Yes
WSR Function			RSR Function		
MEMCTL ← AR[t]			AR[t] ← MEMCTL		
Other Changes to the Register			Other Effects of the Register		
			Loop Buffer Enable, Inst/Data Cache Way Control, Inst/Data Cache Snoop Enable.		

SR#	Name	Description	Reset Value
1.		The reset value is 0x1 if the Loop Option is configured with LoopBufferSize >0 and 0x0 otherwise.	

7.5 User Registers

User Registers hold state added in support of designer's TIE and in some cases of options that Cadence provides. See the *Tensilica Instruction Extension (TIE) Language User's Guide* for more information on adding User Registers to a design. [Numerical List of User Registers](#) shows the User Registers in numerical order with references to a more detailed description. User Registers with numbers greater than or equal to 224 but not listed in [Numerical List of User Registers](#) are reserved for future use.

Table 192: Numerical List of User Registers

Name ¹	Description	Required Configuration Option	User Register Number	More Detail
	Available for designer extensions		0-223	
THREADPTR	Thread pointer	Thread Pointer Option on page 255	231	THREADPTR – User Register #231
FCR	Floating point control register	Floating-Point Coprocessor Option on page 99	232	FCR – User Register #232
FSR	Floating point status register	Floating-Point Coprocessor Option on page 99	233	FSR – User Register #233
1. Used in <code>RUR.*</code> and <code>WUR.*</code> instructions.				

7.5.1 Reading and Writing User Registers

Use the `RUR.*` and `WUR.*` instructions to access the user registers. The accesses to the User Registers act as separate instructions in many ways. Replace the `***` in the instructions with the name of the User Register as specified by the designer or given in [FCR – User Register #232](#) and [FSR – User Register #233](#).

RUR.* instructions move values from a User Register to a general (AR) register. WUR.* instructions move values from a general (AR) register to a User Register. The User Registers are fully interlocked in hardware and do not need SYNC instructions.

7.5.2 The List of User Registers

[THREADPTR - User Register #231](#) through [FSR - User Register #233](#) list detailed information for each of the User Registers that Cadence Options define.

The first row shows the User Register number, the name (which is used in the RUR.*, WUR.* instruction names), a short description, and the value immediately after reset.

The second row shows the Option that creates the User Register, the count or number of such User Registers, the number of bits in the User Register, and whether access to the register is privileged (requires CRING=0) or not. The option that creates the User Register is described in [Architectural Options](#) on page 73 including more information on each User Register.

The third row shows the function of the WUR.* and RUR.* instructions for this User Register.

The fourth row shows the other instructions that affect or are affected by this User Register.

The last row of each User Register's table shows that SYNC instructions are not required.

User Registers 0-223 are reserved for designer's use, and are never used by Cadence Options. User Registers 224-255 can be used by a designer but their use may prohibit compatibility with some Cadence-provided Options either now or in the future. Additional state registers may be added without built-in access instructions.

Table 193: THREADPTR - User Register #231

UR#	Name	Description			Reset Value
231	THREADPTR	Thread pointer			Undefined
Option		Count	Bits	Privileged?	
Thread Pointer Option on page 255		1	32	No	
WUR Function			RUR Function		
THREADPTR ← AR[t]			AR[t] ← THREADPTR		
Other Changes to the Register			Other Effects of the Register		

Table 194: FCR - User Register #232

UR#	Name	Description			Reset Value
232	FCR	Floating point control register			Undefined
Option		Count	Bits	Privileged?	
<i>Floating-Point Coprocessor Option</i> on page 99		1	7	No	
WUR Function			RUR Function		
FCR ← AR[t]			AR[t] ← FCR		
Other Changes to the Register			Other Effects of the Register		
			Most floating point computations		

Table 195: FSR - User Register #233

UR#	Name	Description			Reset Value
233	FSR	Floating point status register			Undefined
Option		Count	Bits	Privileged?	
<i>Floating-Point Coprocessor Option</i> on page 99		1	5	No	
WUR Function			RUR Function		
FSR ← AR[t]			AR[t] ← FSR		
Other Changes to the Register			Other Effects of the Register		
Most floating point computations					

7.6 TLB Entries

Although some information for the instruction and data TLBs is held in the Special Registers, the protection and translation entries themselves are held in a special type of state called ITLB entries and DTLB entries. These entries are added by the Region Protection Option and the MMU Option.

These entries are accessed by special instructions for reading and writing the entries. There are also instructions for probing to see if an entry exists that will match a particular virtual address. In addition, there are instructions for invalidating particular entries. The instructions added for these purposes are listed under the Region Protection Option and the MMU Option.

After changing an Instruction TLB entry, an `ISYNC` must be executed before executing any instruction that is accessed using that TLB. After changing a data TLB entry, a `DSYNC` must be executed before any load or store that uses that entry (see [Region Protection Option Memory Attributes](#) on page 200, [Region Translation Option Formats for Accessing TLB Entries](#) on page 203, [Formats for Writing MMU Option TLB Entries](#) on page 227, and [Format for Invalidating MMU Option TLB Entries](#) on page 230 for more detailed information).

7.7 Additional Register Files

Additional register files also hold state added in support of designer's TIE and in some cases of Cadence-provided Options. There are no built-in instructions for accessing added register files in the same manner as the `RUR.*`, and `WUR.*` instructions can be used to access the user registers. See the *Tensilica Instruction Extension (TIE) Language User's Guide* for more information on adding register files to a design.

As shown in [Alphabetical List of Processor State](#), the Floating-Point Coprocessor Option creates the `FR` register file, which is an instance of this capability in a Cadence-provided Option. The `FR` register file contains sixteen registers of 32 bits each in support of the floating point instruction set. There is no windowing in the `FR` register file.

Reads from and writes to these additional register files are always interlocked by hardware. No synchronization instructions are ever required by them.

The contents of these additional register files are undefined after reset.

7.8 Caches and Local Memories

Local memories are always architectural state. However, for many purposes caches are not architectural state in that they merely reflect the contents of main memory but provide lower latency access for the processor. When considering the cache control instructions added with the caches or the requirements placed upon software for maintaining coherence between processors/devices in their views of memory, caches sometimes act like architectural state.

Self-modifying code is not automatically supported in Xtensa processors. The instruction cache is not kept coherent with main memory because there is no hardware for observing writes to memory and determining whether or not those writes could have any affect on the instruction cache. Any time memory that could possibly be contained in the instruction cache is changed, the OS must ensure that the changes have been written back to system memory

and invalidate either the specific locations that have been changed or else the entire instruction cache. See the description of the `ISYNC` instruction for more details.

In addition, because the instruction unit of the Xtensa processor fetches ahead, synchronization instructions are needed whenever an instruction local memory or instruction cache is modified before it can be certain that the instruction fetch engine will see the changes. For local memories, this means an `ISYNC` instruction is needed after any change to the instruction memory and before the execution of any instruction involved in the change. For instruction caches, this means an `ISYNC` instruction is needed after any change to the cache data, or the cache tag (including the invalidation required when main memory that could possibly be held in the icache is modified) and before the execution of any instruction involved in the change.

The operation of all instructions to data local memory or data cache is fully interlocked in hardware. And except for the instruction fetch discussed above, the operation of all instructions to instruction local memory or instruction cache is fully interlocked in hardware. Loads and stores, tag accesses, cache invalidations, cache line locks/unlocks, prefetches, and write backs all operate in order to the same cache locations because of the hardware interlocking. Accesses to different addresses are not necessarily in order (see [Multiprocessor Synchronization Option](#) on page 115).

Both the data and the tag stores of instruction caches and data caches are ordinary synchronous SRAMs, which are not expected to be defined after reset.

8. Instruction Descriptions

Topics:

- [Instruction Word](#)
- [Instruction Exception Groups](#)
- [Instructions](#)

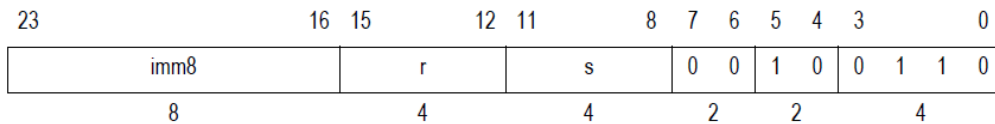
This chapter describes, in alphabetical order, each of the Xtensa ISA instructions in the Core Architecture described in [Core Architecture](#) on page 45, or in Architecture Options described in [Architectural Options](#) on page 73.

Before reading this chapter, Cadence recommends reviewing the notation defined in [Uses Of Instruction Fields](#).

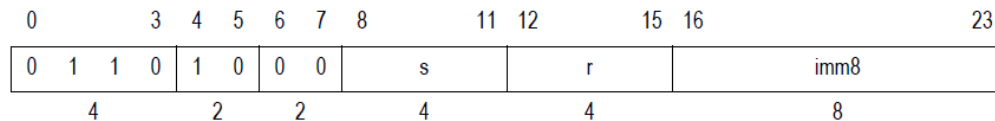
Note that instructions with a “Required Configuration Option” specification other than “Core Architecture” are illegal if the corresponding option is not enabled, and will raise an illegal instruction exception.

8.1 Instruction Word

The instruction word included with each instruction is the little-endian version (see [Bit and Byte Order](#) on page 38 and [Instruction Formats and Opcodes](#) on page 655). The big-endian instruction word may be determined for any instruction by separating the little-endian instruction word at the vertical bars and reassembling the pieces in the reverse order. For example, following is the little-endian instruction word shown for the `BEQI` instruction:



Following is the derived big-endian instruction word for the `BEQI` instruction:



The format listed after the instruction word at the top of each instruction page can also be used along with [Formats](#) on page 656 to derive the big-endian encoding.

For each instruction, the exceptions that can possibly result from its execution are listed. Because many of the potential exceptions are common to a large number of instructions, exception groups are used to save space and improve understanding. [Instruction Exception Groups](#) on page 322 lists the common exception groups that are referenced in the instructions. A reference to one of these groups means that any of the exceptions in the group can be raised by that instruction. Note that the groups often include previous groups.

8.2 Instruction Exception Groups

In the following groups and in the instruction descriptions, `GenExcep()` is a general exception that goes to `UserExceptionVector`, `KernelExceptionVector`, or `DoubleExceptionVector`; the parentheses contain the cause that will appear in `EXCCAUSE`. `DebugExcep()` is a debug exception that goes to the high level interrupt for debug and the parentheses contain the cause that will appear in `DEBUGCAUSE`. `WindowOverExcep` is one of the three sizes of windowed register overflow exceptions¹ and `WindowUnderExcep` is one of the three sizes of windowed register underflow exceptions². After any exceptions in the list there is an option without which that exception cannot be taken.

EveryInst Group:

¹ `WindowOverflow4`, `WindowOverflow8`, or `WindowOverflow12`.

² `WindowUnderflow4`, `WindowUnderflow8`, or `WindowUnderflow12`.

- GenExcep(InstructionFetchErrorCause) if Exception Option 2
- GenExcep(InstTLBMissCause) if Region Protection Option or MMU Option
- GenExcep(InstTLBMultiHitCause) if Region Protection Option or MMU Option
- GenExcep(InstFetchPrivilegeCause) if Region Protection Option or MMU Option
- GenExcep(InstFetchProhibitedCause) if Region Protection Option or MMU Option
- MemoryErrorException on Instruction-fetch if Memory ECC/Parity Option
- DebugExcep(ICOUNT) if Debug Option
- DebugExcep(IBREAK) if Debug Option

EveryInstR Group:

- EveryInst Group (see [EveryInst Group](#))
- WindowOverExcep if Windowed Register Option

Memory Group:

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(LoadStoreErrorCause) if Exception Option 2
- GenExcep(LoadStoreTLBMissCause) if Region Protection Option or MMU Option
- GenExcep(LoadStoreTLBMultiHitCause) if Region Protection Option or MMU Option
- GenExcep(LoadStorePrivilegeCause) if Region Protection Option or MMU Option
- MemoryErrorException on non-Instruction-fetch if Memory ECC/Parity Option

Memory Load Group:

- Memory Group (see [EveryInstR Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- GenExcep(LoadStoreAlignmentCause) if Unaligned Exception Option
- DebugExcep(DBREAK) if Debug Option

Memory Store Group:

- Memory Group (see [EveryInstR Group](#))
- GenExcep(StoreProhibitedCause) if Region Protection Option or MMU Option
- GenExcep(LoadStoreAlignmentCause) if Unaligned Exception Option
- DebugExcep(DBREAK) if Debug Option

Privileged Instruction Group:

- GenExcep(PrivilegedCause) if Exception Option 2

Coprocessor Group:

- GenExcep(Coprocessor0Disabled) if Exception Option 2 and Coprocessor Context Option

8.3 Instructions

The following sections contain instruction descriptions.

8.3.1 ABS—Absolute Value

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	1	1	0	0	0	0	0	r	0	0	0	1	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 45)

Assembler Syntax

```
ABS ar, at
```

Description

ABS calculates the absolute value of the contents of address register `at` and writes it to address register `ar`. Arithmetic overflow is not detected.

Operation

```
AR[r] ← if AR[t]31 then -AR[t] else AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.2 ABS.D—Absolute Value Double

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
1	1	1	1	1	1	1	1	r	s	0	0	0	1	0	0	0	0
4				4				4				4					

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ABS.D fr, fs
```

Description

ABS.D computes the double-precision absolute value of the contents of floating-point register *fs* and writes the result to floating-point register *fr*.

Operation

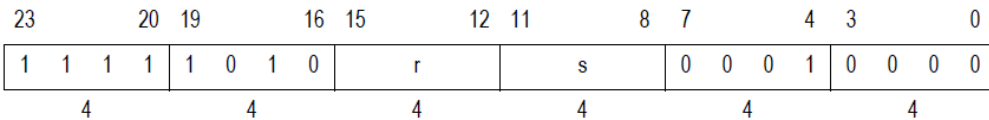
```
FR[r] ← absD(FR[s])
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.3 ABS.S—Absolute Value Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ABS.S fr, fs
```

Description

ABS.S computes the single-precision absolute value of the contents of floating-point register *fs* and writes the result to floating-point register *fr*.

Operation

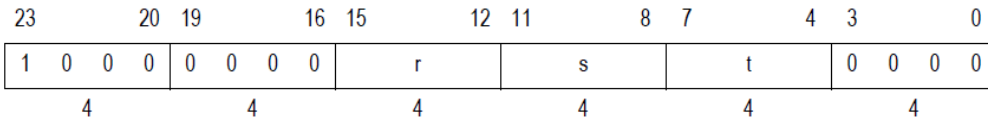
```
FR[r] ← absS(FR[s])
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.4 ADD—Add

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 45)

Assembler Syntax

```
ADD ar, as, at
```

Description

ADD calculates the two's complement 32-bit sum of address registers `as` and `at`. The low 32 bits of the sum are written to address register `ar`. Arithmetic overflow is not detected.

ADD is a 24-bit instruction. The `ADD.N` density-option instruction performs the same operation in a 16-bit encoding.

Assembler Note

The assembler may convert `ADD` instructions to `ADD.N` when the Code Density Option is enabled. Prefixing the `ADD` instruction with an underscore (`_ADD`) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

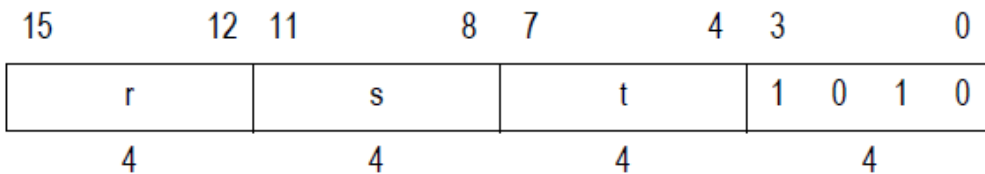
```
AR[r] ← AR[s] + AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.5 ADD.N—Narrow Add

Instruction Word (RRRN)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

```
ADD.N ar, as, at
```

Description

This performs the same operation as the `ADD` instruction in a 16-bit encoding.

`ADD.N` calculates the two's complement 32-bit sum of address registers `as` and `at`. The low 32 bits of the sum are written to address register `ar`. Arithmetic overflow is not detected.

Assembler Note

The assembler may convert `ADD.N` instructions to `ADD`. Prefixing the `ADD.N` instruction with an underscore (`_ADD.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

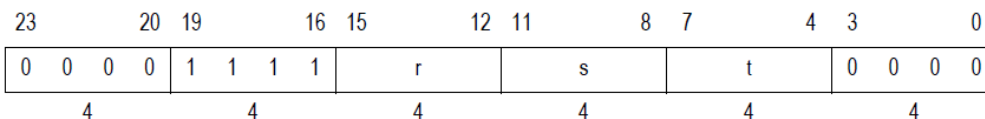
```
AR[r] ← AR[s] + AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.6 ADD.D—Add Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ADD.D fr, fs, ft
```

Description

ADD.D computes the IEEE754 double-precision sum of the contents of floating-point registers *fs* and *ft*, and writes the result to floating-point register *fr*.

Operation

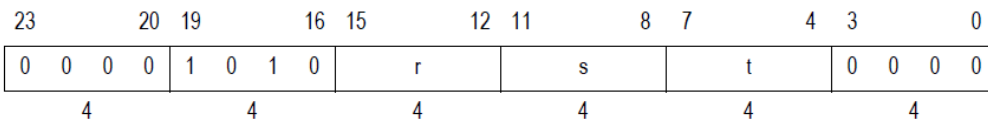
```
FR[r] ← FR[s] +D FR[t]  
FSR[StatusFlags: VOI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.7 ADD.S—Add Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ADD.S fr, fs, ft
```

Description

ADD.S computes the IEEE754 single-precision sum of the contents of floating-point registers *fs* and *ft*, and writes the result to floating-point register *fr*.

Operation

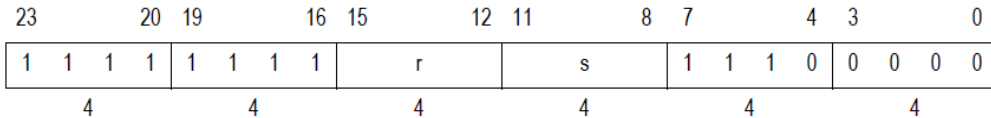
```
FR[r] ← FR[s] +S FR[t]  
FSR[StatusFlags: VOI] ← Or in update
```


Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.8 ADDEXP.D—Add Exponent Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ADDEXP.D fr, fs
```

Description

ADDEXP.D adds the unbiased exponent of the double-precision number in floating-point register fs to the exponent of the double-precision number in floating-point register fr . It also XORs the sign of the double-precision number in floating-point register fs with the sign of the double-precision number in floating-point register fr . It places these two results back into floating-point register fr . The mantissa of floating-point register fr is unchanged.

ADDEXP.D is used in divide and square root algorithms (see [Divide and Square Root Sequences](#) on page 110) and is not intended for use anywhere else.

Operation

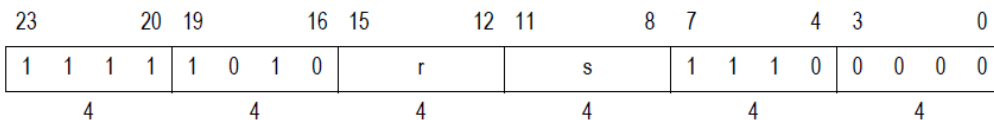
```
FR[r]63 ← FR[r]63 xor FR[s]63  
FR[r]62..52 ← FR[r]62..52 + FR[s]62..52 - 1023  
FR[r]51..0 ← FR[r]51..0
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.9 ADDEXP.S—Add Exponent Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ADDEXP.S fr, fs
```

Description

ADDEXP.S adds the unbiased exponent of the single-precision number in floating-point register `fs` to the exponent of the single-precision number in floating-point register `fr`. It also XORs the sign of the single-precision number in floating-point register `fs` with the sign of the single-precision number in floating-point register `fr`. It places these two results back into floating-point register `fr`. The mantissa of floating-point register `fr` is unchanged.

ADDEXP.S is used in divide and square root algorithms (see [Divide and Square Root Sequences](#) on page 110) and is not intended for use anywhere else.

Operation

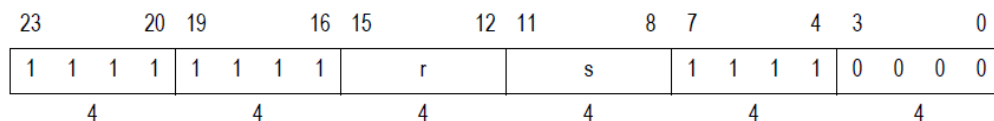
```
FR[r]31 ← FR[r]31 xor FR[s]31
FR[r]30..23 ← FR[r]30..23 + FR[s]30..23 - 127
FR[r]22..0 ← FR[r]22..0
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.10 ADDEXPM.D—Add Exponent from Mantissa Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ADDEXPM.D fr, fs
```

Description

ADDEXPM.D adds bits of the mantissa of the double-precision number in floating-point register `fs` representing an unbiased exponent to the exponent of the double-precision number in floating-point register `fr`. It also XORs a bit of the mantissa of the double-precision number in floating-point register `fs` with the sign of the double-precision number in floating-point register `fr`. It places these two results back into floating-point register `fr`. The mantissa of floating-point register `fr` is unchanged. ADDEXPM.D is very similar to ADDEXP.D (see [Assembler Syntax](#)) except that bits of the `fs` mantissa are used in place of its sign and exponent.

ADDEXPM.D is used in divide and square root algorithms (see [Divide and Square Root Sequences](#) on page 110) and is not intended for use anywhere else.

Operation

```
FR[r]63 ← FR[r]63 xor FR[s]51  
FR[r]62..52 ← FR[r]62..52 + FR[s]50..40 - 1023  
FR[r]51..0 ← FR[r]51..0
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.11 ADDEXPM.S—Add Exponent from Mantissa Single

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
1	1	1	1	1	0	1	0	r	s	1	1	1	1	0	0	0	0
4				4				4				4					

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ADDEXPM.S fr, fs
```

Description

ADDEXPM.S adds bits of the mantissa of the single-precision number in floating-point register `fs` representing an unbiased exponent to the exponent of the single-precision number in floating-point register `fr`. It also XORs a bit of the mantissa of the single-precision number in floating-point register `fs` with the sign of the single-precision number in floating-point register `fr`. It places these two results back into floating-point register `fr`. The mantissa of floating-point register `fr` is unchanged. ADDEXPM.S is very similar to ADDEXP.S (see [Assembler Syntax](#)) except that bits of the `fs` mantissa are used in place of its sign and exponent.

ADDEXP.S is used in divide and square root algorithms (see [Divide and Square Root Sequences](#) on page 110) and is not intended for use anywhere else.

Operation

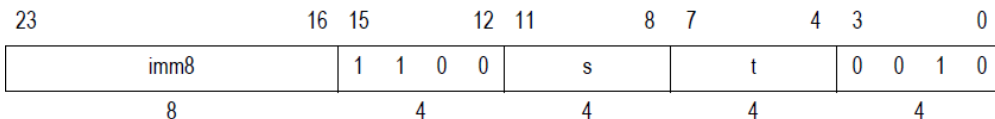
```
FR[r]31 ← FR[r]31 xor FR[s]22
FR[r]30..23 ← FR[r]30..23 + FR[s]21..14 - 127
FR[r]22..0 ← FR[r]22..0
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.12 ADDI—Add Immediate

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
ADDI at, as, -128..127
```

Description

ADDI calculates the two's complement 32-bit sum of address register `as` and a constant encoded in the `imm8` field. The low 32 bits of the sum are written to address register `at`. Arithmetic overflow is not detected.

The immediate operand encoded in the instruction can range from -128 to 127. It is decoded by sign-extending `imm8`.

`ADDI` is a 24-bit instruction. The `ADDI.N` density-option instruction performs a similar operation (the immediate operand has less range) in a 16-bit encoding.

Assembler Note

The assembler may convert `ADDI` instructions to `ADDI.N` when the Code Density Option is enabled and the immediate operand falls within the available range. If the immediate is too large the assembler may substitute an equivalent sequence. Prefixing the `ADDI` instruction with an underscore (`_ADDI`) disables these optimizations and forces the assembler to generate the wide form of the instruction or an error instead.

Operation

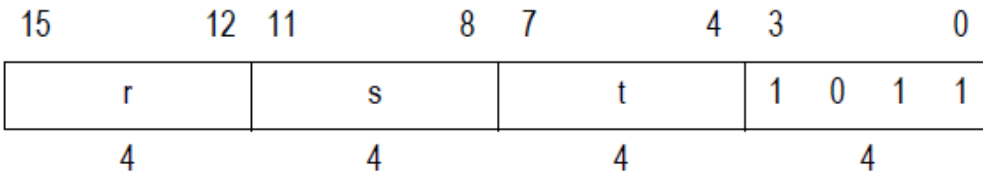
```
AR[t] ← AR[s] + (imm8724imm8)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.13 `ADDI.N`—Narrow Add Immediate

Instruction Word (RRRN)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

```
ADDI.N ar, as, imm
```

Description

`ADDI.N` is similar to `ADDI`, but has a 16-bit encoding and supports a smaller range of immediate operand values encoded in the instruction word.

`ADDI.N` calculates the two's complement 32-bit sum of address register `as` and an operand encoded in the `t` field. The low 32 bits of the sum are written to address register `ar`. Arithmetic overflow is not detected.

The operand encoded in the instruction can be -1 or one to 15. If t is zero, then a value of -1 is used, otherwise the value is the zero-extension of t .

Assembler Note

The assembler may convert `ADDI.N` instructions to `ADDI`. Prefixing the `ADDI.N` instruction with an underscore (`_ADDI.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction. In the assembler syntax, the number to be added to the register operand is specified. When the specified value is -1, the assembler encodes it as zero.

Operation

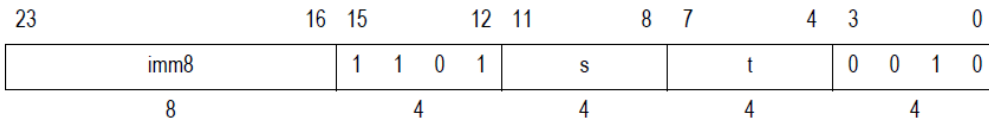
$$AR[r] \leftarrow AR[s] + (\text{if } t = 0^4 \text{ then } 1^{32} \text{ else } 0^{28}!t)$$

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.14 ADDMI—Add Immediate with Shift by 8

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
ADDMI at, as, -32768..32512
```

Description

`ADDMI` extends the range of constant addition. It is often used in conjunction with load and store instructions to extend the range of the base, plus offset the calculation.

`ADDMI` calculates the two's complement 32-bit sum of address register `as` and an operand encoded in the `imm8` field. The low 32 bits of the sum are written to address register `at`. Arithmetic overflow is not detected.

The operand encoded in the instruction can have values that are multiples of 256 ranging from -32768 to 32512. It is decoded by sign-extending `imm8` and shifting the result left by eight bits.

Assembler Note

In the assembler syntax, the value to be added to the register operand is specified. The assembler encodes this into the instruction by dividing by 256.

Operation

$$AR[t] \leftarrow AR[s] + (imm8_7|16|imm8_10^8)$$

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.15 ADDX2—Add with Shift by 1

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	0	1	0	0	0	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
ADDX2 ar, as, at
```

Description

ADDX2 calculates the two's complement 32-bit sum of address register `as` shifted left by one bit and address register `at`. The low 32 bits of the sum are written to address register `ar`. Arithmetic overflow is not detected.

ADDX2 is frequently used for address calculation and as part of sequences to multiply by small constants.

Operation

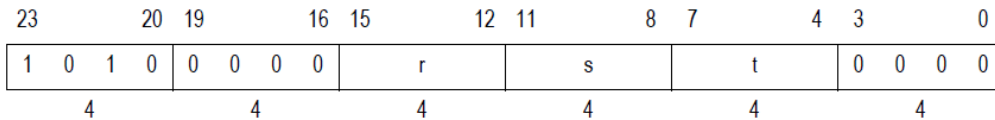
$$AR[r] \leftarrow (AR[s]_{30..0} \ll 1) + AR[t]$$

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.16 ADDX4—Add with Shift by 2

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
ADDX4 ar, as, at
```

Description

ADDX4 calculates the two's complement 32-bit sum of address register `as` shifted left by two bits and address register `at`. The low 32 bits of the sum are written to address register `ar`. Arithmetic overflow is not detected.

ADDX4 is frequently used for address calculation and as part of sequences to multiply by small constants.

Operation

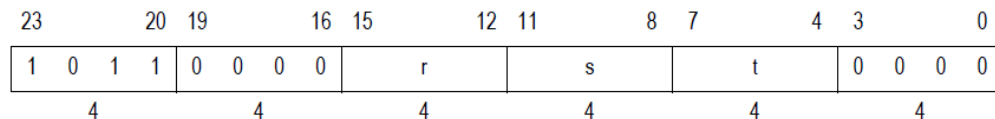
$$AR[r] \leftarrow (AR[s]_{29..0} \ll 2) + AR[t]$$

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.17 ADDX8—Add with Shift by 3

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
ADDX8 ar, as, at
```

Description

ADDX8 calculates the two's complement 32-bit sum of address register `as` shifted left by 3 bits and address register `at`. The low 32 bits of the sum are written to address register `ar`. Arithmetic overflow is not detected.

ADDX8 is frequently used for address calculation and as part of sequences to multiply by small constants.

Operation

$$AR[r] \leftarrow (AR[s]_{28..0} \ll 3) + AR[t]$$

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.18 ALL4—All 4 Booleans True

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	0	0	0	0	0	0	0	1	0	0	1	s	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
ALL4 bt, bs
```

Description

ALL4 sets Boolean register `bt` to the logical and of the four Boolean registers `bs+0`, `bs+1`, `bs+2`, and `bs+3`. `bs` must be a multiple of four (`b0`, `b4`, `b8`, or `b12`); otherwise the operation of this instruction is not defined. ALL4 reduces four test results such that the result is true if all four tests are true.

When the sense of the `bs` Booleans is inverted ($0 \rightarrow \text{true}$, $1 \rightarrow \text{false}$), use `ANY4` and an inverted test of the result.

Operation

```
BRt ← BRs+3 and BRs+2 and BRs+1 and BRs+0
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.19 ALL8—All 8 Booleans True

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0
0	0	0	0	0	0	0	1	0	1	1	s
4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
ALL8 bt, bs
```

Description

`ALL8` sets Boolean register `bt` to the logical and of the eight Boolean registers `bs+0`, `bs+1`, ..., `bs+6`, and `bs+7`. `bs` must be a multiple of eight (`b0` or `b8`); otherwise the operation of this instruction is not defined. `ALL8` reduces eight test results such that the result is true if all eight tests are true.

When the sense of the `bs` Booleans is inverted ($0 \rightarrow \text{true}$, $1 \rightarrow \text{false}$), use `ANY8` and an inverted test of the result.

Operation

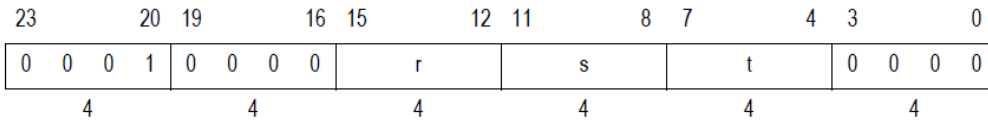
```
BRt ← BRs+7 and ... and BRs+0
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.20 AND—Bitwise Logical And

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
AND ar, as, at
```

Description

AND calculates the bitwise logical and of address registers `as` and `at`. The result is written to address register `ar`.

Operation

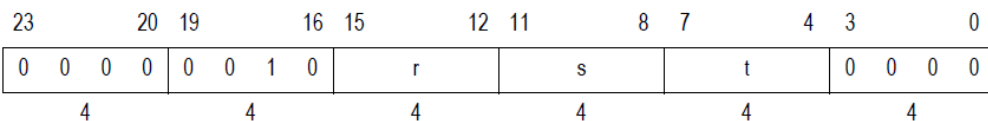
```
AR[r] ← AR[s] and AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.21 ANDB—Boolean And

Instruction Word (RRR)



Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
ANDB br, bs, bt
```

Description

ANDB performs the logical and of Boolean registers `bs` and `bt` and writes the result to Boolean register `br`.

When the sense of one of the source Booleans is inverted ($0 \rightarrow \text{true}$, $1 \rightarrow \text{false}$), use `ANDBC`. When the sense of both of the source Booleans is inverted, use `ORB` and an inverted test of the result.

Operation

```
BRr ← BRs and BRt
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.22 ANDBC—Boolean And with Complement

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	0	0	1	0	0	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
ANDBC br, bs, bt
```

Description

ANDBC performs the logical and of Boolean register `bs` with the logical complement of Boolean register `bt`, and writes the result to Boolean register `br`.

Operation

```
BRr ← BRs and not BRt
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.23 ANY4—Any 4 Booleans True

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0
0	0	0	0	0	0	0	0	0	0	0	0
4				4				4		4	

Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
ANY4 bt, bs
```

Description

ANY4 sets Boolean register `bt` to the logical or of the four Boolean registers `bs+0`, `bs+1`, `bs+2`, and `bs+3`. `bs` must be a multiple of four (`b0`, `b4`, `b8`, or `b12`); otherwise the operation of this instruction is not defined. ANY4 reduces four test results such that the result is true if any of the four tests are true.

When the sense of the `bs` Booleans is inverted (`0` → true, `1` → false), use ALL4 and an inverted test of the result.

Operation

```
BRt ← BRs+3 or BRs+2 or BRs+1 or BRs+0
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.24 ANY8—Any 8 Booleans True

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0
0	0	0	0	0	0	0	0	0	0	0	0
4				4				4		4	

Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
ANY8 bt, bs
```

Description

ANY8 sets Boolean register `bt` to the logical or of the eight Boolean registers `bs+0`, `bs+1`, ..., `bs+6`, and `bs+7`. `bs` must be a multiple of eight (`b0` or `b8`); otherwise the operation of this instruction is not defined. ANY8 reduces eight test results such that the result is true if any of the eight tests are true.

When the sense of the `bs` Booleans is inverted (`0` → true, `1` → false), use ALL8 and an inverted test of the result.

Operation

```
BRt ← BRs+7 or ... or BRs+0
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.25 BALL—Branch if All Bits Set

Instruction Word (RR18)

23	16	15	12	11	8	7	4	3	0		
imm8		0	1	0	0	s	t	0	1	1	1
8		4		4	4	4		4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BALL as, at, label
```

Description

BALL branches if all the bits specified by the mask in address register `at` are set in address register `as`. The test is performed by taking the bitwise logical and of `at` and the complement of `as`, and testing if the result is zero.

The target instruction address of the branch is given by the address of the `BALL` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If any of the masked bits are clear, execution continues with the next sequential instruction.

The inverse of `BALL` is `BNALL`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BALL`) disables this feature and forces the assembler to generate an error in this case.

Operation

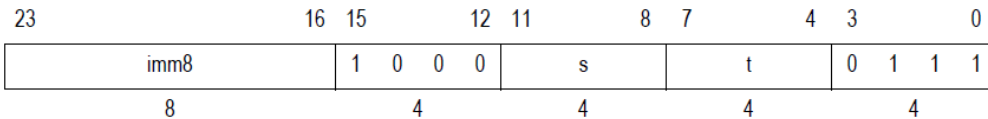
```
if ((not AR[s]) and AR[t]) = 032 then
    nextPC ← PC + (imm87:24imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.26 BANY—Branch if Any Bit Set

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BANY as, at, label
```

Description

`BANY` branches if any of the bits specified by the mask in address register `at` are set in address register `as`. The test is performed by taking the bitwise logical and of `as` and `at` and testing if the result is non-zero.

The target instruction address of the branch is given by the address of the `BANY` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If all of the masked bits are clear, execution continues with the next sequential instruction.

The inverse of `BANY` is `BNONE`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BANY`) disables this feature and forces the assembler to generate an error in this case.

Operation

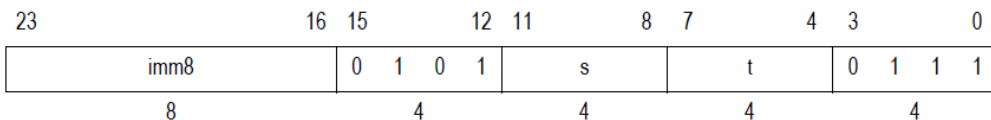
```
if (AR[s] and AR[t]) ≠ 032 then
    nextPC ← PC + (imm8724imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.27 BBC—Branch if Bit Clear

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BBC as, at, label
```

Description

`BBC` branches if the bit specified by the low five bits of address register `at` is clear in address register `as`. For little-endian processors, bit 0 is the least significant bit and bit 31 is the most significant bit. For big-endian processors, bit 0 is the most significant bit and bit 31 is the least significant bit.

The target instruction address of the branch is given by the address of the `BBC` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the specified bit is set, execution continues with the next sequential instruction.

The inverse of `BBC` is `BBS`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BBCI`) disables this feature and forces the assembler to generate an error in this case.

Operation

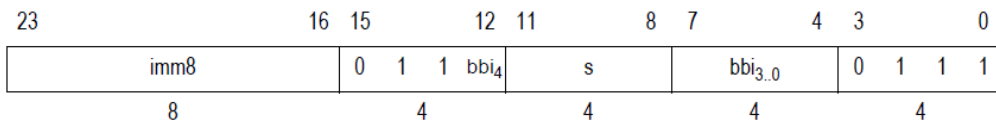
```
b ← bbi xor msbFirst5
if AR[s]b = 0 then
    nextPC ← PC + (imm824||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.29 BBCI.L—Branch if Bit Clear Immediate LE

Instruction Word (RRI8)



Required Configuration Option

Assembler Macro

Assembler Syntax

```
BBCI.L as, 0..31, label
```

Description

`BBCI.L` is an assembler macro for `BBCI` that always uses little-endian bit numbering. That is, it branches if the bit specified by its immediate is clear in address register `as`, where bit 0 is the least significant bit and bit 31 is the most significant bit.

The inverse of `BBCI.L` is `BBSI.L`.

Assembler Note

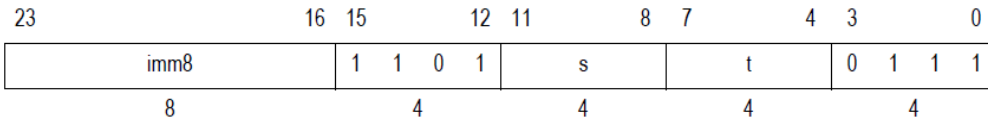
For little-endian processors, `BBCI.L` and `BBCI` are identical. For big-endian processors, the assembler will convert `BBCI.L` instructions to `BBCI` by changing the encoded immediate value to `31-imm`.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.30 BBS—Branch if Bit Set

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BBS as, at, label
```

Description

BBS branches if the bit specified by the low five bits of address register `at` is set in address register `as`. For little-endian processors, bit 0 is the least significant bit and bit 31 is the most significant bit. For big-endian processors, bit 0 is the most significant bit and bit 31 is the least significant bit.

The target instruction address of the branch is given by the address of the BBS instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the specified bit is clear, execution continues with the next sequential instruction.

The inverse of BBS is BBC.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BBS`) disables this feature and forces the assembler to generate an error in this case.

Operation

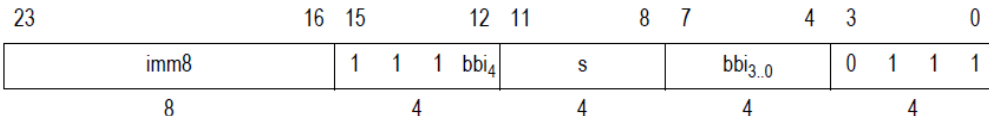
```
b ← AR[t]4..0 xor msbFirst5
if AR[s]b ≠ 0 then
    nextPC ← PC + (imm87:24 | imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.31 BBSI—Branch if Bit Set Immediate

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BBSI as, 0..31, label
```

Description

BBSI branches if the bit specified by the constant encoded in the `bbi` field of the instruction word is set in address register `as`. For little-endian processors, bit 0 is the least significant bit and bit 31 is the most significant bit. For big-endian processors, bit 0 is the most significant bit and bit 31 is the least significant bit. The `bbi` field is split, with bits 3..0 in bits 7..4 of the instruction word, and bit 4 in bit 12 of the instruction word.

The target instruction address of the branch is given by the address of the BBSI instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the specified bit is clear, execution continues with the next sequential instruction.

The inverse of BBSI is BBCI.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BBSI`) disables this feature and forces the assembler to generate an error in this case.

Operation

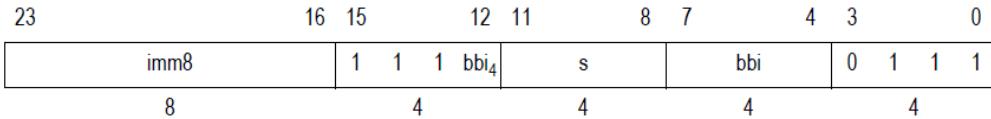
```
b ← bbi xor msbFirst5
if AR[s]b ≠ 0 then
    nextPC ← PC + (imm87:2 || imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.32 BBSI.L—Branch if Bit Set Immediate LE

Instruction Word (RRI8)



Required Configuration Option

Assembler Macro

Assembler Syntax

```
BBSI.L as, 0..31, label
```

Description

`BBSI.L` is an assembler macro for `BBSI` that always uses little-endian bit numbering. That is, it branches if the bit specified by its immediate is set in address register `as`, where bit 0 is the least significant bit and bit 31 is the most significant bit.

The inverse of `BBSI.L` is `BBCI.L`.

Assembler Note

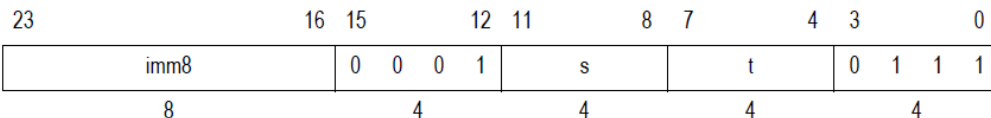
For little-endian processors, `BBSI.L` and `BBSI` are identical. For big-endian processors, the assembler will convert `BBSI.L` instructions to `BBSI` by changing the encoded immediate value to $31-imm$.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.33 BEQ—Branch if Equal

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BEQ as, at, label
```

Description

BEQ branches if address registers *as* and *at* are equal.

The target instruction address of the branch is given by the address of the BEQ instruction plus the sign-extended 8-bit *imm8* field of the instruction plus four. If the registers are not equal, execution continues with the next sequential instruction.

The inverse of BEQ is BNE.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BEQ`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if AR[s] = AR[t] then
    nextPC ← PC + (imm8,24|imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.34 BEQI—Branch if Equal Immediate

Instruction Word (BR18)

23	16	15	12	11	8	7	6	5	4	3	0		
imm8				r	s	0	0	1	0	0	1	1	0
8				4	4	2		2		4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BEQI as, imm, label
```

Description

The target instruction address of the branch is given by the address of the `BEQZ` instruction, plus the sign-extended 12-bit `imm12` field of the instruction plus four. If register `as` is not equal to zero, execution continues with the next sequential instruction.

The inverse of `BEQZ` is `BNEZ`.

Assembler Note

The assembler may convert `BEQZ` instructions to `BEQZ.N` when the Code Density Option is enabled and the branch target is reachable with the shorter instruction. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BEQZ`) disables these features and forces the assembler to generate the wide form of the instruction and an error when the label is out of range).

Operation

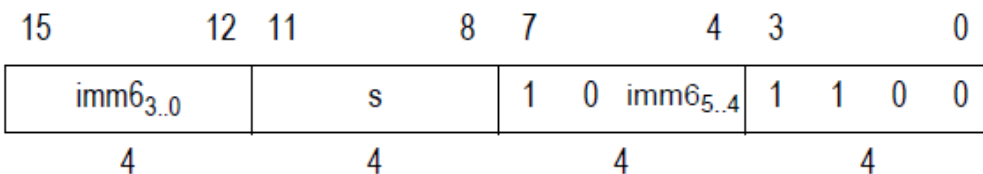
```
if AR[s] = 032 then
    nextPC ← PC + (imm121120imm12) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.36 BEQZ.N—Narrow Branch if Equal Zero

Instruction Word (RI6)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

```
BEQZ.N as, label
```

Description

This performs the same operation as the `BEQZ` instruction in a 16-bit encoding. `BEQZ.N` branches if address register `as` is equal to zero. `BEQZ.N` provides six bits of target range instead of the 12 bits available in `BEQZ`.

The target instruction address of the branch is given by the address of the `BEQZ.N` instruction, plus the zero-extended 6-bit `imm6` field of the instruction plus four. Because the offset is unsigned, this instruction can only be used to branch forward. If register `as` is not equal to zero, execution continues with the next sequential instruction.

The inverse of `BEQZ.N` is `BNEZ.N`.

Assembler Note

The assembler may convert `BEQZ.N` instructions to `BEQZ`. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BEQZ.N`) disables these features and forces the assembler to generate the narrow form of the instruction and an error when the label is out of range.

Operation

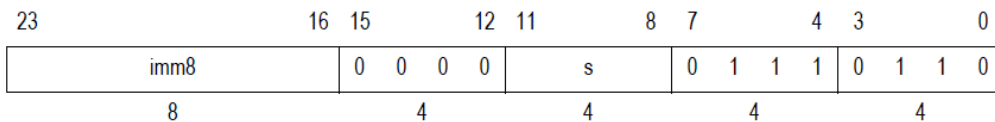
```
if AR[s] = 032 then
    nextPC ← PC + (026||imm6) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.37 BF—Branch if False

Instruction Word (RRI8)



Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
BF bs, label
```

Description

`BF` branches to the target address if Boolean register `bs` is false.

The target instruction address of the branch is given by the address of the `BF` instruction plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the Boolean register `bs` is true, execution continues with the next sequential instruction.

The inverse of `BF` is `BT`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BF`) disables this feature and forces the assembler to generate an error when the label is out of range.

Operation

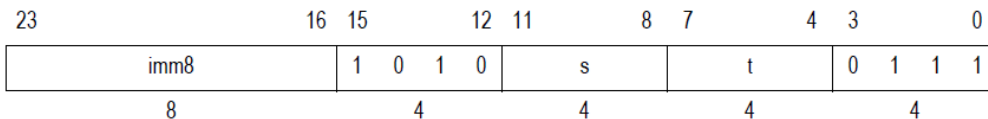
```
if not BRs then
    nextPC ← PC + (imm8,24||imm8) + 4
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.38 BGE—Branch if Greater Than or Equal

Instruction Word (RR18)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BGE as, at, label
```

Description

`BGE` branches if address register `as` is two's complement greater than or equal to address register `at`.

The target instruction address of the branch is given by the address of the `BGE` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the address register `as` is less than address register `at`, execution continues with the next sequential instruction.

The inverse of `BGE` is `BLT`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BGE`) disables this feature and forces the assembler to generate an error in this case.

Operation

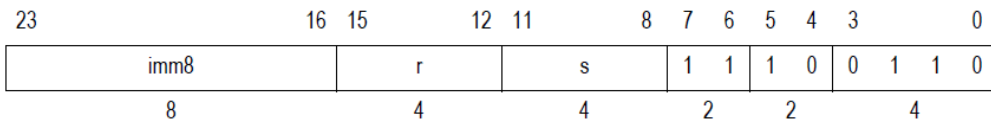
```
if AR[s] > AR[t] then
    nextPC ← PC + (imm8,24||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.39 BGEI—Branch if Greater Than or Equal Immediate

Instruction Word (BRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BGEI as, imm, label
```

Description

`BGEI` branches if address register `as` is two's complement greater than or equal to the constant encoded in the `r` field. The constant values encoded in the `r` field are not simply 0..15. For the constant values that can be encoded by `r`, see [Branch Immediate \(b4const\) Encodings](#).

The target instruction address of the branch is given by the address of the `BGEI` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the address register `as` is less than the constant, execution continues with the next sequential instruction.

The inverse of `BGEI` is `BLTI`.

Assembler Note

The assembler may convert `BGEI` instructions to `BGEZ` when given an immediate operand that evaluates to zero. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BGEI`) disables these features and forces the assembler to generate an error instead.

Operation

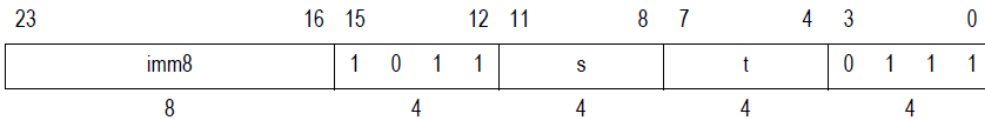
```
if AR[s] ≥ B4CONST(r) then
    nextPC ← PC + (imm8,24|imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.40 BGEU—Branch if Greater Than or Equal Unsigned

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BGEU as, at, label
```

Description

`BGEU` branches if address register `as` is unsigned greater than or equal to address register `at`.

The target instruction address of the branch is given by the address of the `BGEU` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the address register `as` is unsigned less than address register `at`, execution continues with the next sequential instruction.

The inverse of `BGEU` is `BLTU`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BGEU`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if (O1AR[s]) > (O1AR[t]) then
    nextPC ← PC + (imm8,24|imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.41 BGEUI—Branch if Greater Than or Eq Unsigned Imm

Instruction Word (BRI8)

23	16	15	12	11	8	7	6	5	4	3	0				
imm8				r		s		1	1	1	1	0	1	1	0
8				4		4		2		2		4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BGEUI as, imm, label
```

Description

BGEUI branches if address register `as` is unsigned greater than or equal to the constant encoded in the `r` field. The constant values encoded in the `r` field are not simply 0..15. For the constant values that can be encoded by `r`, see [Branch Unsigned Immediate \(b4constu\) Encodings](#).

The target instruction address of the branch is given by the address of the BGEUI instruction plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the address register `as` is less than the constant, execution continues with the next sequential instruction.

The inverse of BGEUI is BLTUI.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BGEUI`) disables this feature and forces the assembler to generate an error in this case.

Operation

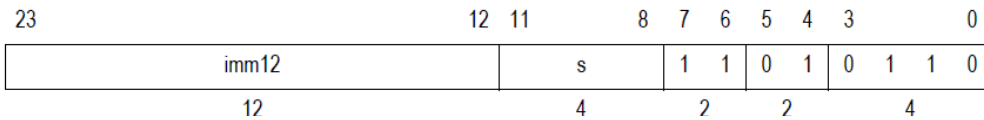
```
if (01AR[s]) ≥ (01B4CONSTU(r)) then
    nextPC ← PC + (imm8,24|imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.42 BGEZ—Branch if Greater Than or Equal to Zero

Instruction Word (BRI12)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BGEZ as, label
```

Description

BGEZ branches if address register `as` is greater than or equal to zero (the most significant bit is clear). BGEZ provides 12 bits of target range instead of the eight bits available in most conditional branches.

The target instruction address of the branch is given by the address of the BGEZ instruction plus the sign-extended 12-bit `imm12` field of the instruction plus four. If register `as` is less than zero, execution continues with the next sequential instruction.

The inverse of BGEZ is BLTZ.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BGEZ`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
if AR[s]31 = 0 then
```

```

nextPC ← PC + (imm121120imm12) + 4
endif

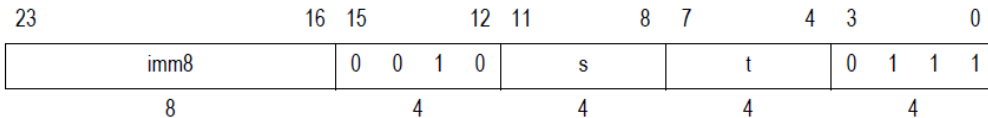
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.43 BLT—Branch if Less Than

Instruction Word (RR18)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```

BLT as, at, label

```

Description

BLT branches if address register *as* is two's complement less than address register *at*.

The target instruction address of the branch is given by the address of the BLT instruction plus the sign-extended 8-bit *imm8* field of the instruction plus four. If the address register *as* is greater than or equal to address register *at*, execution continues with the next sequential instruction.

The inverse of BLT is BGE.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BLT`) disables this feature and forces the assembler to generate an error in this case.

Operation

```

if AR[s] < AR[t] then
    nextPC ← PC + (imm8724imm8) + 4
endif

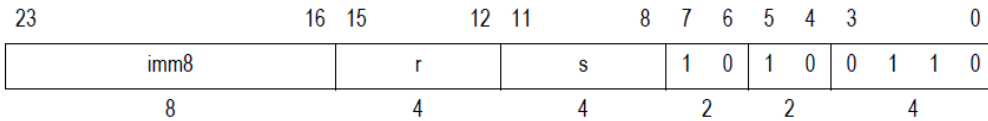
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.44 BLTI—Branch if Less Than Immediate

Instruction Word (BRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BLTI as, imm, label
```

Description

BLTI branches if address register `as` is two's complement less than the constant encoded in the `r` field. The constant values encoded in the `r` field are not simply 0..15. For the constant values that can be encoded by `r`, see [Branch Immediate \(*b4const*\) Encodings](#).

The target instruction address of the branch is given by the address of the BLTI instruction plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the address register `as` is greater than or equal to the constant, execution continues with the next sequential instruction.

The inverse of BLTI is BGEI.

Assembler Note

The assembler may convert BLTI instructions to BLTZ when given an immediate operand that evaluates to zero. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BLTI`) disables these features and forces the assembler to generate an error instead.

Operation

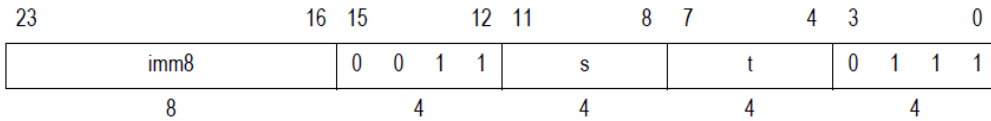
```
if AR[s] < B4CONST(r) then
    nextPC ← PC + (imm8,24||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.45 BLTU—Branch if Less Than Unsigned

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BLTU as, at, label
```

Description

BLTU branches if address register `as` is unsigned less than address register `at`.

The target instruction address of the branch is given by the address of the BLTU instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the address register `as` is greater than or equal to address register `at`, execution continues with the next sequential instruction.

The inverse of BLTU is BGEU.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BLTU`) disables this feature and forces the assembler to generate an error in this case.

Operation

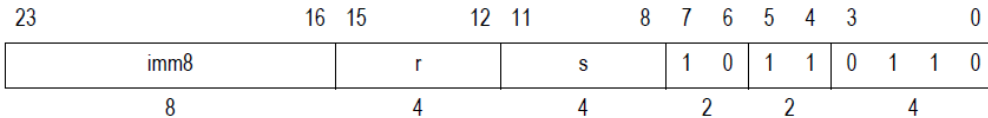
```
if (0!AR[s]) < (0!AR[t]) then
    nextPC ← PC + (imm8,24|imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.46 BLTUI—Branch if Less Than Unsigned Immediate

Instruction Word (BRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BLTUI as, imm, label
```

Description

BLTUI branches if address register `as` is unsigned less than the constant encoded in the `r` field. The constant values encoded in the `r` field are not simply 0..15. For the constant values that can be encoded by `r`, see [Branch Unsigned Immediate \(*b4constu*\) Encodings](#).

The target instruction address of the branch is given by the address of the BLTUI instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the address register `as` is greater than or equal to the constant, execution continues with the next sequential instruction.

The inverse of BLTUI is BGEUI.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BLTUI`) disables this feature and forces the assembler to generate an error in this case.

Operation

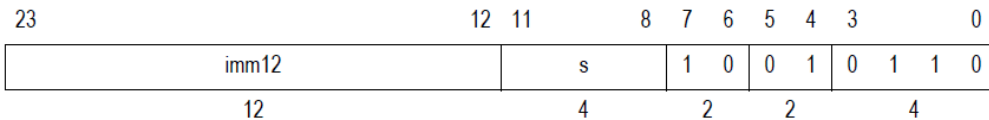
```
if (01AR[s]) < (01B4CONSTU(r)) then
    nextPC ← PC + (imm87:2||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.47 BLTZ—Branch if Less Than Zero

Instruction Word (BRI12)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BLTZ as, label
```

Description

`BLTZ` branches if address register `as` is less than zero (the most significant bit is set). `BLTZ` provides 12 bits of target range instead of the eight bits available in most conditional branches.

The target instruction address of the branch is given by the address of the `BLTZ` instruction, plus the sign-extended 12-bit `imm12` field of the instruction plus four. If register `as` is greater than or equal to zero, execution continues with the next sequential instruction.

The inverse of `BLTZ` is `BGEZ`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BLTZ`) disables this feature and forces the assembler to generate an error in this case.

Operation

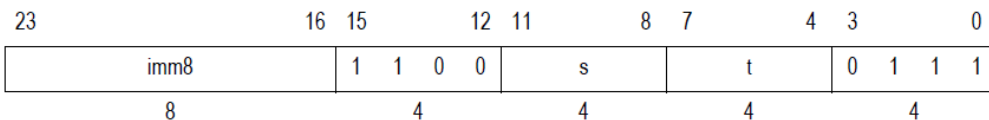
```
if AR[s]31 ≠ 0 then
    nextPC ← PC + (imm121120imm12) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.48 BNALL—Branch if Not-All Bits Set

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BNALL as, at, label
```

Description

`BNALL` branches if any of the bits specified by the mask in address register `at` are clear in address register `as` (that is, if they are not all set). The test is performed by taking the bitwise logical and of `at` with the complement of `as` and testing if the result is non-zero.

The target instruction address of the branch is given by the address of the `BNALL` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If all of the masked bits are set, execution continues with the next sequential instruction.

The inverse of `BNALL` is `BALL`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BNALL`) disables this feature and forces the assembler to generate an error in this case.

Operation

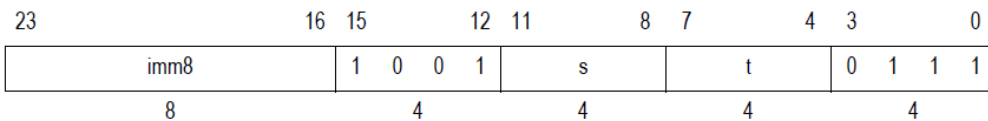
```
if ((not AR[s]) and AR[t]) ≠ 032 then
    nextPC ← PC + (imm87:24 | imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.49 BNE—Branch if Not Equal

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BNE as, at, label
```

Description

BNE branches if address registers *as* and *at* are not equal.

The target instruction address of the branch is given by the address of the **BNE** instruction, plus the sign-extended 8-bit *imm8* field of the instruction plus four. If the registers are equal, execution continues with the next sequential instruction.

The inverse of **BNE** is **BEQ**.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (**_BNE**) disables this feature and forces the assembler to generate an error in this case.

Operation

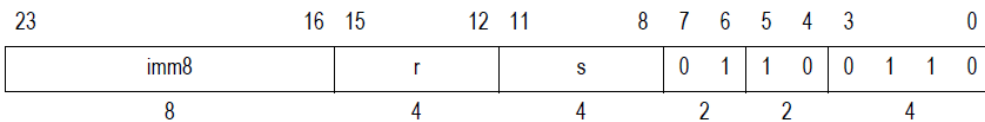
```
if AR[s] ≠ AR[t] then
    nextPC ← PC + (imm8,24||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.50 BNEI—Branch if Not Equal Immediate

Instruction Word (BRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BNEI as, imm, label
```

Description

`BNEI` branches if address register `as` and a constant encoded in the `r` field are not equal. The constant values encoded in the `r` field are not simply 0..15. For the constant values that can be encoded by `r`, see [Branch Immediate \(*b4const*\) Encodings](#).

The target instruction address of the branch is given by the address of the `BNEI` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the register is equal to the constant, execution continues with the next sequential instruction.

The inverse of `BNEI` is `BEQI`.

Assembler Note

The assembler may convert `BNEI` instructions to `BNEZ` or `BNEZ.N` when given an immediate operand that evaluates to zero. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BNEI`) disables these features and forces the assembler to generate an error instead.

Operation

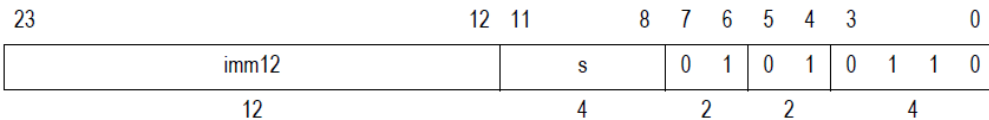
```
if AR[s] ≠ B4CONST(r) then
    nextPC ← PC + (imm8,24||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.51 BNEZ—Branch if Not-Equal to Zero

Instruction Word (BRI12)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BNEZ as, label
```

Description

`BNEZ` branches if address register `as` is not equal to zero. `BNEZ` provides 12 bits of target range instead of the eight bits available in most conditional branches.

The target instruction address of the branch is given by the address of the `BNEZ` instruction, plus the sign-extended 12-bit `imm12` field of the instruction plus four. If register `as` is equal to zero, execution continues with the next sequential instruction.

The inverse of `BNEZ` is `BEQZ`.

Assembler Note

The assembler may convert `BNEZ` instructions to `BNEZ.N` when the Code Density Option is enabled and the branch target is reachable with the shorter instruction. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BNEZ`) disables these features and forces the assembler to generate the `BNEZ` form of the instruction and an error when the label is out of range.

Operation

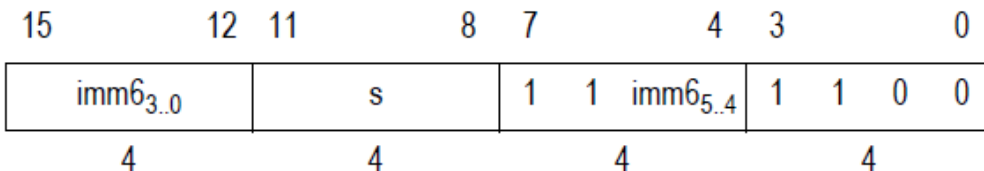
```
if AR[s] ≠ 032 then
    nextPC ← PC + (imm121120imm12) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.52 BNEZ.N—Narrow Branch if Not Equal Zero

Instruction Word (RI6)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

```
BNEZ.N as, label
```

Description

This performs the same operation as the `BNEZ` instruction in a 16-bit encoding. `BNEZ.N` branches if address register `as` is not equal to zero. `BNEZ.N` provides six bits of target range instead of the 12 bits available in `BNEZ`.

The target instruction address of the branch is given by the address of the `BNEZ.N` instruction, plus the zero-extended 6-bit `imm6` field of the instruction plus four. Because the offset is unsigned, this instruction can only be used to branch forward. If register `as` is equal to zero, execution continues with the next sequential instruction.

The inverse of `BNEZ.N` is `BEQZ.N`.

Assembler Note

The assembler may convert `BNEZ.N` instructions to `BNEZ`. The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BNEZ.N`) disables these features and forces the assembler to generate the narrow form of the instruction and an error when the label is out of range.

Operation

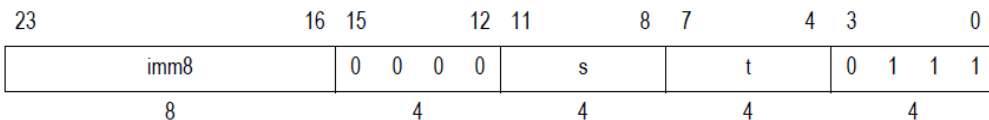
```
if AR[s] ≠ 032 then
    nextPC ← PC + (026imm6) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.53 BNONE—Branch if No Bit Set

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
BNONE as, at, label
```

Description

`BNONE` branches if all of the bits specified by the mask in address register `at` are clear in address register `as` (that is, if none of them are set). The test is performed by taking the bitwise logical and of `as` with `at` and testing if the result is zero.

The target instruction address of the branch is given by the address of the `BNONE` instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If any of the masked bits are set, execution continues with the next sequential instruction.

The inverse of `BNONE` is `BANY`.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BNONE`) disables this feature and forces the assembler to generate an error in this case.

Operation

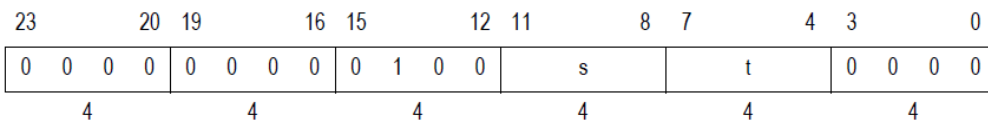
```
if (AR[s] and AR[t]) = 032 then
    nextPC ← PC + (imm8,24||imm8) + 4
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.54 BREAK—Breakpoint

Instruction Word (RRR)



Required Configuration Option

Debug Option (See [Debug Option](#) on page 256)

Assembler Syntax

```
BREAK 0..15, 0..15
```

Description

Under the Debug Option, this instruction simply raises an exception when it is executed and `PS.INTLEVEL < DEBUGLEVEL`. The high-priority vector for `DEBUGLEVEL` is used. The `DEBUGCAUSE` register is written as part of raising the exception to indicate that `BREAK` raised the debug exception. The address of the `BREAK` instruction is stored in `EPC[DEBUGLEVEL]`.

The `s` and `t` fields of the instruction word are not used by the processor; they are available for use by the software. When `PS.INTLEVEL ≥ DEBUGLEVEL`, `BREAK` is a no-op.

The `BREAK` instruction typically calls a debugger when program execution reaches a certain point (a “breakpoint”). The instruction at the breakpoint is replaced with the `BREAK` instruction. To continue execution after a breakpoint is reached, the debugger must re-write the `BREAK` to the original instruction, single-step by one instruction, and then put back the `BREAK` instruction again.

Writing instructions requires special consideration. See the `ISYNC` instruction for more information.

When it is not possible to write the instruction memory (for example, for ROM code), the `IBREAKA` feature provides breakpoint capabilities (see [Debug Option](#) on page 256).

Software can also use `BREAK` to indicate an error condition that requires the programmer’s attention. The `s` and `t` fields may encode information about the situation.

`BREAK` is a 24-bit instruction. The `BREAK.N` density-option instruction performs a similar operation in a 16-bit encoding.

Assembler Note

The assembler may convert `BREAK` instructions to `BREAK.N` when the Code Density Option is enabled and the second `imm` is zero. Prefixing the instruction mnemonic with an underscore (`_BREAK`) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

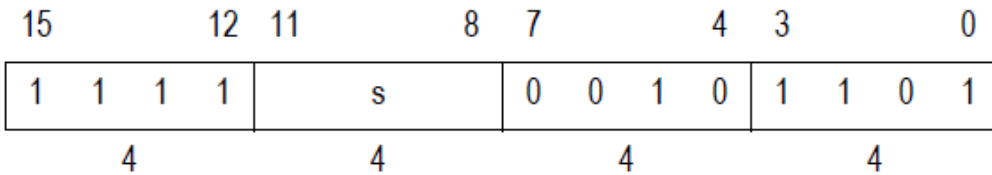
```
if PS.INTLEVEL < DEBUGLEVEL && Debug Option then
    EPC[DEBUGLEVEL] ← PC
    EPS[DEBUGLEVEL] ← PS
    DEBUGCAUSE ← 001000
    nextPC ← InterruptVector[DEBUGLEVEL]
    PS.EXCM ← 1
    PS.INTLEVEL ← DEBUGLEVEL
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- DebugExcep(BREAK) if Debug Option

8.3.55 `BREAK.N`—Narrow Breakpoint

Instruction Word (RRRN)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82) and either Debug Option (See [Debug Option](#) on page 256)

Assembler Syntax

```
BREAK.N 0..15
```

Description

`BREAK.N` is similar in operation to `BREAK` ([Assembler Syntax](#)), except that it is encoded in a 16-bit format instead of 24 bits, there is only a 4-bit `imm` field, and a different bit is set in `DEBUGCAUSE`. Use this instruction to set breakpoints on 16-bit instructions.

Assembler Note

The assembler may convert `BREAK.N` instructions to `BREAK`. Prefixing the `BREAK.N` instruction with an underscore (`_BREAK.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

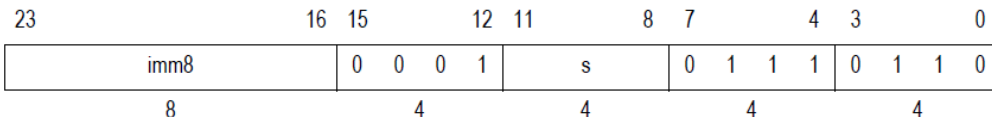
```
if PS.INTLEVEL < DEBUGLEVEL && Debug Option then
    EPC[DEBUGLEVEL] ← PC
    EPS[DEBUGLEVEL] ← PS
    DEBUGCAUSE ← 010000
    nextPC ← InterruptVector[DEBUGLEVEL]
    PS.EXCM ← 1
    PS.INTLEVEL ← DEBUGLEVEL
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- DebugExcep(BREAK.N) if Debug Option

8.3.56 BT—Branch if True

Instruction Word (RRI8)



Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)s

Assembler Syntax

```
BT bs, label
```

Description

BT branches to the target address if Boolean register `bs` is true.

The target instruction address of the branch is given by the address of the BT instruction, plus the sign-extended 8-bit `imm8` field of the instruction plus four. If the Boolean register `bs` is false, execution continues with the next sequential instruction.

The inverse of BT is BF.

Assembler Note

The assembler will substitute an equivalent sequence of instructions when the label is out of range. Prefixing the instruction mnemonic with an underscore (`_BT`) disables this feature and forces the assembler to generate an error when the label is out of range.

Operation

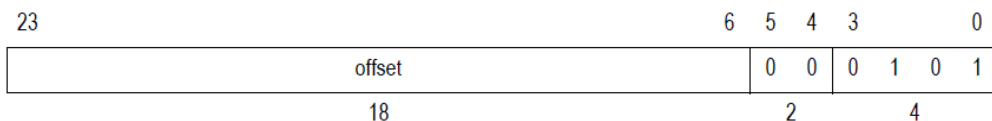
```
if BRs then
    nextPC ← PC + (imm8,24||imm8) + 4
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.57 CALL0—Non-windowed Call

Instruction Word (CALL)



Required Configuration Option

Assembler Syntax

```
CALL4 label
```

Description

`CALL4` calls subroutines using the register windows mechanism, requesting the callee rotate the window by four registers. The `CALL4` instruction does not rotate the window itself, but instead stores the window increment for later use by the `ENTRY` instruction. The return address and window increment are placed in the caller's `a4` (the callee's `a0`), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the `CALL4` instruction plus three). The window increment is also stored in the `CALLINC` field of the `PS` register, where it is accessed by the `ENTRY` instruction.

The target instruction address must be a 32-bit aligned `ENTRY` instruction. This allows `CALL4` to have a larger effective range (-524284 to 524288 bytes). The target instruction address of the call is given by the address of the `CALL4` instruction with the two least significant bits set to zero plus the sign-extended 18-bit `offset` field of the instruction shifted by two, plus four.

See the `CALLX4` instruction for calling routines where the target address is given by the contents of a register.

Use the `RETW` and `RETW.N` instructions to return from a subroutine called by `CALL4`.

The window increment stored with the return address register in `a4` occupies the two most significant bits of the register, and therefore those bits must be filled in by the sub-routine return. The `RETW` and `RETW.N` instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the `CALL0` instruction for calling routines using the non-windowed subroutine protocol.

The caller's `a4..a15` are the same registers as the callee's `a0..a11` after the callee executes the `ENTRY` instruction. You can use these registers for parameter passing. The caller's `a0..a3` are hidden by `CALL4`, and therefore you can use them to keep values that are live across the call.

Operation

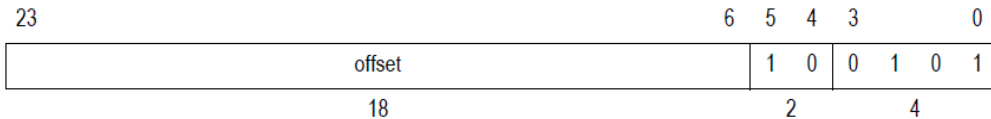
```
WindowCheck (00, 00, 01)
PS.CALLINC ← 01
AR[0100] ← 01|(nextPC3)29..0
nextPC ← (PC31..2 + (offset1712|offset) + 1) #00
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.59 CALL8—Call PC-relative, Rotate Window by 8

Instruction Word (CALL)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
CALL8 label
```

Description

Under the Windowed Register Option, `CALL8` calls subroutines using the register windows mechanism, requesting the callee rotate the window by eight registers. The `CALL8` instruction does not rotate the window itself, but instead stores the window increment for later use by the `ENTRY` instruction. The return address and window increment are placed in the caller's `a8` (the callee's `a0`), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the `CALL8` instruction plus three). The window increment is also stored in the `CALLINC` field of the `PS` register, where it is accessed by the `ENTRY` instruction.

The target instruction address must be a 32-bit aligned `ENTRY` instruction. This allows `CALL8` to have a larger effective range (–524284 to 524288 bytes). The target instruction address of the call is given by the address of the `CALL8` instruction with the two least significant bits set to zero, plus the sign-extended 18-bit `offset` field of the instruction shifted by two, plus four.

See the `CALLX8` instruction for calling routines where the target address is given by the contents of a register.

Use the `RETW` and `RETW.N` instructions to return from a subroutine called by `CALL8`.

Under the Windowed Register Option, the window increment stored with the return address register in `a8` occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The `RETW` and `RETW.N` instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the `CALL0` instruction for calling routines using the non-windowed subroutine protocol.

The caller's `a8..a15` are the same registers as the callee's `a0..a7` after the callee executes the `ENTRY` instruction. You can use these registers for parameter passing. The caller's `a0..a7` are

hidden by `CALL8`, and therefore you may use them to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 10)
PS.CALLINC ← 10 if Windowed Register Option
AR[1000] ← 10|(nextPC)29..0 if Windowed Register Option

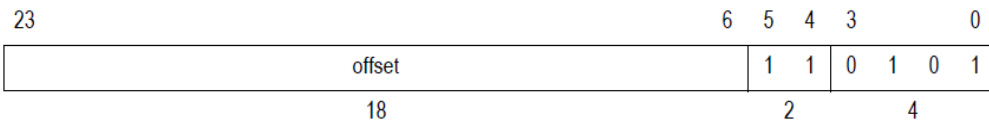
nextPC ← (PC31..2 + (offset1712|offset) + 1)100
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.60 CALL12—Call PC-relative, Rotate Window by 12

Instruction Word (CALL)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
CALL12 label
```

Description

`CALL12` calls subroutines using the register windows mechanism, requesting the callee rotate the window by 12 registers. The `CALL12` instruction does not rotate the window itself, but instead stores the window increment for later use by the `ENTRY` instruction. The return address and window increment are placed in the caller's `a12` (the callee's `a0`), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the `CALL12` instruction plus three). The window increment is also stored in the `CALLINC` field of the `PS` register, where it is accessed by the `ENTRY` instruction.

The target instruction address must be a 32-bit aligned `ENTRY` instruction. This allows `CALL12` to have a larger effective range (-524284 to 524288 bytes). The target instruction address of the call is given by the address of the `CALL12` instruction with the two least significant bits set to zero, plus the sign-extended 18-bit `offset` field of the instruction shifted by two, plus four.

See the `CALLX12` instruction for calling routines where the target address is given by the contents of a register.

The `RETW` and `RETW.N` instructions return from a subroutine called by `CALL12`.

The window increment stored with the return address register in `a12` occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The `RETW` and `RETW.N` instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the `CALL0` instruction for calling routines using the non-windowed subroutine protocol.

The caller's `a12..a15` are the same registers as the callee's `a0..a3` after the callee executes the `ENTRY` instruction. You can use these registers for parameter passing. The caller's `a0..a11` are hidden by `CALL12`, and therefore you may use them to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 11)
PS.CALLINC ← 11
AR[1100] ← 11|(nextPC)29..0
nextPC ← (PC31..2 + (offset1712|offset) + 1) #00
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.61 CALLX0—Non-windowed Call Register

Instruction Word (CALLX)

23	20	19	16	15	12	11	8	7	6	5	4	3	0								
0	0	0	0	0	0	0	0	0	0	0	0	s	1	1	0	0	0	0	0	0	
4				4				4				4		2		2		4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
CALLX0 as
```

Description

CALLX0 calls subroutines without using register windows. The return address is placed in a0, and the processor then branches to the target address. The return address is the address of the CALLX0 instruction, plus three.

The target instruction address of the call is given by the contents of address register as.

The RET and RET.N instructions return from a subroutine called by CALLX0.

To call using the register window mechanism, see the CALLX4, CALLX8, and CALLX12 instructions.

Operation

```
tmp ← nextPC
nextPC ← AR[s]
AR[0] ← tmp
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.62 CALLX4—Call Register, Rotate Window by 4

Instruction Word (CALLX)

23	20	19	16	15	12	11	8	7	6	5	4	3	0							
0	0	0	0	0	0	0	0	0	0	0	0	s	1	1	0	1	0	0	0	0
4				4				4				2		2		4				

Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
CALLX4 as
```

Description

CALLX4 calls subroutines using the register windows mechanism, requesting the callee rotate the window by four registers. The CALLX4 instruction does not rotate the window itself, but instead stores the window increment for later use by the ENTRY instruction. The return address and window increment are placed in the caller's a4 (the callee's a0), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the CALLX4 instruction plus three). The window increment is also stored in the CALLINC field of the PS register, where it is accessed by the ENTRY instruction.

The target instruction address of the call is given by the contents of address register `as`. The target instruction must be an `ENTRY` instruction.

See the `CALL4` instruction for calling routines where the target address is given by a PC-relative offset in the instruction.

The `RETW` and `RETW.N` instructions return from a subroutine called by `CALLX4`.

The window increment stored with the return address register in `a4` occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The `RETW` and `RETW.N` instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the `CALLX0` instruction for calling routines using the non-windowed subroutine protocol.

The caller's `a4..a15` are the same registers as the callee's `a0..a11` after the callee executes the `ENTRY` instruction. You can use these registers for parameter passing. The caller's `a0..a3` are hidden by `CALLX4`, and therefore you may use them to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 01)
PS.CALLINC ← 01
tmp ← nextPC
nextPC ← AR[s]
AR[01100] ← 011(tmp)29..0
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.63 CALLX8—Call Register, Rotate Window by 8

Instruction Word (CALLX)

23	20	19	16	15	12	11	8	7	6	5	4	3	0							
0	0	0	0	0	0	0	0	0	0	0	0	s	1	1	1	0	0	0	0	0
4				4				4				2		2		4				

Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240))

Assembler Syntax

```
CALLX8 as
```

Description

Under the Windowed Register Option, `CALLX8` calls subroutines using the register windows mechanism, requesting the callee rotate the window by eight registers. The `CALLX8` instruction does not rotate the window itself, but instead stores the window increment for later use by the `ENTRY` instruction. The return address and window increment are placed in the caller's `a8` (the callee's `a0`), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the `CALLX8` instruction plus three). The window increment is also stored in the `CALLINC` field of the `PS` register, where it is accessed by the `ENTRY` instruction.

The target instruction address of the call is given by the contents of address register `as`. The target instruction must be an `ENTRY` instruction.

See the `CALL8` instruction for calling routines where the target address is given by a PC-relative offset in the instruction.

The `RETW` and `RETW.N` [Assembler Syntax](#) instructions return from a subroutine called by `CALLX8`.

Under the Windowed Register Option, The window increment stored with the return address register in `a8` occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The `RETW` and `RETW.N` instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the `CALLX0` instruction for calling routines using the non-windowed subroutine protocol.

The caller's `a8..a15` are the same registers as the callee's `a0..a7` after the callee executes the `ENTRY` instruction. You can use these registers for parameter passing. The caller's `a0..a7` are hidden by `CALLX8`, and therefore you may use them to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 10)
PS.CALLINC ← 10 if Windowed Register Option
AR[1000] ← 10|(nextPC)29..0 if Windowed Register Option

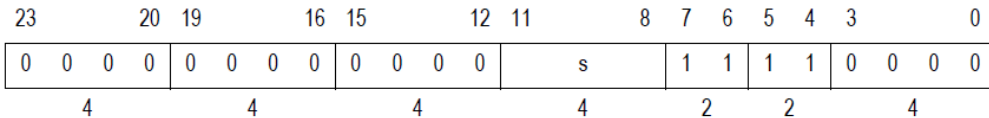
nextPC ← AR[s]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.64 CALLX12—Call Register, Rotate Window by 12

Instruction Word (CALLX)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
CALLX12 as
```

Description

`CALLX12` calls subroutines using the register windows mechanism, requesting the callee rotate the window by 12 registers. The `CALLX12` instruction does not rotate the window itself, but instead stores the window increment for later use by the `ENTRY` instruction. The return address and window increment are placed in the caller's `a12` (the callee's `a0`), and the processor then branches to the target address. The return address is the address of the next instruction (the address of the `CALLX12` instruction plus three). The window increment is also stored in the `CALLINC` field of the `PS` register, where it is accessed by the `ENTRY` instruction.

The target instruction address of the call is given by the contents of address register `as`. The target instruction must be an `ENTRY` instruction.

See the `CALL12` instruction for calling routines where the target address is given by a PC-relative offset in the instruction.

The `RETW` and `RETW.N` instructions return from a subroutine called by `CALLX12`.

The window increment stored with the return address register in `a12` occupies the two most significant bits of the register, and therefore those bits must be filled in by the subroutine return. The `RETW` and `RETW.N` instructions fill in these bits from the two most significant bits of their own address. This prevents register-window calls from being used to call a routine in a different 1GB region of the address space.

See the `CALLX0` instruction for calling routines using the non-windowed subroutine protocol.

The caller's `a12..a15` are the same registers as the callee's `a0..a3` after the callee executes the `ENTRY` instruction. These registers may be used for parameter passing. The caller's `a0..a11` are hidden by `CALLX12`, and therefore may be used to keep values that are live across the call.

Operation

```
WindowCheck (00, 00, 11)
PS.CALLINC ← 11
```

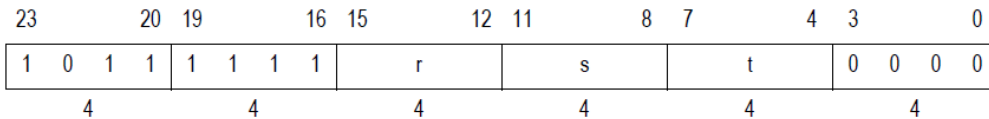
```
tmp ← nextPC
nextPC ← AR[s]
AR[11#00] ← 11#(tmp)29..0
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.65 CEIL.D—Ceiling Double to Fixed

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
CEIL.D ar, fs, 0..15
```

Description

CEIL.D converts the contents of floating-point register fs from double-precision to signed integer format, rounding toward $+\infty$. The double-precision value is first scaled by a power of two constant value encoded in the t field, with 0..15 representing 1.0, 2.0, 4.0, ..., 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for $t=0$ and moves to the left as t increases, until for $t=15$ there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $> 2^{31} - 1$), positive infinity, or NaN, $32'h7fffffff$ is returned; for negative overflow (scaled argument $\leq -2^{31} - 1$) or negative infinity, $32'h80000000$ is returned. The result is written to address register ar .

Operation

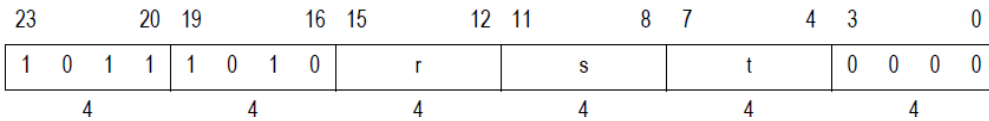
```
AR[r] ← ceilD(FR[s] ×D powD(2.0, t))
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.66 CEIL.S—Ceiling Single to Fixed

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
CEIL.S ar, fs, 0..15
```

Description

CEIL.S converts the contents of floating-point register *fs* from single-precision to signed integer format, rounding toward $+\infty$. The single-precision value is first scaled by a power of two constant value encoded in the *t* field, with 0..15 representing 1.0, 2.0, 4.0, ..., 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for *t*=0 and moves to the left as *t* increases, until for *t*=15 there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $> 2^{31} - 1$), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (scaled argument $\leq -2^{31} - 1$) or negative infinity, `32'h80000000` is returned. The result is written to address register *ar*.

Operation

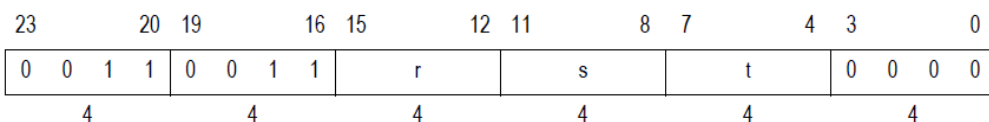
```
AR[r] ← ceils(FR[s] ×s pows(2.0,t))  
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.67 CLAMPS—Signed Clamp

Instruction Word (RRR)



Required Configuration Option

Miscellaneous Operations Option (See [Miscellaneous Operations Option](#) on page 94)

Assembler Syntax

```
CLAMPS ar, as, 7..22
```

Description

CLAMPS tests whether the contents of address register *as* fits as a signed value of *imm*+1 bits (in the range 7 to 22). If so, the value is written to address register *ar*; if not, the largest value of *imm*+1 bits with the same sign as *as* is written to *ar*. Thus CLAMPS performs the function

$$y \leftarrow \min(\max(x, -2^{imm}), 2^{imm-1})$$

CLAMPS may be used in conjunction with instructions such as ADD, SUB, MUL16S, and so forth to implement saturating arithmetic.

Assembler Note

The immediate values accepted by the assembler are 7 to 22. The assembler encodes these in the *t* field of the instruction using 0 to 15.

Operation

```
sign ← AR[s]31
AR[r] ← if AR[s]30..t+7 = sign24-t
        then AR[s]
        else sign25-t | (not sign)t+7
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.68 CLREX—Clear Exclusive

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0
0	0	0	0	0	0	0	0	1	0	0	0
4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

Exclusive Access Option (See [Exclusive Access Option](#) on page 123)

Assembler Syntax

```
CLREX
```

Description

CLREX clears the micro-architectural exclusive access mark set by L32EX. See [Exclusive Access Option](#) on page 123. This should be necessary only in certain operating system code, such as a full process context swap.

Operation

```
clrmonitor()
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.69 CONST.D—Constant Double

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0					
1	1	1	1	1	1	1	1	1	0	0	1	1	0	0	0	0
4				4				4				4				

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
CONST.D fr, 0..15
```

Description

CONST.D creates a double-precision constant and places it in floating-point register `fr`. The constant is chosen by the value of the `s` field as shown in the table below.:

s	Decimal Value	Hex Value
0x0	+0.0	0x0000_0000_0000_0000
0x1	+1.0	0x3FF0_0000_0000_0000

s	Decimal Value	Hex Value
0x2	+2.0	0x4000_0000_0000_0000
0x3	+0.5	0x3FE0_0000_0000_0000
0x4-F	<i>Reserved</i>	<i>Reserved</i>

Operation

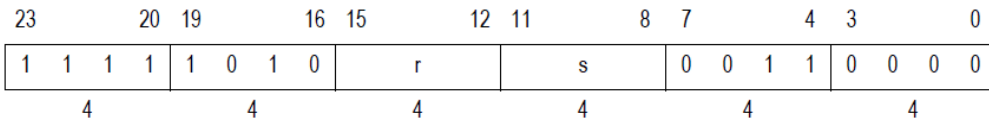
```
FR[r] ← table[s]
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.70 CONST.S—Constant Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
CONST.S fr, 0..15
```

Description

CONST.S creates a double-precision constant and places it in floating-point register `fr`. The constant is chosen by the value of the `s` field as shown in the table below.:

s	Decimal Value	Hex Value
0x0	+0.0	0x0000_0000
0x1	+1.0	0x3F80_0000

s	Decimal Value	Hex Value
0x2	+2.0	0x4000_0000
0x3	+0.5	0x3F00_0000
0x4-F	<i>Reserved</i>	<i>Reserved</i>

Operation

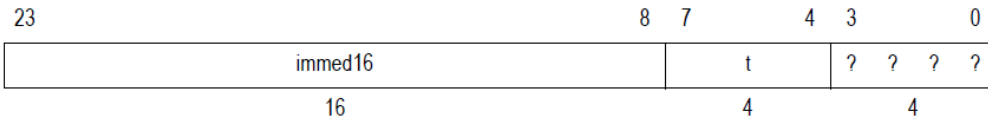
```
FR[r] ← table[s]
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.71 CONST16—Shift In 16-bit Constant

Instruction Word (RI16)



Required Configuration Option

No Option, bits[3-0] vary with configuration

Assembler Syntax

```
CONST16 at, 0..65535
```

Description

A pair of `CONST16` instructions form a load of a 32-bit constant from the instruction stream into an address register. It is typically used to load constant values into a register when the constant cannot be encoded in a `MOVI` instruction.

`CONST16` does a logical shift left by 16 of address register `at` and then inserts a 16-bit immediate in the 16 bits. The low 32 bits of the result are written to address register `at`.

If `CONST16` operates twice on the same address register, it replaces the original contents of the address register with the concatenation of the 16-bit immediates of the two instructions. The pair, then, inserts a 32-bit immediate into an address register.

The `CONST16` instruction requires a large amount of encoding space and is not used in most configurations. It is, therefore, not allocated a permanent encoding. Documentation for the particular configuration gives the encoding. This instruction is a leading candidate for a future variable encoding mechanism.

Operation

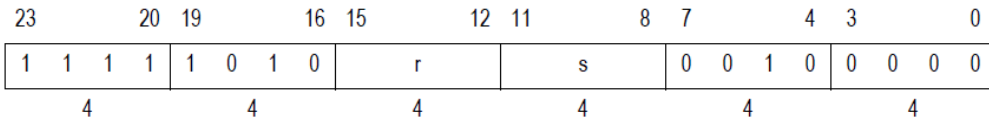
```
AR[t] ← AR[t]15..0 | immed16
```

Exceptions

- EveryInstR Group (see [EveryInst Group](#))

8.3.72 CVTD.S—Convert Single to Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
CVTD.S fr, fs
```

Description

`CVTD.S` reads the contents of floating-point register `fs`, interpreted as a single-precision floating-point number. It converts the value to a double-precision floating-point value and writes the result to floating-point register `fr`.

Operation

```
FR[r] ← ConvertToDouble(FR[s])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.73 CVTS.D—Convert Double to Single

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	1	1	1	1	1	1	0	0	1	0	0	0	0	0
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
CVTS.D fr, fs
```

Description

CVTS.D reads the contents of floating-point register `fs`, interpreted as a double-precision floating-point number. It converts the value to a single-precision floating-point value, with rounding according to the rounding control in the FCR register. The result is written to floating-point register `fr`.

Operation

```
FR[r] ← ConvertToSingle(FR[s])  
FSR[StatusFlags: VOUI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.74 DCI—Data Cache Coherent Hit Invalidate

Instruction Word (RRI4)

23	20	19	16	15	12	11	8	7	4	3	0							
imm4	0	0	0	1	0	1	1	1	s	1	0	0	0	0	0	0	1	0
4				4				4				4						

Required Configuration Option

Data Cache Coherent Invalidation Option (See [Data Cache Coherent Invalidation Option](#))

Assembler Syntax

```
DCI as, 0..240
```

Description

DCI invalidates the specified line in the level-1 data cache, if it is present. If the specified address is not in the data cache, then this instruction has no local effect. If the specified address is present, it is invalidated even if it contains dirty data. If the specified line has been locked by a `DPFL` instruction, then no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by a `DHU` or `DIU` instruction before it can be invalidated.

If hardware coherence is supported, a transaction is sent on the bus which requests other hardware coherent cores also to invalidate any copies of the line they might have, regardless of whether they are clean or dirty.

DCI forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135) as if the instruction were storing to the virtual address.

Because the organization of caches is implementation-specific, the operation below specifies only a call to the implementation's `dhitinval` function.

Whether or not DCI is a privileged instruction is implementation dependent.

Assembler Note

To form a virtual address DCI calculates the sum of address register `as` and the `imm4` field of the instruction word times sixteen. Therefore, the machine-code offset is in terms of 16-byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by sixteen.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (024∥imm4∥04)
    (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
```

```

    Exception (cause)
else
    dhitinval(vAddr, pAddr)
    send bus transaction to other coherent cores
endif
endif
endif

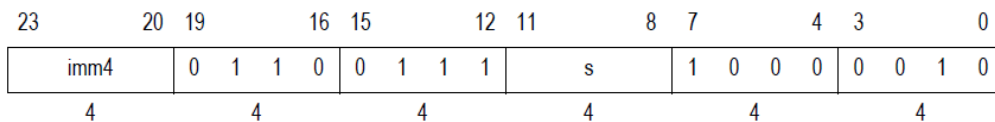
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Memory Group (see [Memory Group](#))
- GenExcep(StoreProhibitedCause) if Region Protection Option or MMU Option
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.75 DCWB—Data Cache Coherent Hit Writeback

Instruction Word (RRI4)



Required Configuration Option

Data Cache Coherent Invalidation Option (See [Data Cache Coherent Invalidation Option](#))

Assembler Syntax

```
DCWB as, 0..240
```

Description

This instruction forces dirty data in the data cache to be written back to memory. If the specified address is not in the data cache or is present but unmodified, then this instruction has no local effect. If the specified address is present and modified in the data cache, the line containing it is written back, and marked unmodified.

If hardware coherence is supported, a transaction is sent on the bus which requests other hardware coherent cores also to writeback any dirty copies of the line they might have.

DCWB forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see [The Exception Cause Register](#))

([EXCCAUSE](#)) under the [Exception Option 2](#) on page 135) as if it were loading from the virtual address.

Because the organization of caches is implementation-specific, the operation below specifies only a call to the implementation's `dhitwriteback` function.

Assembler Note

To form a virtual address `DCWB` calculates the sum of address register `as` and the `imm4` field of the instruction word times sixteen. Therefore, the machine-code offset is in terms of 16-byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by sixteen.

Operation

```

vAddr ← AR[s] + (024imm4104)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if invalid(attributes) then
    EXCVADDR ← vAddr
    Exception (cause)
else
    dhitwriteback(vAddr, pAddr)
    send bus transaction to other coherent cores
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Memory Group (see [Memory Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option

Implementation Notes

- Some Xtensa ISA implementations do not support write-back caches. For these implementations, the `DCWB` instruction only sends the bus transaction.

8.3.76 DCWBI—Data Cache Coherent Hit WB Invalidate

Instruction Word (RRI4)

23	20	19	16	15	12	11	8	7	4	3	0							
imm4	0	1	1	1	0	1	1	1	1	s	1	0	0	0	0	0	1	0
4	4				4				4				4					

Required Configuration Option

Data Cache Coherent Invalidation Option (See [Data Cache Coherent Invalidation Option](#))

Assembler Syntax

```
DCWBI as, 0..240
```

Description

DCWBI forces dirty data in the data cache to be written back to memory. If the specified address is not in the data cache, then this instruction has no local effect. If the specified address is present and modified in the data cache, the line containing it is written back. After the write-back, if any, the line containing the specified address is invalidated if present. If the specified line has been locked by a `DPFL` instruction, then no invalidation is done and no exception is raised because of the lock. The line is written back but remains in the cache unmodified and must be unlocked by a `DHU` or `DIU` instruction before it can be invalidated.

If hardware coherence is supported, a transaction is sent on the bus which requests other hardware coherent cores also to writeback and invalidate any copies of the line they might have.

DCWBI forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135) as if it were loading from the virtual address.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dhitwritebackinval` function.

Assembler Note

To form a virtual address, DCWBI calculates the sum of address register `as` and the `imm4` field of the instruction word times sixteen. Therefore, the machine-code offset is in terms of 16-byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by sixteen.

Operation

```
vAddr ← AR[s] + (024 | imm4 | 04)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if invalid(attributes) then
    EXCVADDR ← vAddr
    Exception (cause)
else
    dhitwritebackinval(vAddr, pAddr)
```

```

    send bus transaction to other coherent cores
endif

```

Exceptions

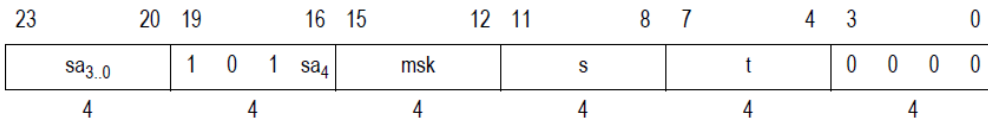
- EveryInstR Group (see [EveryInstR Group](#))
- Memory Group (see [Memory Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option

Implementation Notes

- Some Xtensa ISA implementations do not support write-back caches. For these implementations DCWBI is identical to DCI.

8.3.77 DEPBITS—Deposit Bits

Instruction Word (RRR)



Required Configuration Option

Deposit Bits Option (See [Deposit Bits Option](#) on page 96)

Assembler Syntax

```
DEPBITS at, as, shiftimm, maskimm
```

Description

DEPBITS deposits a field into an arbitrary position in a 32-bit address register. Specifically, it shifts the `maskimm` low bits of address register `as` left by `shiftimm` and replaces the corresponding bits of address register `at`. `maskimm` can take the values 1 to 16 and is encoded as `maskimm-1` in bits 15 to 12 of the instruction word. `shiftimm` can take the values 0 to 31 and is placed in bits 16 and 23 to 20 of the instruction word (the `sa` fields).

The operation of this instruction when `shiftimm + maskimm > 32` is undefined and reserved for future use.

Operation

```

mask ← 132-maskimm-shiftimm | 0maskimm | 1shiftimm
AR[t] ← (mask and AR[t]) or (032-maskimm-shiftimm | AR[s]maskimm-1..0 | 0shiftimm)

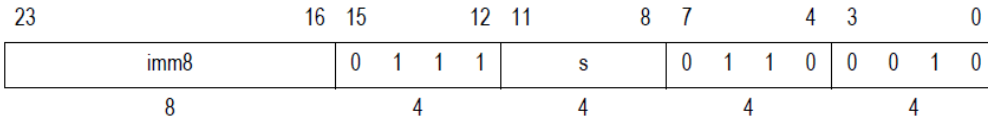
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.78 DHI—Data Cache Hit Invalidate

Instruction Word (RRI8)



Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165)

Assembler Syntax

```
DHI as, 0..1020
```

Description

DHI invalidates the specified line in the level-1 data cache, if it is present. If the specified address is not in the data cache, then this instruction has no effect. If the specified address is present, it is invalidated even if it contains dirty data. If the specified line has been locked by a DPFL instruction, then no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by a DHU or DIU instruction before it can be invalidated. This instruction is useful before a DMA write to memory that overwrites the entire line.

DHI forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135). Protection is tested as if the instruction were storing to the virtual address.

Because the organization of caches is implementation-specific, the operation below specifies only a call to the implementation's `dhitinval` function.

Whether or not DHI is a privileged instruction is implementation dependent.

Assembler Note

To form a virtual address DHI calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4

byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```

if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (022imm8|02)
    (pAddr, attributes, cause) ← stranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
    else
        dhitinval(vAddr, pAddr)
    endif
endif

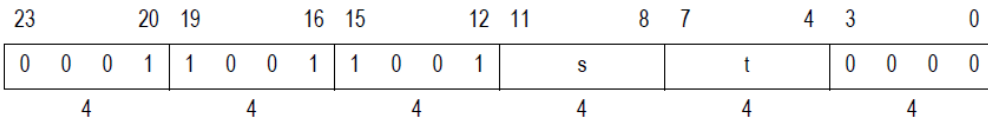
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Memory Group (see [Memory Group](#))
- GenExcep(StoreProhibitedCause) if Region Protection Option or MMU Option
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.79 DHI.B—Block Data Cache Hit Invalidate

Instruction Word (RRR)



Required Configuration Option

Block Prefetch Option (See [Block Prefetch Option](#))

Assembler Syntax

```
DHI.B as, at
```

Description

DHI.B operates on a block of bytes in the data cache which begins at the virtual address contained in address register *as*. The block is contiguous in virtual address space and its length is indicated by address register *at*. Execution breaks up the block into zero, one or two partial cache lines at the beginning and/or end of the block and some number of full

cache lines between. It does a `DHWBI` operation on the partial cache lines at the beginning and/or end and a `DHI` operation on each full cache line between.

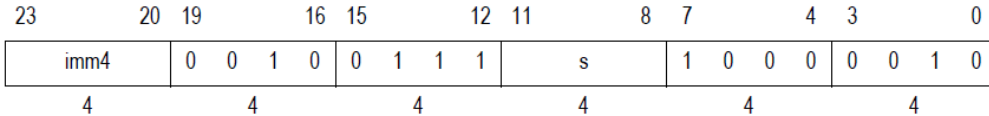
To maintain locally immediate functionality, if the processor does a subsequent load, store or software prefetch instruction to a memory location which is within the block but has not yet been operated on, the subsequent instruction waits until after the block operation has been completed on its location.

Exceptions

- Memory Group (see [Memory Group](#))
-
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option

8.3.80 DHU—Data Cache Hit Unlock

Instruction Word (RRI4)



Required Configuration Option

Data Cache Index Lock Option (See [Data Cache Index Lock Option](#))

Assembler Syntax

```
DHU as, 0..240
```

Description

`DHU` performs a data cache unlock if hit. The purpose of `DHU` is to remove the lock created by a `DPFL` instruction. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed.

`DHU` checks whether the line containing the specified address is present in the data cache, and if so, it clears the lock associated with that line. To unlock by index without knowing the address of the locked line, use the `DIU` instruction.

`DHU` forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see [The Exception Cause Register](#)

([EXCCAUSE](#)) under the [Exception Option 2](#) on page 135) as if it were loading from the virtual address.

DHU is a privileged instruction.

Assembler Note

To form a virtual address DHU calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```

if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (024||imm4||04)
    (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
    else
        dhitunlock(vAddr, pAddr)
    endif
endif

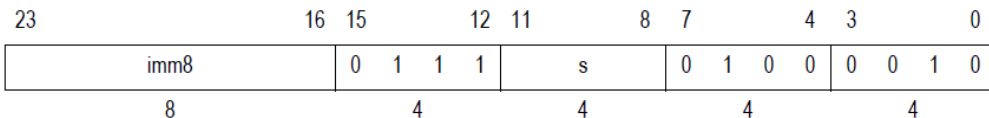
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Memory Group (see [Memory Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.81 DHWB—Data Cache Hit Writeback

Instruction Word (RRI8)



Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165)

Assembler Syntax

```
DHWB as, 0..1020
```

Description

This instruction forces dirty data in the data cache to be written back to memory. If the specified address is not in the data cache or is present but unmodified, then this instruction has no effect. If the specified address is present and modified in the data cache, the line containing it is written back, and marked unmodified. This instruction is useful before a DMA read from memory, to force writes to a frame buffer to become visible, or to force writes to memory shared by two processors.

DHWB forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135) as if it were loading from the virtual address.

Because the organization of caches is implementation-specific, the operation below specifies only a call to the implementation's `dhitwriteback` function.

Assembler Note

To form a virtual address DHWB calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + (022imm8102)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if invalid(attributes) then
    EXCVADDR ← vAddr
    Exception (cause)
else
    dhitwriteback(vAddr, pAddr)
endif
```

Exceptions

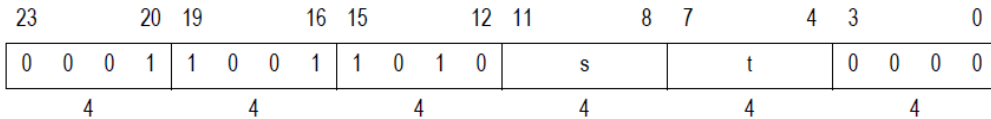
- EveryInstR Group (see [EveryInstR Group](#))
- Memory Group (see [Memory Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option

Implementation Notes

Some Xtensa ISA implementations do not support write-back caches. For these implementations, the DHWB instruction performs no operation.

8.3.82 DHWB.B—Block Data Cache Hit Writeback

Instruction Word (RRR)



Required Configuration Option

Block Prefetch Option (See [Block Prefetch Option](#))

Assembler Syntax

```
DHWB.B as, at
```

Description

DHWB.B operates on a block of bytes in the data cache which begins at the virtual address contained in address register `as`. The block is contiguous in virtual address space and its length is indicated by address register `at`. Execution breaks up the block into zero, one or two partial cache lines at the beginning and/or end of the block and some number of full cache lines between. It does a DHWB operation on the partial cache lines and on each full cache line between.

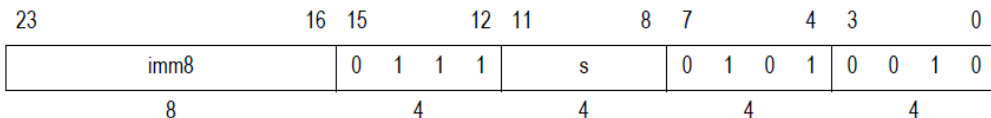
To maintain locally immediate functionality, if the processor does a subsequent store to a memory location which is within the block but has not yet been operated on, the store waits until after the operation has been completed on its location.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Memory Group (see [Memory Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option

8.3.83 DHWB.I—Data Cache Hit Writeback Invalidate

Instruction Word (RRI8)



Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165)

Assembler Syntax

```
DHWBI as, 0..1020
```

Description

`DHWBI` forces dirty data in the data cache to be written back to memory. If the specified address is not in the data cache, then this instruction has no effect. If the specified address is present and modified in the data cache, the line containing it is written back. After the write-back, if any, the line containing the specified address is invalidated if present. If the specified line has been locked by a `DPFL` instruction, then no invalidation is done and no exception is raised because of the lock. The line is written back but remains in the cache unmodified and must be unlocked by a `DHU` or `DIU` instruction before it can be invalidated. This instruction is useful in the same circumstances as `DHWB` and before a DMA write to memory or write from another processor to memory. If the line is certain to be completely overwritten by the write, you can use a `DHI` (as it is faster), but otherwise use a `DHWBI`.

`DHWBI` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises an exception (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135) as if it were loading from the virtual address.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dhitwritebackinval` function.

Assembler Note

To form a virtual address, `DHWBI` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + (022imm8102)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if invalid(attributes) then
    EXCVADDR ← vAddr
    Exception (cause)
else
    dhitwritebackinval(vAddr, pAddr)
endif
```

Exceptions

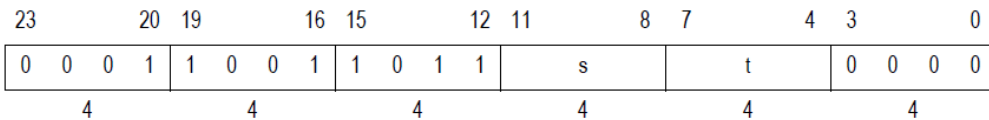
- EveryInstR Group (see [EveryInstR Group](#))
- Memory Group (see [Memory Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option

Implementation Notes

Some Xtensa ISA implementations do not support write-back caches. For these implementations `DHWBI` is identical to `DHI`.

8.3.84 *DHWBI.B—Block Data Cache Hit Writeback Inv.*

Instruction Word (RRR)



Required Configuration Option

Block Prefetch Option (See [Block Prefetch Option](#))

Assembler Syntax

```
DHWBI.B as, at
```

Description

`DHWBI.B` operates on a block of bytes in the data cache which begins at the virtual address contained in address register `as`. The block is contiguous in virtual address space and its length is indicated by address register `at`. Execution breaks up the block into zero, one or two partial cache lines at the beginning and/or end of the block and some number of full cache lines between. It does a `DHWBI` operation on the partial cache lines and on each full cache line between.

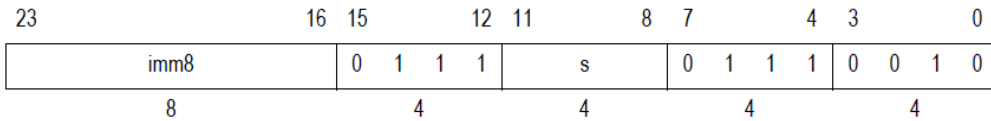
To maintain locally immediate functionality, if the processor does a subsequent load, store or software prefetch instruction to a memory location which is within the block but has not yet been operated on, the subsequent instruction waits until after the block operation has been completed on its location.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Memory Group (see [Memory Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option

8.3.85 DII—Data Cache Index Invalidate

Instruction Word (RRI8)



Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165)

Assembler Syntax

```
DII as, 0..1020
```

Description

DII uses the virtual address to choose a location in the data cache and invalidates the specified line. If the chosen line has been locked by a DPFL instruction, then no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by a DHU or DIU instruction before it can be invalidated. The method for mapping the virtual address to a data cache location is implementation-specific. This instruction is primarily useful for data cache initialization after powerup.

DII forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dindexinval` function.

DII is a privileged instruction.

Assembler Note

To form a virtual address, DII calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (022imm8!02)
```

```
dindexinval (vAddr)
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index $\text{Addr}_{x-1..z}$ in a direct-mapped cache or way $\text{Addr}_{x-1..y}$ and index $\text{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache) the instruction does nothing. In some implementations all ways at index $\text{Addr}_{y-1..z}$ are invalidated regardless of the specified way, but for future compatibility this behavior should not be assumed.

The additional ways invalidated in some implementations mean that care is needed in using this instruction with write-back caches. Dirty data in any way (at the specified index) of the cache will be lost and not just dirty data in the specified way. Because the instruction is primarily used at reset, this will not usually cause any difficulty.

8.3.86 DIU—Data Cache Index Unlock

Instruction Word (RR14)

23	20	19		16	15		12	11		8	7		4	3		0		
imm4	0	0	1	1	0	1	1	1	1	s	1	0	0	0	0	0	1	0
4																		

Required Configuration Option

Data Cache Index Lock Option (See [Data Cache Index Lock Option](#))

Assembler Syntax

```
DIU as, 0..240
```

Description

DIU uses the virtual address to choose a location in the data cache and unlocks the chosen line. The purpose of DIU is to remove the lock created by a DPFL instruction. The method for mapping the virtual address to a data cache location is implementation-specific. This instruction is primarily useful for unlocking the entire data cache. Xtensa ISA implementations

that do not implement cache locking must raise an illegal instruction exception when this opcode is executed.

To unlock a specific cache line if it is in the cache, use the `DHU` instruction.

`DIU` forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dindexunlock` function.

`DIU` is a privileged instruction.

Assembler Note

To form a virtual address `DIU` calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (024||imm4||04)
    dindexunlock(vAddr)
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index $\text{Addr}_{x-1..z}$ in a direct-mapped cache or way $\text{Addr}_{x-1..y}$ and index $\text{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

8.3.87 DIV0.D—Divide Begin Double

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	1	1	1	1	1	1	0	1	1	1	0	0	0	0
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
DIV0.D fr, fs
```

Description

DIV0.D is the first step of a Newton-Raphson divide sequence which includes corrections to make it an IEEE compliant divide. The double-precision argument in floating-point register `fs` first has its range narrowed in the same way as the `NEXP01.D` instruction (see [Assembler Syntax](#)), but without the negation. A rough approximation of the reciprocal of that result is computed by table lookup and placed in `fr`. No status flags are updated. This instruction is not intended for use anywhere but in a divide sequence. For more on the IEEE exact divide sequence, see [Divide and Square Root Sequences](#) on page 110.

Operation

```
FR[r] ← begin_divide_sequence(FR[s])
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.88 DIV0.S—Divide Begin Single

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	1	1	1	0	1	0	0	1	1	1	0	0	0	0
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
DIV0.S fr, fs
```

Description

`DIV0.S` is the first step of a Newton-Raphson divide sequence which includes corrections to make it an IEEE compliant divide. The single-precision argument in floating-point register `fs` first has its range narrowed in the same way as the `NEXP01.S` instruction (see [Assembler Syntax](#)), but without the negation. A rough approximation of the reciprocal of that result is computed by table lookup and placed in `fr`. No status flags are updated. This instruction is not intended for use anywhere but in a divide sequence. For more on the IEEE exact divide sequence, see [Divide and Square Root Sequences](#) on page 110.

Operation

```
FR[r] ← begin_divide_sequence(FR[s])
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.89 DIVN.D—Divide Final Double

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	1	1	1	1	1	1	1	r	s	t	0	0	0	0
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
DIVN.D fr, fs, ft
```

Description

Using IEEE754 double-precision arithmetic, `DIVN.D` multiplies the contents of floating-point registers `fs` and `ft`, adds the product to the contents of floating-point register `fr`, and then writes the sum back to floating-point register `fr`. The computation is performed with no

intermediate round. But `DIVN.D` differs from `MADD.D` in that it interprets the exponents of the arguments in `fr` and `ft` as containing adjustments appropriate to finishing the divide or square root sequences and in that it does not set the Invalid flag. For more on the divide and square root sequences (see [Divide and Square Root Sequences](#) on page 110).

Operation

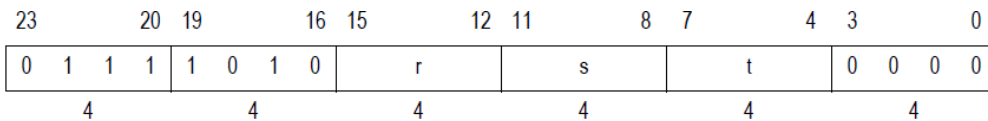
```
FR[r] ← FR[r] +D (FR[s] ×D FR[t]) (×D does not round, special exp interpretation)
FSR[StatusFlags: OUI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.90 DIVN.S—Divide Final Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
DIVN.S fr, fs, ft
```

Description

Using IEEE754 double-precision arithmetic, `DIVN.S` multiplies the contents of floating-point registers `fs` and `ft`, adds the product to the contents of floating-point register `fr`, and then writes the sum back to floating-point register `fr`. The computation is performed with no intermediate round. But `DIVN.S` differs from `MADD.S` in that it interprets the exponents of the arguments in `fr` and `ft` as containing adjustments appropriate to finishing the divide or square root sequences and in that it does not set the Invalid flag. For more on the divide and square root sequences (see [Divide and Square Root Sequences](#) on page 110).

Operation

```
FR[r] ← FR[r] +S (FR[s] ×S FR[t]) (×S does not round, special exp interpretation)
FSR[StatusFlags: OUI] ← Or in update
```


Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.91 DIWB—Data Cache Index Write Back

Instruction Word (RRI4)

23	20	19	16	15	12	11	8	7	4	3	0						
imm4	0	1	0	0	0	1	1	1	s	1	0	0	0	0	0	1	0
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165) (added in T1050)

Assembler Syntax

```
DIWB as, 0..240
```

Description

`DIWB` uses the virtual address to choose a line in the data cache and writes that line back to memory if it is dirty. The method for mapping the virtual address to a data cache line is implementation-specific. This instruction is primarily useful for forcing all dirty data in the cache back to memory. If the chosen line is present but unmodified, then this instruction has no effect. If the chosen line is present and modified in the data cache, it is written back, and marked unmodified. For set-associative caches, only one line out of one way of the cache is written back. Some Xtensa ISA implementations do not support writeback caches. For these implementations `DIWB` does nothing.

This instruction is useful for the same purposes as `DHWB`, but when either the address is not known or when the range of addresses is large enough that it is faster to operate on the entire cache.

`DIWB` forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dindexwriteback` function.

`DIWB` is a privileged instruction.

Assembler Note

To form a virtual address `DIWB` calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```

if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (024||imm4||04)
    dindexwriteback(vAddr)
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```

x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)

```

The cache line specified by index $\text{Addr}_{x-1..z}$ in a direct-mapped cache or way $\text{Addr}_{x-1..y}$ and index $\text{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

Some Xtensa ISA implementations do not support write-back caches. For these implementations, the `DIWB` instruction has no effect.

8.3.92 *DIWBI—Data Cache Index Write Back Invalidate*

Instruction Word (RRI4)

23	20	19	16	15	12	11	8	7	4	3	0						
imm4	0	1	0	1	0	1	1	1	s	1	0	0	0	0	0	1	0
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165) (added in T1050)

Assembler Syntax

```
DIWBI as, 0..240
```

Description

`DIWBI` uses the virtual address to choose a line in the data cache and forces that line to be written back to memory if it is dirty. After the writeback, if any, the line is invalidated. The method for mapping the virtual address to a data cache location is implementation-specific. If the chosen line is already invalid, then this instruction has no effect. If the chosen line has been locked by a `DPFL` instruction, then dirty data is written back but no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by a `DHU` or `DIU` instruction before it can be invalidated. For set-associative caches, only one line out of one way of the cache is written back and invalidated. Some Xtensa ISA implementations do not support write-back caches. For these implementations `DIWBI` is similar to `DII` but invalidates only one line.

This instruction is useful for the same purposes as the `DHWBI` but when either the address is not known, or when the range of addresses is large enough that it is faster to operate on the entire cache.

`DIWBI` forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dindexwritebackinval` function.

`DIWBI` is a privileged instruction.

Assembler Note

To form a virtual address, `DIWBI` calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (024||imm4||04)
    dindexwritebackinval(vAddr)
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index $\text{Addr}_{x-1..z}$ in a direct-mapped cache or way $\text{Addr}_{x-1..y}$ and index $\text{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

8.3.93 DIWBUI.P—Data Cache Empty

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0
0	0	0	0	1	1	1	1	0	1	1	0
4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165)

Assembler Syntax

```
DIWBUI.P as
```

Description

DIWBUI.P uses the virtual address to choose a line in the data cache, unlocks that line, forces that line to be written back to memory if it is dirty, invalidates the line, and increments the address register `as` by the size of a data cache line. The method for mapping the virtual address to a data cache location is implementation-specific. For set-associative caches, only one line out of one way of the cache is written back and invalidated. Some Xtensa ISA implementations do not support write-back caches.

This instruction is useful for the fastest clearing of the data cache, including locked lines, without destruction of data. It may be used before shutting down all or part of the cache.

DIWBUI.P forms a virtual address simply by using the contents of address register `as`. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dindexwritebackinval` function.

`DIWBUI.P` is a privileged instruction.

Operation

```

if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s]
    dindexunlockwritebackinval(vAddr)
    AR[s] ← AR[s] + DataCacheLineBytes
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```

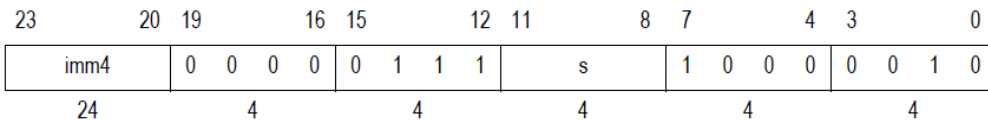
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)

```

The cache line specified by index $\text{Addr}_{x-1..z}$ in a direct-mapped cache or way $\text{Addr}_{x-1..y}$ and index $\text{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

8.3.94 DPFL—Data Cache Prefetch and Lock

Instruction Word (RRI4)



Required Configuration Option

Data Cache Index Lock Option (See [Data Cache Index Lock Option](#))

Assembler Syntax

```
DPFL as, 0..240
```

Description

DPFL performs a data cache prefetch and lock. The purpose of DPFL is to improve performance, and not to affect state defined by the ISA. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed. In general, the performance improvement from using this instruction is implementation-dependent.

DPFL checks if the line containing the specified address is present in the data cache, and if not, it begins the transfer of the line from memory to the cache. The line is placed in the data cache and the line marked as locked, that is not replaceable by ordinary data cache misses. To unlock the line, use DHU or DIU. To prefetch without locking, use the DPFR, DPFW, DPFR0, or DPFW0 instructions.

DPFL forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. The translation is done as if the address were for a load.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dprefetch` function.

DPFL is a privileged instruction.

Assembler Note

To form a virtual address, DPFL calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (024||imm4||04)
    (pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
    else
        dprefetch(vAddr, pAddr, 0, 0, 1)
    endif
endif
```

Exceptions

- Memory Group (see [Memory Group](#))

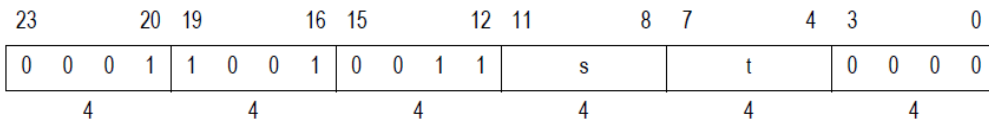
-
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- Privileged Instruction Group (see [Privileged Instruction Group](#))

Implementation Notes

If, before the instruction executes, there are not two available DataCache ways at the required index, a Load Store Error exception (GenExcep(LoadStoreErrorCause) is raised.

8.3.95 DPFM.B—Block Data Cache Prefetch and Modify

Instruction Word (RRR)



Required Configuration Option

Block Prefetch Option (See [Block Prefetch Option](#))

Assembler Syntax

```
DPFM.B as, at
```

Description

DPFM.B operates on a block of bytes in the data cache which begins at the virtual address contained in address register *as*. The block is contiguous in virtual address space and its length is indicated by address register *at*. Execution breaks up the block into zero, one or two partial cache lines at the beginning and/or end of the block and some number of full cache lines between. It does a DDPFW operation on the partial cache lines at the beginning and/or end and for every full cache line between it allocates a line and sets the data in the line to an arbitrary value without necessarily reading the current value of the line. The purpose is to reduce the bandwidth that would otherwise have been wasted reading the current value of the line. Coherency is maintained in coherent systems.

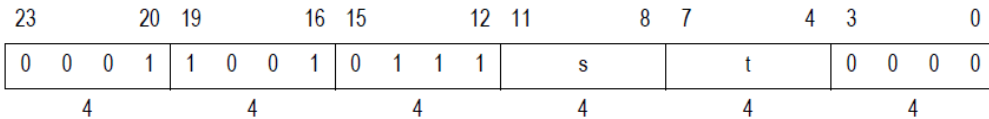
To maintain locally immediate functionality, if the processor does a subsequent load or store instruction to a memory location which is within the block but has not yet been prefetched or allocated, the subsequent instruction waits until after the block operation has been completed on its location. Similarly, if the processor does a subsequent cache operation which would invalidate a memory location which is within the block but has not yet been prefetched or allocated, the subsequent instruction waits until after the block operation has been completed on its location.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.96 DPFM.BF—Block Data Cache Prefetch/Modify First

Instruction Word (RRR)



Required Configuration Option

Block Prefetch Option (See [Block Prefetch Option](#))

Assembler Syntax

```
DPFM.BF as, at
```

Description

DPFM.BF operates on a block of bytes in the data cache which begins at the virtual address contained in address register *as*. The block is contiguous in virtual address space and its length is indicated by address register *at*. Execution breaks up the block into zero, one or two partial cache lines at the beginning and/or end of the block and some number of full cache lines between. It does a DDPFW operation on the partial cache lines at the beginning and/or end and for every full cache line between it allocates a line and sets the data in the line to an arbitrary value without necessarily reading the current value of the line. The purpose is to reduce the bandwidth that would otherwise have been wasted reading the current value of the line. Coherency is maintained in coherent systems.

In addition to its operation, DPFM.BF affects the execution of multiple block operations. Instead of interleaving its operation with previous block operations, it waits until all previous block operations have completed. It also causes all following block operations to wait in the same way.

To maintain locally immediate functionality, if the processor does a subsequent load or store instruction to a memory location which is within the block but has not yet been prefetched or allocated, the subsequent instruction waits until after the block operation has been completed on its location. Similarly, if the processor does a subsequent cache operation which would invalidate a memory location which is within the block but has not yet been prefetched or allocated, the subsequent instruction waits until after the block operation has been completed on its location.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.97 DPFRR—Data Cache Prefetch for Read

Instruction Word (RRI8)

Operation

```
vAddr ← AR[s] + (022Imm8102)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if not invalid(attributes) then
    dprefetch(vAddr, pAddr, 0, 0, 0)
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.98 DPFR.B—Block Data Cache Prefetch for Read

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	0	0	1	1	0	0	1	0	0	0	1	s	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Block Prefetch Option (See [Block Prefetch Option](#))

Assembler Syntax

```
DPFR.B as, at
```

Description

DPFR.B operates on a block of bytes in the data cache which begins at the virtual address contained in address register `as`. The block is contiguous in virtual address space and its length is indicated by address register `at`. Execution breaks up the block into zero, one or two partial cache lines at the beginning and/or end of the block and some number of full cache lines between. It does a DPFR operation on the partial cache lines and on each full cache line between.

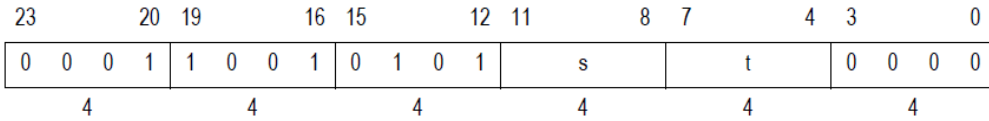
To maintain locally immediate functionality, if the processor does a subsequent cache operation which would invalidate a memory location which is within the block but has not yet been prefetched, the subsequent instruction waits until after the block operation has been completed on its location.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.99 DPFR.BF—Block Data Cache Prefetch for Read First

Instruction Word (RRR)



Required Configuration Option

Block Prefetch Option (See [Block Prefetch Option](#))

Assembler Syntax

```
DPFR.BF as, at
```

Description

DPFR.BF operates on a block of bytes in the data cache which begins at the virtual address contained in address register *as*. The block is contiguous in virtual address space and its length is indicated by address register *at*. Execution breaks up the block into zero, one or two partial cache lines at the beginning and/or end of the block and some number of full cache lines between. It does a DPFR operation on the partial cache lines and on each full cache line between.

In addition to its operation, DPFR.BF affects the execution of multiple block operations. Instead of interleaving its prefetches with previous block operations, it waits until all previous block operations have completed. It also causes all following block operations to wait in the same way.

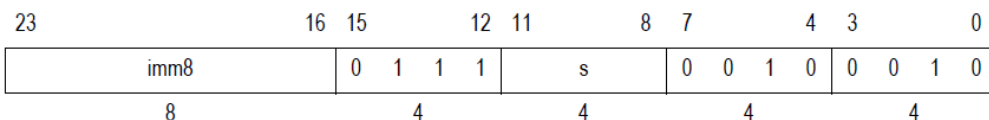
To maintain locally immediate functionality, if the processor does a subsequent cache operation which would invalidate a memory location which is within the block but has not yet been prefetched, the subsequent instruction waits until after the block operation has been completed on its location.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.100 DPFR.O—Data Cache Prefetch for Read Once

Instruction Word (RRI8)



Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165)

Assembler Syntax

```
DPFRO as, 0..1020
```

Description

`DPFRO` performs a data cache prefetch for read once. The purpose of `DPFRO` is to improve performance, but not to affect state defined by the ISA. Therefore, some Xtensa ISA implementations may choose to implement this instruction as a simple “no-operation” instruction. In general, the performance improvement from using this instruction is implementation-dependent. Refer to a specific *Xtensa Microprocessor Data Book* for more details.

In some Xtensa ISA implementations, `DPFRO` checks whether the line containing the specified address is present in the data cache, and if not, it begins the transfer of the line from memory. Four data prefetch instructions provide different “hints” about how the data is likely to be used in the future. `DPFRO` indicates that the data is only likely to be read once before it is replaced by another line in the cache. In some implementations, this hint might be used to select a specific cache way or to select a streaming buffer instead of the cache.

`DPFRO` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. No exception is raised on either translation or memory reference. Instead of raising an exception, the prefetch is dropped and the instruction becomes a `nop`.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation’s `dprefetch` function.

Assembler Note

To form a virtual address, `DPFRO` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + (022imm8|02)  
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
```

```

if not invalid(attributes) then
    dprefetch(vAddr, pAddr, 0, 1, 0)
endif

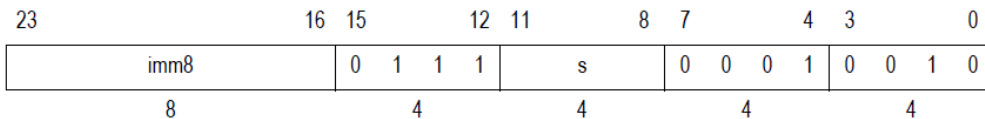
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.101 DPFW—Data Cache Prefetch for Write

Instruction Word (RRI8)



Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165)

Assembler Syntax

```
DPFW as, 0..1020
```

Description

DPFW performs a data cache prefetch for write. The purpose of DPFW is to improve performance, but not to affect the ISA state. Therefore, some Xtensa ISA implementations may choose to implement this instruction as a simple “no-operation” instruction. In general, the performance improvement from using this instruction is implementation-dependent. Refer to a specific *Xtensa Microprocessor Data Book* for more details.

In some Xtensa ISA implementations, DPFW checks whether the line containing the specified address is present in the data cache, and if not, begins the transfer of the line from memory. Four data prefetch instructions provide different “hints” about how the data is likely to be used in the future. DPFW indicates that the data is likely to be written before it is replaced by another line in the cache. In some implementations, this fetches the data with write permission (for example, in a system with shared and exclusive states).

DPFW forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. No exception is raised on either translation or memory reference. Instead of raising an exception, the prefetch is dropped and the instruction becomes a nop.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `dprefetch` function.

Assembler Note

To form a virtual address `DPFW` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offsets and encodes this into the instruction by dividing by four.

Operation

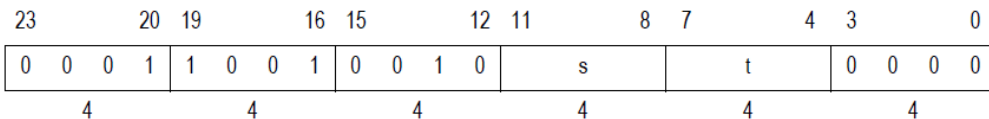
```
vAddr ← AR[s] + (022imm8102)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if not invalid(attributes) then
    dprefetch(vAddr, pAddr, 1, 0, 0)
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.102 DPFW.B—Block Data Cache Prefetch for Write

Instruction Word (RRR)



Required Configuration Option

Block Prefetch Option (See [Block Prefetch Option](#))

Assembler Syntax

```
DPFW.B as, at
```

Description

`DPFW.B` operates on a block of bytes in the data cache which begins at the virtual address contained in address register `as`. The block is contiguous in virtual address space and its length is indicated by address register `at`. Execution breaks up the block into zero, one or two partial cache lines at the beginning and/or end of the block and some number of full cache lines between. It does a `DPFW` operation on the partial cache lines and on each full cache line between.

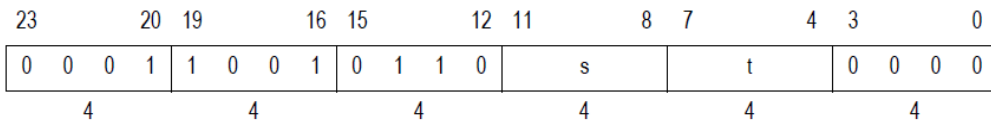
To maintain locally immediate functionality, if the processor does a subsequent cache operation which would invalidate a memory location which is within the block but has not yet been prefetched, the subsequent instruction waits until after the block operation has been completed on its location.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.103 DPFW.BF—Block Data Cache Prefetch for Write First

Instruction Word (RRR)



Required Configuration Option

Block Prefetch Option (See [Block Prefetch Option](#))

Assembler Syntax

```
DPFW.BF as, at
```

Description

DPFW.BF operates on a block of bytes in the data cache which begins at the virtual address contained in address register *as*. The block is contiguous in virtual address space and its length is indicated by address register *at*. Execution breaks up the block into zero, one or two partial cache lines at the beginning and/or end of the block and some number of full cache lines between. It does a DPFW operation on the partial cache lines and on each full cache line between.

In addition to its operation, DPFW.BF affects the execution of multiple block operations. Instead of interleaving its prefetches with previous block operations, it waits until all previous block operations have completed. It also causes all following block operations to wait in the same way.

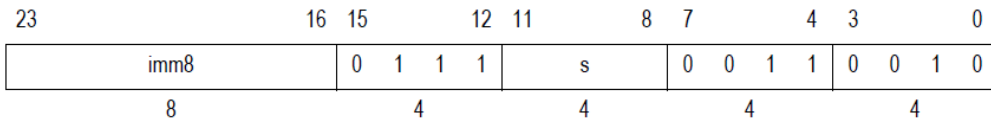
To maintain locally immediate functionality, if the processor does a subsequent cache operation which would invalidate a memory location which is within the block but has not yet been prefetched, the subsequent instruction waits until after the block operation has been completed on its location.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.104 DPFWO—Data Cache Prefetch for Write Once

Instruction Word (RRI8)



Required Configuration Option

Data Cache Option (See [Data Cache Option](#) on page 165)

Assembler Syntax

```
DPFWO as, 0..1020
```

Description

DPFWO performs a data cache prefetch for write once. The purpose of DPFWO is to improve performance, but not to affect the ISA state. Therefore, some Xtensa ISA implementations may choose to implement this instruction as a simple “no-operation” instruction. In general, the performance improvement from using this instruction is implementation-dependent. Refer to a specific *Xtensa Microprocessor Data Book* for more details.

In some Xtensa ISA implementations, DPFWO checks whether the line containing the specified address is present in the data cache, and if not, begins the transfer of the line from memory. Four data prefetch instructions provide different “hints” about how the data is likely to be used in the future. DPFWO indicates that the data is likely to be read and written once before it is replaced by another line in the cache. In some implementations, this write hint fetches the data with write permission (for example, in a system with shared and exclusive states). The write-once hint might be used to select a specific cache way or to select a streaming buffer instead of the cache.

DPFWO forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. No exception is raised on either translation or memory reference. Instead of raising an exception, the prefetch is dropped and the instruction becomes a `nop`.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation’s `dpprefetch` function.

Assembler Note

To form a virtual address `DPFWO` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

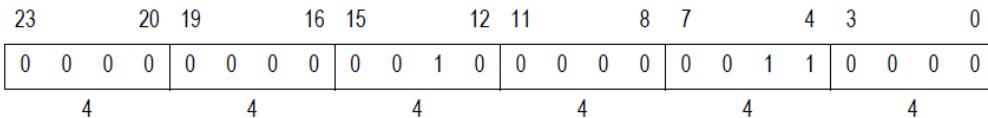
```
vAddr ← AR[s] + (022||imm8||02)
(pAddr, attributes, cause) ← ltranslate(vAddr, CRING)
if not invalid(attributes) then
    dprefetch(vAddr, pAddr, 1, 1, 0)
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.105 DSYNC—Load/Store Synchronize

Instruction Word (RRR)



Required Configuration Option

(Core Architecture) (See [Core Architecture](#) on page 77)

Assembler Syntax

```
DSYNC
```

Description

`DSYNC` waits for all previously fetched `WSR.*`, `XSR.*`, `WDTLB`, and `IDTLB` instructions to be performed before interpreting the virtual address of the next load or store instruction. This operation is also performed as part of `ISYNC`, `RSYNC`, and `ESYNC`.

This instruction is appropriate after `WSR.DBREAKC*` and `WSR.DBREAKA*` instructions. See the Special Register Tables in [Special Registers](#) on page 272 and [TLB Entries](#) on page 317 for a complete description of the uses of the `DSYNC` instruction.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation's `dsync` function.

Operation

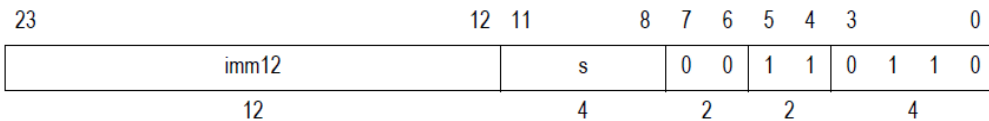
```
dsync ()
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.106 ENTRY—Subroutine Entry

Instruction Word (BRI12)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
ENTRY as, 0..32760
```

Description

`ENTRY` is intended to be the first instruction of all subroutines called with `CALL4`, `CALL8`, `CALL12`, `CALLX4`, `CALLX8`, or `CALLX12`. This instruction is not intended to be used by a routine called by `CALL0` or `CALLX0`.

`ENTRY` serves two purposes:

1. First, it increments the register window pointer (`WindowBase`) by the amount requested by the caller as recorded in the `PS.CALLINC` field.
2. Second, it copies the stack pointer from caller to callee and allocates the callee's stack frame. The `as` operand specifies the stack pointer register; it must specify one of `a0..a3` or the operation of `ENTRY` is undefined. It is read before the window is moved, the stack frame size is subtracted, and then the `as` register in the moved window is written.

The stack frame size is specified as the 12-bit unsigned `imm12` field in units of eight bytes. The size is zero-extended, shifted left by 3, and subtracted from the caller's stack pointer to get the callee's stack pointer.

Stack frames up to 32760 bytes can be specified. The initial stack frame size must be a constant, but subsequently the `MOVSP` instruction can be used to allocate dynamically sized objects on the stack, or to further extend a constant stack frame larger than 32760 bytes.

The windowed subroutine call protocol is described in [Windowed Procedure-Call Protocol](#) on page 248.

ENTRY is undefined under the Windowed Register Option if PS.WOE is 0 or if PS.EXCM is 1. Some implementations raise an illegal instruction exception in these cases, as a debugging aid.

Assembler Note

In the assembler syntax, the number of bytes to be subtracted from the stack pointer is specified as the immediate. The assembler encodes this into the instruction by dividing by eight.

Operation

```
WindowCheck (00, PS.CALLINC, 00)

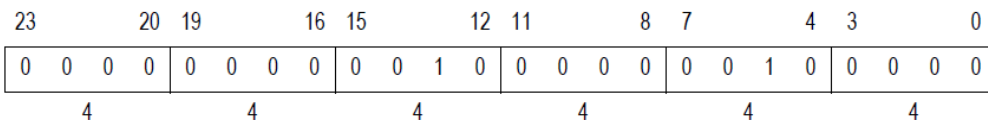
elsif as > 3 | PS.WOE = 0 | PS.EXCM = 1 then
    -- undefined operation
    -- may raise illegal instruction exception
else
    AR[PS.CALLINC#s1..0] ← AR[s] - (017#imm12#03)
    WindowBase ← WindowBase + (02#PS.CALLINC)
    WindowStartWindowBase ← 1
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.107 ESYNC—Execute Synchronize

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
ESYNC
```

Description

ESYNC waits for all previously fetched WSR.* , and XSR.* instructions to be performed before the next instruction uses any register values. This operation is also performed as part of ISYNC and RSYNC. DSYNC is performed as part of this instruction.

This instruction is appropriate after WSR.EPC* instructions. See the Special Register Tables in [Special Registers](#) on page 272 for a complete description of the uses of the ESYNC instruction.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation’s esync function.

Operation

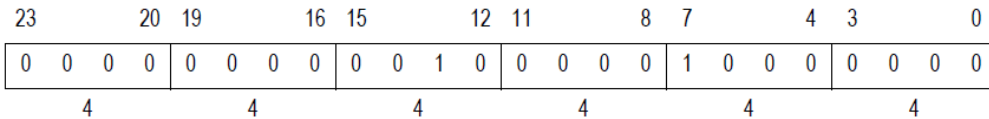
```
esync ()
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.108 EXCW—Exception Wait

Instruction Word (RRR)



Required Configuration Option

Exception Option 2 (See [Exception Option 2](#) on page 126)

Assembler Syntax

```
EXCW
```

Description

EXCW waits for any exceptions of previously executed instructions to be handled. Some Xtensa ISA implementations may have imprecise exceptions; on these implementations EXCW waits until exceptions raised by all previous instructions are taken or the instructions are known to be exception-free. Because the instruction execution pipeline and exception handling is implementation-specific, the operation section below specifies only a call to the implementation’s ExceptionWait function.

Operation

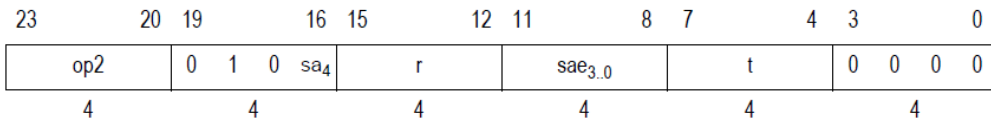
```
ExceptionWait()
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.109 EXTUI—Extract Unsigned Immediate

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
EXTUI ar, at, shiftimm, maskimm
```

Description

EXTUI performs an unsigned bit field extraction from a 32-bit register value. Specifically, it shifts the contents of address register `at` right by the shift amount `shiftimm`, which is a value `0..31` stored in bits `16` and `11..8` of the instruction word (the `sa` fields). The shift result is then ANDed with a mask of `maskimm` least-significant 1 bits and the result is written to address register `ar`. The number of mask bits, `maskimm`, may take the values `1..16`, and is stored in the `op2` field as `maskimm-1`. The bits extracted are therefore `sa+op2..sa`.

The operation of this instruction when `sa+op2 > 31` is undefined and reserved for future use.

Operation

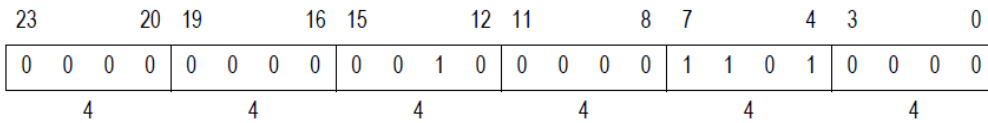
```
mask ← 031-op21op2+1  
AR[r] ← (032 | AR[t])31+sa..sa and mask
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.110 EXTW—External Wait

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77) (added in RA-2004.1)

Assembler Syntax

```
EXTW
```

Description

EXTW is a superset of MEMW. EXTW ensures that both

- all previous load, store, acquire, release, prefetch, and cache instructions; and
- any other effect of any previous instruction which is visible at the pins of the Xtensa processor

complete (or perform as described in [Memory Access Ordering](#) on page 115) before either

- any subsequent load, store, acquire, release, prefetch, or cache instructions; or
- external effects of the execution of any following instruction is visible at the pins of the Xtensa processor (not including instruction prefetch or TIE Queue pops)

is allowed to begin.

While MEMW is intended to implement the `volatile` attribute of languages such as C and C++, EXTW is intended to be an ordering guarantee for all external effects that the processor can have, including processor pins defined in TIE.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation's `extw` function.

Operation

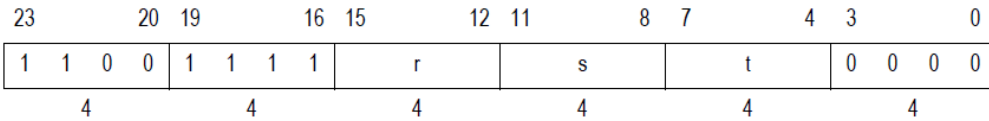
```
extw()
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.111 FLOAT.D—Convert Fixed to Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
Float.D fr, as, 0..15
```

Description

Float.D converts the contents of address register `as` from signed integer to double-precision format. The converted integer value is then scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 0.5, 0.25, ..., $1.0 \div_D 32768.0$. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0` and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. The result is written to floating-point register `fr`.

Operation

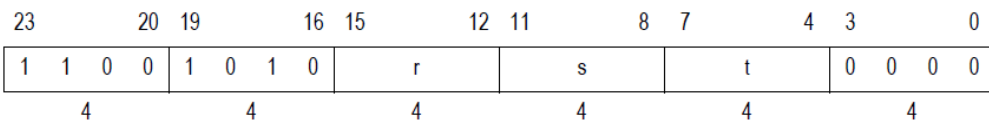
```
FR[r] ← floatD(AR[s]) ×D powD(2.0, -t)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.112 FLOAT.S—Convert Fixed to Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
FLOAT.S fr, as, 0..15
```

Description

`FLOAT.S` converts the contents of address register `as` from signed integer to single-precision format, rounding according to the current rounding mode. The converted integer value is then scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 0.5, 0.25, ..., $1.0 \div_s 32768.0$. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0` and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. The result is written to floating-point register `fr`.

Operation

```
FR[r] ← floats(AR[s]) ×s pows(2.0, -t)  
FSR[StatusFlags: I] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.113 FLOOR.D—Floor Double to Fixed

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	1	0	1	1	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
FLOOR.D ar, fs, 0..15
```

Description

`FLOOR.D` converts the contents of floating-point register `fs` from double-precision to signed integer format, rounding toward $-\infty$. The double-precision value is first scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 2.0, 4.0, ..., 32768.0.

The scaling allows for a fixed point notation where the binary point is at the right end of the integer for $t=0$ and moves to the left as t increases until for $t=15$ there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $\geq 2^{31}$), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (scaled argument $< -2^{31}$) or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

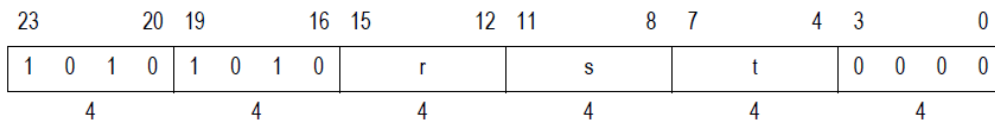
```
AR[r] ← floorD(FR[s] ×D powD(2.0,t))
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.114 FLOOR.S—Floor Single to Fixed

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
FLOOR.S ar, fs, 0..15
```

Description

`FLOOR.S` converts the contents of floating-point register `fs` from single-precision to signed integer format, rounding toward $-\infty$. The single-precision value is first scaled by a power of two constant value encoded in the `t` field, with `0..15` representing 1.0, 2.0, 4.0, ..., 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for $t=0$ and moves to the left as t increases until for $t=15$ there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $\geq 2^{31}$), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (scaled argument $< -2^{31}$) or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

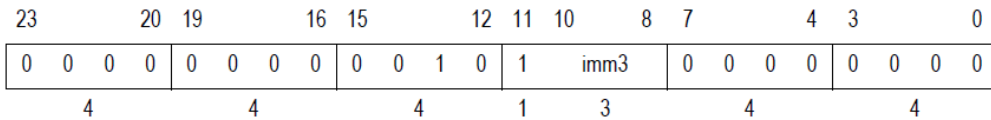
```
AR[r] ← floors(FR[s] ×s pows(2.0,t))  
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
Coprocessor Group (see [Coprocessor Group](#))

8.3.115 FSYNC—Fetch Synchronize

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
FSYNC
```

Description

FSYNC provides for synchronizations internal to the Instruction Fetch Unit. Its operation is implementation defined.

Operation

```
fsync ()
```

Exceptions

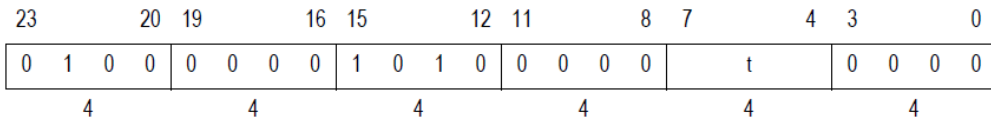
- EveryInst Group (see [EveryInst Group](#))

Implementation Notes

- FSYNC may consume considerably more cycles than RSYNC, ESYNC, or DSYNC.

8.3.116 GETEX—Get Exclusive Result

Instruction Word (RRR)



Required Configuration Option

Exclusive Access Option (See [Exclusive Access Option](#) on page 123)

Assembler Syntax

```
GETEX at
```

Description

GETEX waits for any outstanding update to bit[8] ATOMCTL from an outstanding S32EX and then exchanges bit[8] of ATOMCTL with bit[0] of the address register at and zeros the remaining bits of address register at. See [Exclusive Access Option](#) on page 123.

GETEX is intended to follow an S32EX instruction (see [Assembler Syntax](#)). The pair implements what is a store exclusive in some architectures. The two are separated to improve interrupt latency. If both functions were done with a single instruction, the state save for an interrupt would need to wait for the memory system to acknowledge the write.

Operation

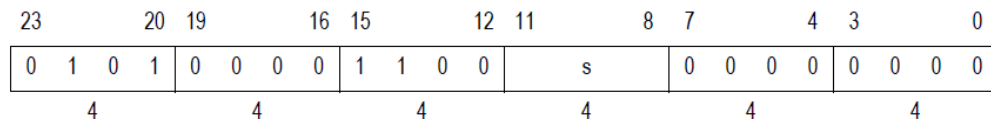
```
temp ← ATOMCTL8
ATOMCTL8 ← AR[t]0
AR[t] ← 031||temp
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.117 IDTLB—Invalidate Data TLB Entry

Instruction Word (RRR)



Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
IDTLB as
```

Description

IDTLB invalidates the data TLB entry specified by the contents of address register `as`. See [Options for Memory Protection and Translation](#) on page 183 for information on the address register formats for specific Memory Protection and Translation Options. The point at which the invalidation is effected is implementation-specific. Any translation that would be affected by this invalidation before the execution of a `DSYNC` instruction is therefore undefined.

IDTLB is a privileged instruction.

The representation of validity in Xtensa TLBs is implementation-specific, and thus the operation section below writes the implementation-specific value `InvalidDataTLBEntry`.

Operation

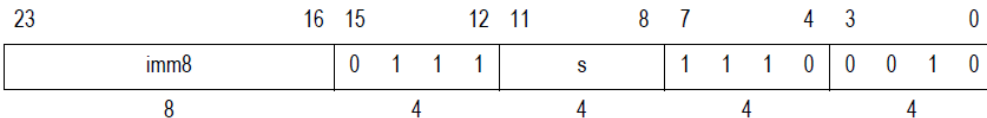
```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    (vpn, ei, wi) ← SplitDataTLBEntrySpec(AR[s])
    DataTLB[wi][ei] ← InvalidDataTLBEntry
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.118 IHI—Instruction Cache Hit Invalidate

Instruction Word (RRI8)



Required Configuration Option

Instruction Cache Option (See [Instruction Cache Option](#) on page 164)

Assembler Syntax

```
IHI as, 0..1020
```

Description

`IHI` performs an instruction cache hit invalidate. It invalidates the specified line in the instruction cache, if it is present. If the specified address is not in the instruction cache, then this instruction has no effect. If the specified line is already invalid, then this instruction has no effect. If the specified line has been locked by an `IPFL` instruction, then no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by an `IHU` or `IIU` instruction before it can be invalidated. Otherwise, if the specified line is present, it is invalidated.

This instruction is required before executing instructions from the instruction cache that have been written by this processor, another processor, or DMA. The writes must first be forced out of the data cache, either by using `DHWB` or by using stores that bypass or write through the data cache. An `ISYNC` instruction should then be used to guarantee that the modified instructions are visible to instruction cache misses. The instruction cache should then be invalidated for the affected addresses using a series of `IHI` instructions. An `ISYNC` instruction should then be used to guarantee that this processor's fetch pipeline does not contain instructions from the invalidated lines.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `ihitival` function.

`IHI` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example, protection violation), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135). The translation is done as if the address were for an instruction fetch.

Assembler Note

To form a virtual address, `IHI` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + (022imm8|02)
(pAddr, attributes, cause) ← ftranslate(vAddr, CRING)
if invalid(attributes) then
    EXCVADDR ← vAddr
    Exception (cause)
else
```

```

    ihibitval(vAddr, pAddr)
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- MemoryErrorException if Memory ECC/Parity Option

8.3.119 IHU—Instruction Cache Hit Unlock

Instruction Word (RRI4)

23	20	19	16	15	12	11	8	7	4	3	0						
imm4	0	0	1	0	0	1	1	1	s	1	1	0	1	0	0	1	0
4	4				4				4	4				4			

Required Configuration Option

Instruction Cache Index Lock Option (See [Instruction Cache Index Lock Option](#))

Assembler Syntax

```

IHU as, 0..240

```

Description

IHU performs an instruction cache unlock if hit. The purpose of IHU is to remove the lock created by an IPFL instruction. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed.

IHU checks whether the line containing the specified address is present in the instruction cache, and if so, it clears the lock associated with that line. To unlock by index without knowing the address of the locked line, use the IIU instruction.

IHU forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation encounters an error (for example or protection violation), the processor takes one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135). The translation is done as if the address were for an instruction fetch.

IHU is a privileged instruction.

Assembler Note

To form a virtual address, IHU calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```

if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (024imm4104)
    (pAddr, attributes, cause) ← ftranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
    else
        ihitunlock(vAddr, pAddr)
    endif
endif

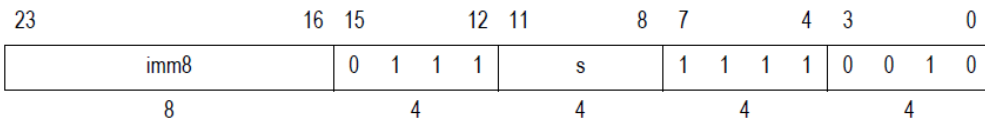
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

8.3.120 III—Instruction Cache Index Invalidate

Instruction Word (RRI8)



Required Configuration Option

Instruction Cache Option (See [Instruction Cache Option](#) on page 164)

Assembler Syntax

```
III as, 0..1020
```

Description

III performs an instruction cache index invalidate. This instruction uses the virtual address to choose a location in the instruction cache and invalidates the specified line. The method for mapping the virtual address to an instruction cache location is implementation-specific. If the chosen line is already invalid, then this instruction has no effect. If the chosen line has

been locked by an `IPFL` instruction, then no invalidation is done and no exception is raised because of the lock. The line remains in the cache and must be unlocked by an `IHU` or `IIU` instruction before it can be invalidated. This instruction is useful for instruction cache initialization after power-up or for invalidating the entire instruction cache. An `ISYNC` instruction should then be used to guarantee that this processor's fetch pipeline does not contain instructions from the invalidated lines.

`III` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `iindexinval` function.

Whether or not `III` is a privileged instruction is implementation dependent.

Assembler Note

To form a virtual address, `III` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (022imm8102)
    iindexinval(vAddr, pAddr)
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

Implementation Notes

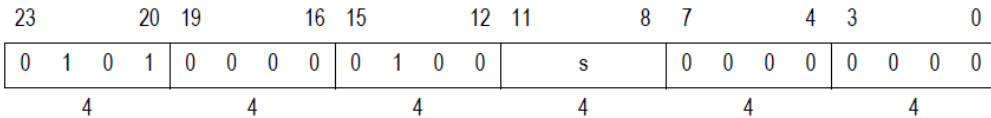
```
x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)
```

The cache line specified by index `Addrx-1..z` in a direct-mapped cache or way `Addrx-1..y` and index `Addry-1..z` in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing. In some

implementations all ways at index $\text{Addr}_{y-1..z}$ are invalidated regardless of the specified way, but for future compatibility this behavior should not be assumed.

8.3.121 IITLB—Invalidate Instruction TLB Entry

Instruction Word (RRR)



Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
IITLB as
```

Description

`IITLB` invalidates the instruction TLB entry specified by the contents of address register `as`. See [Options for Memory Protection and Translation](#) on page 183 for information on the address register formats for specific Memory Protection and Translation options. The point at which the invalidation is effected is implementation-specific. Any translation that would be affected by this invalidation before the execution of an `ISYNC` instruction is therefore undefined.

`IITLB` is a privileged instruction.

The representation of validity in Xtensa TLBs is implementation-specific, and thus the operation section below writes the implementation-specific value `InvalidInstTLBEntry`.

Operation

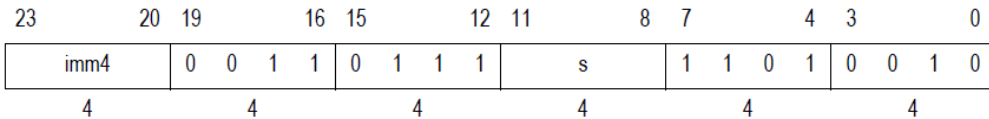
```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    (vpn, ei, wi) ← SplitInstTLBEntrySpec(AR[s])
    InstTLB[wi][ei] ← InvalidInstTLBEntry
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.122 IIU—Instruction Cache Index Unlock

Instruction Word (RRI4)



Required Configuration Option

Instruction Cache Index Lock Option (See [Instruction Cache Index Lock Option](#))

Assembler Syntax

```
IIU as, 0..240
```

Description

IIU uses the virtual address to choose a location in the instruction cache and unlocks the chosen line. The purpose of IIU is to remove the lock created by an IPFL instruction. The method for mapping the virtual address to an instruction cache location is implementation-specific. This instruction is primarily useful for unlocking the entire instruction cache. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed. In some implementations, IIU invalidates the cache line in addition to unlocking it.

To unlock a specific cache line if it is in the cache, use the IHU instruction.

IIU forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. The virtual address chooses a cache line without translation and without raising the associated exceptions.

Because the organization of caches is implementation-specific, the operation section below specifies only a call to the implementation's `iindexunlock` function.

IIU is a privileged instruction.

Assembler Note

To form a virtual address IIU calculates the sum of address register `as` and the `imm4` field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```
if CRING ≠ 0 then
```

```

Exception (PrivilegedCause)
else
  vAddr ← AR[s] + (024Imm4104)
  iindexunlock(vAddr)
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```

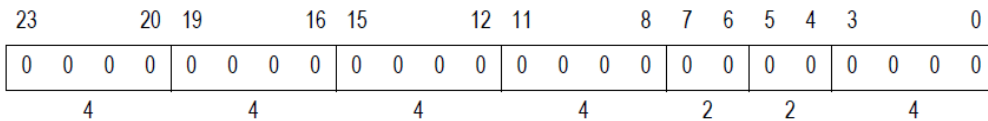
x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)

```

The cache line specified by index $\text{Addr}_{x-1..z}$ in a direct-mapped cache or way $\text{Addr}_{x-1..y}$ and index $\text{Addr}_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing.

8.3.123 ILL—Illegal Instruction

Instruction Word (CALLX)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
ILL
```

Description

ILL is an opcode that does whatever illegal opcodes do in the implementation. Often that is to raise an illegal instruction exception. It provides a way to test what happens to an illegal opcode and reduces the probability that data will be successfully executed. For a 16-bit version, see ILL.N.

Operation

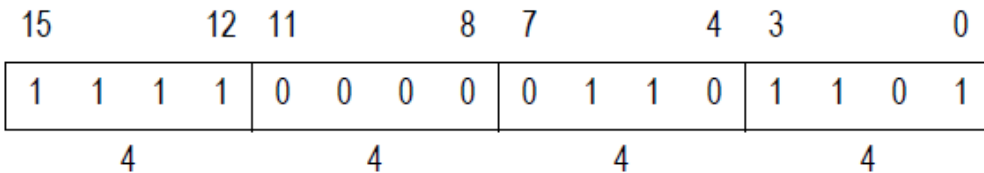
Exception (IllegalInstructionCause)

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2

8.3.124 ILL.N—Narrow Illegal Instruction

Instruction Word (RRRN)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

ILL.N

Description

ILL.N is a 16-bit opcode that does whatever illegal opcodes do in the implementation. For a 24-bit version, see ILL.

Operation

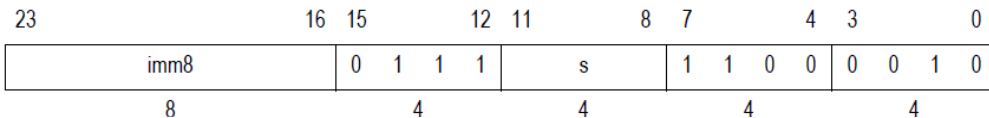
Exception (IllegalInstructionCause)

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2

8.3.125 IPF—Instruction Cache Prefetch

Instruction Word (RRI8)



Required Configuration Option

Instruction Cache Option (See [Instruction Cache Option](#) on page 164)

Assembler Syntax

```
IPF as, 0..1020
```

Description

`IPF` performs an instruction cache prefetch. The purpose of `IPF` is to improve performance, but not to affect state defined by the ISA. Therefore, some Xtensa ISA implementations may choose to implement this instruction as a simple “no-operation” instruction. In general, the performance improvement from using this instruction is implementation-dependent. In some implementations, `IPF` checks whether the line containing the specified address is present in the instruction cache, and if not, it begins the transfer of the line from memory to the instruction cache. Prefetching an instruction line may prevent the processor from taking an instruction cache miss later. Refer to a specific *Xtensa Microprocessor Data Book* for more details.

`IPF` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. The translation is done as if the address were for an instruction fetch. No exception is raised on either translation or memory reference. Instead of raising an exception, the prefetch is dropped and the instruction becomes a `nop`.

Assembler Note

To form a virtual address, `IPF` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + (022imm8102)
(pAddr, attributes, cause) ← ftranslate(vAddr, CRING)
if not invalid(attributes) then
```

```

    iprefetch(vAddr, pAddr, 0)
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.126 IPFL—Instruction Cache Prefetch and Lock

Instruction Word (RRI4)

23	20	19	16	15	12	11	8	7	4	3	0						
imm4	0	0	0	0	0	1	1	1	s	1	1	0	1	0	0	1	0
4	4				4			4	4			4					

Required Configuration Option

Instruction Cache Index Lock Option (See [Instruction Cache Index Lock Option](#))

Assembler Syntax

```

IPFL as, 0..240

```

Description

IPFL performs an instruction cache prefetch and lock. The purpose of IPFL is to improve performance, but not to affect state defined by the ISA. Xtensa ISA implementations that do not implement cache locking must raise an illegal instruction exception when this opcode is executed. In general, the performance improvement from using this instruction is implementation-dependent as implementations may not overlap the cache fill with the execution of other instructions.

In some implementations, IPFL checks whether the line containing the specified address is present in the instruction cache, and if not, begins the transfer of the line from memory to the instruction cache. The line is placed in the instruction cache and marked as locked, so it is not replaceable by ordinary instruction cache misses. To unlock the line, use IHU or IIU. To prefetch without locking, use the IPF instruction.

IPFL forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by four. Therefore, the offset can specify multiples of 16 from zero to 240. If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. The translation is done as if the address were for an instruction fetch. Exceptions are reported exactly as they would be for an instruction fetch. For exceptions fetching the IPFL instruction, EXCVADDR will point to one of the bytes of the

IPFL instruction. For exceptions fetching the cache line, EXCVADDR will point to the cache line. EPC points to the IPFL instruction in both cases.

IPFL is a privileged instruction.

Assembler Note

To form a virtual address, IPFL calculates the sum of address register *as* and the *imm4* field of the instruction word times 16. Therefore, the machine-code offset is in terms of 16 byte units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by 16.

Operation

```

if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (024||imm4||04)
    (pAddr, attributes, cause) ← ftranslate(vAddr, CRING)
    if invalid(attributes) then
        EXCVADDR ← vAddr
        Exception (cause)
    else
        iprefetch(vAddr, pAddr, 1)
    endif
endif

```

Exceptions

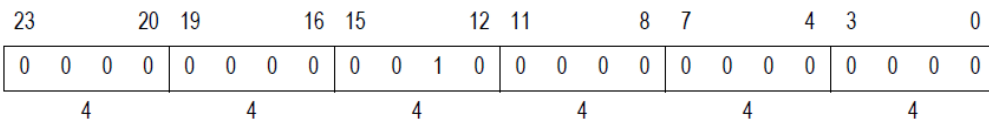
- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

Implementation Notes

If, before the instruction executes, there are not two available InstCache ways at the required index, an Instruction Fetch Error exception (GenExcep(InstructionFetchErrorCause)) is raised.

8.3.127 ISYNC—Instruction Fetch Synchronize

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
ISYNC
```

Description

`ISYNC` waits for all previously fetched load, store, cache, TLB, `WSR.*`, and `XSR.*` instructions that affect instruction fetch to be performed before fetching the next instruction. `RSYNC`, `ESYNC`, and `DSYNC` are performed as part of this instruction.

The proper sequence for writing instructions and then executing them is:

- write instructions
- use `DHWB` to force the data out of the data cache (this step may be skipped if it is not possible for the data to be dirty in the data cache)
- use `MEMW` to wait for the writes to be visible to instruction cache misses
- use multiple `IHI` instructions to invalidate the instruction cache for any lines that were modified (this step may be skipped, along with one of the `ISYNC` steps on either side, if the affected instructions are in InstRAM or cannot be cached)
- use `ISYNC` to ensure that fetch pipeline will see the new instructions

This instruction also waits for all previously executed `WSR.*` and `XSR.*` instructions that affect instruction fetch or register access processor state, including:

- `WSR.LCOUNT`, `WSR.LBEG`, `WSR.LEND`
- `WSR.IBREAKENABLE`, `WSR.IBREAKA[i]`
- `WSR.CCOMPAREn`

See the Special Register Tables in [Special Registers](#) on page 272 and [Caches and Local Memories](#) on page 318, for a complete description of the `ISYNC` instruction's uses.

Operation

```
isync()
```

Exceptions

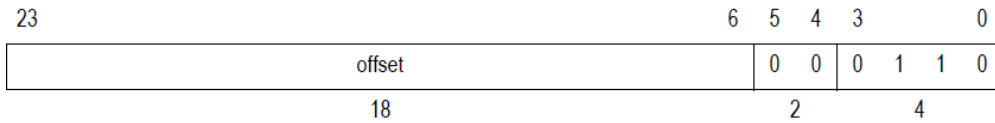
- EveryInst Group (see [EveryInst Group](#))

Implementation Notes

In many implementations, `ISYNC` consumes considerably more cycles than `RSYNC`, `ESYNC`, or `DSYNC`.

8.3.128 J—Unconditional Jump

Instruction Word (CALL)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
J label
```

Description

J performs an unconditional branch to the target address. It uses a signed, 18-bit PC-relative offset to specify the target address. The target address is given by the address of the J instruction plus the sign-extended 18-bit `offset` field of the instruction plus four, giving a range of -131068 to $+131075$ bytes.

Operation

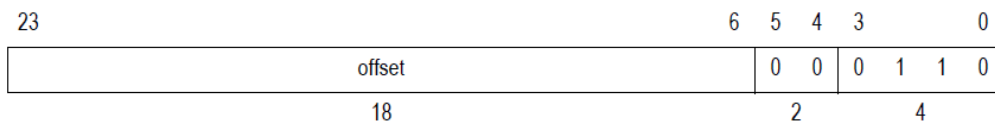
```
nextPC ← PC + (offset17:14 | offset) + 4
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.129 J.L—Unconditional Jump Long

Instruction Word (CALL)



Required Configuration Option

Assembler Macro

Assembler Syntax

```
J.L label, an
```

Description

`J.L` is an assembler macro which generates exactly a `J` instruction as long as the offset will reach the label. If the offset is not long enough, the assembler relaxes the instruction to a literal load into `an` followed by a `JX an`. The AR register `an` may or may not be modified.

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.130 JX—Unconditional Jump Register

Instruction Word (CALLX)

23	20	19	16	15	12	11	8	7	6	5	4	3	0						
0	0	0	0	0	0	0	0	0	s	1	0	1	0	0	0	0	0		
4				4				4				2		2		4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
JX as
```

Description

`JX` performs an unconditional jump to the address in register `as`.

Operation

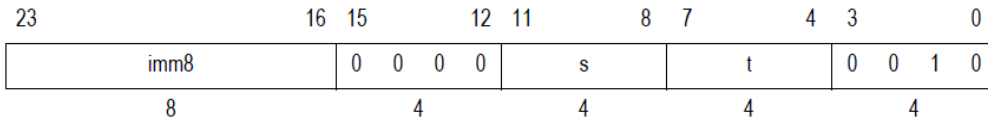
```
nextPC ← AR[s]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.131 L8UI—Load 8-bit Unsigned

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
L8UI at, as, 0..255
```

Description

L8UI is an 8-bit unsigned load from memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word. Therefore, the offset ranges from 0 to 255. Eight bits (one byte) are read from the physical address. This data is then zero-extended and written to address register `at`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Operation

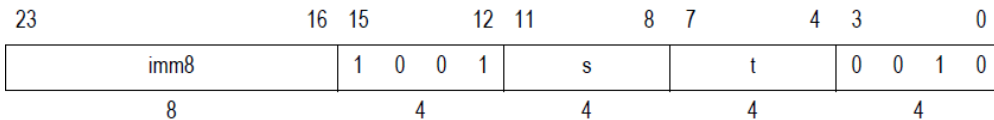
```
vAddr ← AR[s] + (024∥imm8)
(mem8, error) ← Load8(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    AR[t] ← 024∥mem8
endif
```

Exceptions

- Memory Group (see [Memory Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- DebugExcep(DBREAK) if Debug Option

8.3.132 L16SI—Load 16-bit Signed

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
L16SI at, as, 0..510
```

Description

L16SI is a 16-bit signed load from memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by 1. Therefore, the offset can specify multiples of two from zero to 510. Sixteen bits (two bytes) are read from the physical address. This data is then sign-extended and written to address register `at`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation, non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the least significant address bit is ignored; a reference to an odd address produces the same result as a reference to the address minus one. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, L16SI calculates the sum of address register `as` and the `imm8` field of the instruction word times two. Therefore, the machine-code offset is in terms of 16-bit (2 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by two.

Operation

```
vAddr ← AR[s] + (023imm810)
(mem16, error) ← Load16(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
```

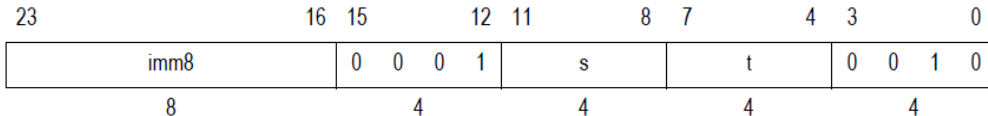
```
AR[t] ← mem1615:16 | mem16
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.133 L16UI—Load 16-bit Unsigned

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
L16UI at, as, 0..510
```

Description

L16UI is a 16-bit unsigned load from memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by 1. Therefore, the offset can specify multiples of two from zero to 510. Sixteen bits (two bytes) are read from the physical address. This data is then zero-extended and written to address register `at`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the least significant address bit is ignored; a reference to an odd address produces the same result as a reference to the address minus one. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, L16UI calculates the sum of address register `as` and the `imm8` field of the instruction word times two. Therefore, the machine-code offset is in terms of 16-bit (2 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by two.

Operation

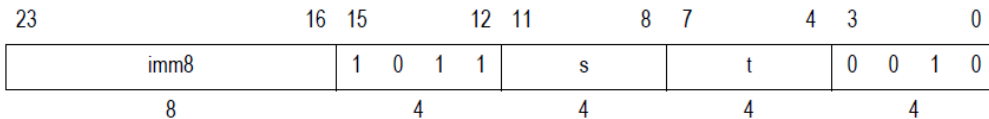
```
vAddr ← AR[s] + (023Imm810)
(mem16, error) ← Load16(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    AR[t] ← 016Imm16
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.134 L32AI—Load 32-bit Acquire

Instruction Word (RRI8)



Required Configuration Option

Multiprocessor Synchronization Option (See [Multiprocessor Synchronization Option](#) on page 115)

Assembler Syntax

```
L32AI at, as, 0..1020
```

Description

L32AI is a 32-bit load from memory with “acquire” semantics. This load performs before any subsequent loads, stores, acquires, or releases are performed. It is typically used to test a synchronization variable protecting a critical region (for example, to acquire a lock).

L32AI forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. 32 bits (four bytes) are read from the physical address. This data is then written to address register `at`. L32AI causes the processor to delay processing of subsequent loads, stores, acquires, and releases until the L32AI is performed. In some Xtensa ISA implementations, this occurs automatically and L32AI is identical to L32I. Other implementations (for example, those with multiple outstanding loads and stores) delay processing as described above. Because the method of

delay is implementation-dependent, this is indicated in the operation section below by the implementation function `acquire`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, `L32AI` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

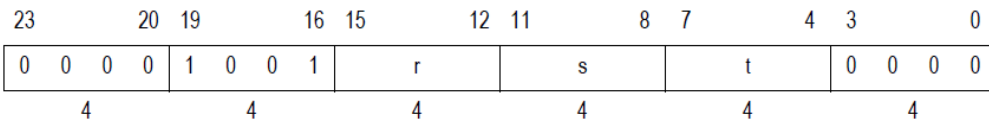
```
vAddr ← AR[s] + (022imm8102)
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    AR[t] ← mem32
    acquire()
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.135 L32E—Load 32-bit for Window Exceptions

Instruction Word (RRI4)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
L32E at, as, -64..-4
```

Description

L32E is a 32-bit load instruction similar to L32I but with semantics required by window overflow and window underflow exception handlers. In particular, memory access checking is done with PS.RING instead of CRING, and the offset used to form the virtual address is a 4-bit one-extended immediate. Therefore, the offset can specify multiples of four from -64 to -4. In configurations without the MMU Option, there is no PS.RING, and L32E is similar to L32I with a negative offset.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

L32E is a privileged instruction.

In the context of special handler interface code, L32E has modified operation.

Assembler Note

To form a virtual address, L32E calculates the sum of address register `as` and the `r` field of the instruction word times four (and one extended). Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    vAddr ← AR[s] + (126∥r∥02)
    ring ← if MMU Option then PS.RING else 0
    (mem32, error) ← Load32Ring(vAddr, ring)
    if error then
        EXCVADDR ← vAddr
        Exception (LoadStoreErrorCause)
    else
        AR[t] ← mem32
```



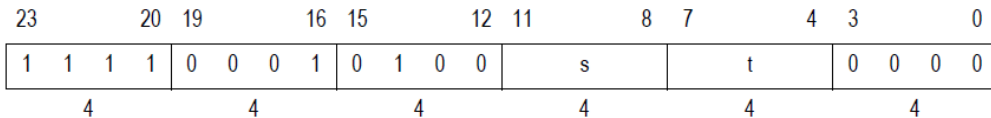
```
endif
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.136 L32EX—Load 32-bit Exclusive

Instruction Word (RRR)



Required Configuration Option

Exclusive Access Option (See [Exclusive Access Option](#) on page 123)

Assembler Syntax

```
L32EX at, as
```

Description

L32EX is a 32-bit load from memory. Its virtual address is the contents of address register `as`. This data is then written to address register `at`. In addition, the physical address being accessed is micro-architecturally marked as an exclusive access. This mark is checked by the S32EX instruction ([Assembler Syntax](#)). See [Exclusive Access Option](#) on page 123. If the target of the virtual address is not able to handle exclusive accesses, the instruction raises the `ExclusiveErrorCause` exception.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation, non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

The operation of L32EX will depend on the memory type associated with its address. It may operate entirely within a cache, by means of an ordinary external bus transaction, by means of a special external bus transaction, or by means of a series of coherent bus transactions.

Operation

```
vAddr ← AR[s]
```

```

(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause or ExclusiveErrorCause)
else
    AR[t] ← mem32
    setmonitor()
endif

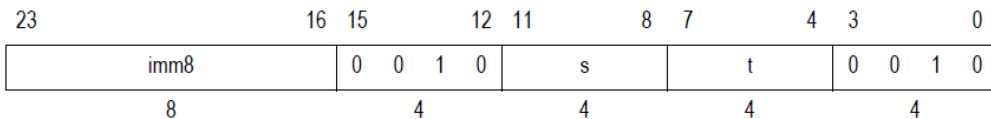
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- GenExcep(ExclusiveErrorCause) if Exception Option 2

8.3.137 L32I—Load 32-bit

Instruction Word (RR18)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
L32I at, as, 0..1020
```

Description

L32I is a 32-bit load from memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. Thirty-two bits (four bytes) are read from the physical address. This data is then written to address register `at`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation, non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

If the Instruction Memory Access Option ([Instruction Memory Access Option](#) on page 167) is configured, `L32I` is one of only a few memory reference instructions that can access instruction RAM/ROM.

Assembler Note

The assembler may convert `L32I` instructions to `L32I.N` when the Code Density Option is enabled and the immediate operand falls within the available range. Prefixing the `L32I` instruction with an underscore (`_L32I`) disables this optimization and forces the assembler to generate the wide form of the instruction.

To form a virtual address, `L32I` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

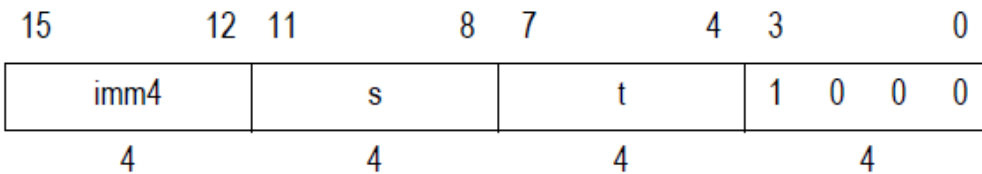
```
vAddr ← AR[s] + (022||imm8||02)
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    AR[t] ← mem32
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.138 L32I.N—Narrow Load 32-bit

Instruction Word (RRRN)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

```
L32I.N at, as, 0..60
```

Description

`L32I.N` is similar to `L32I`, but has a 16-bit encoding and supports a smaller range of offset values encoded in the instruction word.

`L32I.N` is a 32-bit load from memory. It forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 60. Thirty-two bits (four bytes) are read from the physical address. This data is then written to address register `at`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions .

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

If the Instruction Memory Access Option ([Instruction Memory Access Option](#) on page 167) is configured, `L32I.N` is one of only a few memory reference instructions that can access instruction RAM/ROM.

Assembler Note

The assembler may convert `L32I.N` instructions to `L32I`. Prefixing the `L32I.N` instruction with an underscore (`_L32I.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

To form a virtual address, `L32I.N` calculates the sum of address register `as` and the `imm4` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + (026||imm4||02)
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    AR[t] ← mem32
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.139 L32R—Load 32-bit PC-Relative

Instruction Word (RI6)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
L32R at, label
```

Description

L32R is a PC-relative 32-bit load from memory. It is typically used to load constant values into a register when the constant cannot be encoded in a `MOVI` instruction.

L32R forms a virtual address by adding the 16-bit one-extended constant value encoded in the instruction word shifted left by two to the address of the L32R plus three with the two least significant bits cleared. Therefore, the offset can always specify 32-bit aligned addresses from -262141 to -4 bytes from the address of the L32R instruction. 32 bits (four bytes) are read from the physical address. This data is then written to address register `at`.

In the presence of the Extended L32R Option ([Extended L32R Option](#) on page 86) when `LITBASE[0]` is clear, the instruction has the identical operation. When `LITBASE[0]` is set, L32R forms a virtual address by adding the 16-bit one extended constant value encoded in the instruction word shifted left by two to the literal base address indicated by the upper 20 bits of `LITBASE`. The offset can specify 32-bit aligned addresses from -262144 to -4 bytes from the literal base address.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

It is not possible to specify an unaligned address.

If the Instruction Memory Access Option (*Instruction Memory Access Option* on page 167) is configured, `L32R` is one of only a few memory reference instructions that can access instruction RAM/ROM.

Assembler Note

In the assembler syntax, the immediate operand is specified as the address of the location to load from, rather than the offset from the current instruction address. The linker and the assembler both assume that the location loaded by the `L32R` instruction has not been and will not be accessed by any other type of load or store instruction and optimizes according to that assumption.

Performance of L32R Instruction below describes `L32R` instruction performance under different conditions.

Table 196: Performance of L32R Instruction

Memory Type	Without IMA Option ¹	With IMA Option ¹
Instruction RAM/ROM (<i>Direct the Access to Local Memory</i> on page 193)	Raises Load Store Error	Variable Performance ²
Data RAM/ROM or XLMI (<i>Direct the Access to Local Memory</i> on page 193)	Fast	Fast
PIF Access (<i>Direct the Access to PIF</i> on page 196)	Slow (PIF latency)	Slow (PIF latency)
Cacheable Memory (<i>Direct the Access to Cache</i> on page 196)	Fast (thru Data Cache)	Fast (thru Data Cache)
<ol style="list-style-type: none"> 1. Column header refers to whether or not the Instruction Memory Access Option (<i>Instruction Memory Access Option</i> on page 167) is configured. 2. Fast in newer implementations but several cycles in older implementations. For older implementations it is desirable to place literal sections in another memory type. Refer to a specific <i>Xtensa Microprocessor Data Book</i> for more details. 		

Operation

```

if Extended L32R Option and LITBASE0 then
    vAddr ← (LITBASE31..121012) + (114imm16102)
else
    vAddr ← ((PC + 3)31..2102) + (114imm16102)
endif
(mem32, error) ← Load32(vAddr)
if error then

```

```

EXCVADDR ← vAddr
Exception (LoadStoreErrorCause)
else
  AR[t] ← mem32
endif

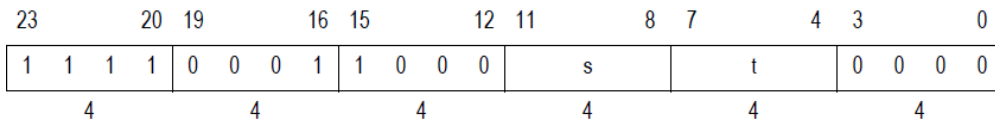
```

Exceptions

- Memory Group (see [Memory Group](#))
- GenExcep(LoadProhibitedCause) if Region Protection Option or MMU Option
- DebugExcep(DBREAK) if Debug Option

8.3.140 LDCT—Load Data Cache Tag

Instruction Word (RRR)



Required Configuration Option

Data Cache Test Option (See [Data Cache Test Option](#))

Assembler Syntax

```
LDCT at, as
```

Description

LDCT is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA and its exact method of addressing the cache may depend on the implementation.

LDCT is intended for reading the RAM arrays that implement the data cache tags or other data related memories as part of manufacturing test.

LDCT uses the contents of address register `as` to select a line in the data cache, reads the tag associated with this line, and writes the result to address register `at`. The value written to `at` is described under Cache Tag Format in [Cache Tag Format](#) on page 163. The upper four bits of address register `as` may, in some implementations, be used to choose a RAM type to access. Since LDCT addresses memory differently than most memory accesses, it is only certain to see the results of a previous store if there has been a MEMW between the two.

LDCT is a privileged instruction.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    index ← AR[s]x-1..z-2
    AR[t] ← DataCacheTag[index] // see Implementation Notes below
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

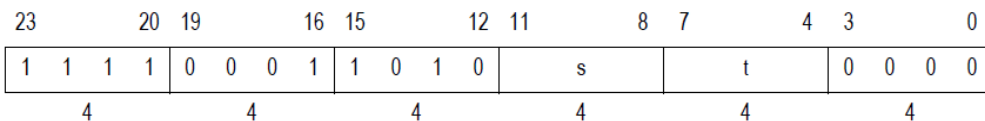
Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index $AR[s]_{x-1..z}$ in a direct-mapped cache or way $AR[s]_{x-1..y}$ and index $AR[s]_{y-1..z}$ in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction loads an undefined value. In configurations with more than one copy of Tag RAM, $AR[s]_{z-1..z-2}$ will determine which copy is read.

8.3.141 LDCW—Load Data Cache Word

Instruction Word (RRR)



Required Configuration Option

Data Cache Test Option (See [Data Cache Test Option](#))

Assembler Syntax

```
LDCW at, as
```

Description

LDCW is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA and its exact method of addressing the cache may depend on the implementation.

LDCW is intended for reading the RAM arrays that implement the data cache or other data related memories as part of manufacturing test.

LDCW uses the contents of address register `as` to select a line in the data cache and one 32-bit quantity within that line, reads that data, and writes the result to address register `at`. The upper four bits of address register `as` may, in some implementations, be used to choose a RAM type to access. Since LDCW addresses memory differently than most memory accesses, it is only certain to see the results of a previous store if there has been a MEMW between the two.

LDCW is a privileged instruction.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    index ← AR[s]x-1..2
    AR[t] ← DataCacheData [index] // see Implementation Notes below
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

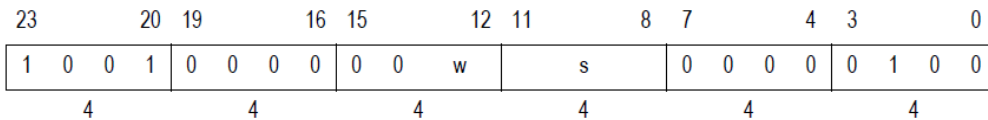
Implementation Notes

```
x ← ceil(log2(DataCacheBytes))
y ← log2(DataCacheBytes ÷ DataCacheWayCount)
z ← log2(DataCacheLineBytes)
```

The cache line specified by index `AR[s]x-1..z` in a direct-mapped cache or way `AR[s]x-1..y` and index `AR[s]y-1..z` in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction loads an undefined value. Within the cache line, `AR[s]z-1..2` is used to determine which 32-bit quantity within the line is loaded.

8.3.142 LDDEC—Load with Autodecrement

Instruction Word (RRR)



Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
LDDEC mw, as
```

Description

LDDEC loads MAC16 register `mw` from memory using auto-decrement addressing. It forms a virtual address by subtracting 4 from the contents of address register `as`. 32 bits (four bytes) are read from the physical address. This data is then written to MAC16 register `mw`, and the virtual address is written back to address register `as`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

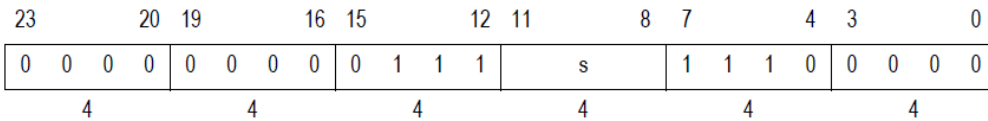
```
vAddr ← AR[s] - 4
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    MR[w] ← mem32
    AR[s] ← vAddr
endif
```

Exceptions

Memory Load Group (see [Memory Load Group](#))

8.3.143 LDDR32.P—Load to DDR Register

Instruction Word (RRR)



Required Configuration Option

Debug Option (See [Debug Option](#) on page 256) and OCD, Implementation-Specific

Assembler Syntax

```
LDDR32.P as
```

Description

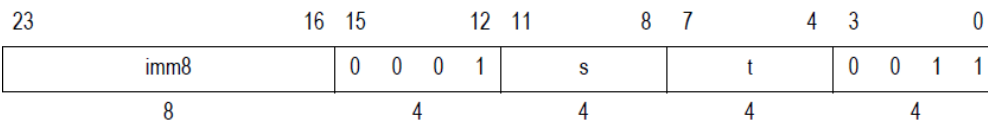
This instruction is used only in On-Chip Debug Mode and exists only in some implementations. It is an illegal instruction when the processor is not in On-Chip Debug Mode. See the *Xtensa Debug Guide* for a description of its operation.

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2

8.3.144 LDI—Load Double Immediate

Instruction Word (RR18)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
LDI ft, as, 0..2040
```

Description

LDI is a 64-bit load from memory to the floating-point register file. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value

encoded in the instruction word shifted left by three. Therefore, the offset can specify multiples of eight from zero to 2040. Sixty-four bits (eight bytes) are read from the physical address. This data is then written to floating-point register *ft*.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the three least significant bits of the address are ignored. A reference to an address that is not 0 mod 8 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, `LDI` calculates the sum of address register *as* and the *imm8* field of the instruction word times eight. Therefore, the machine-code offset is in terms of 64-bit (8 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by eight.

Operation

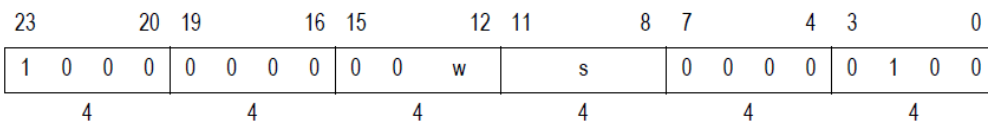
```
vAddr ← AR[s] + (021||imm8||03)
(mem64, error) ← Load64(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    FR[t] ← mem64
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.145 LDINC—Load with Autoincrement

Instruction Word (RRR)



Required Configuration Option

[MAC16 Option](#) on page 91 (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
LDINC mw, as
```

Description

LDINC loads MAC16 register *mw* from memory using auto-increment addressing. It forms a virtual address by adding 4 to the contents of address register *as*. 32 bits (four bytes) are read from the physical address. This data is then written to MAC16 register *mw*, and the virtual address is written back to address register *as*.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

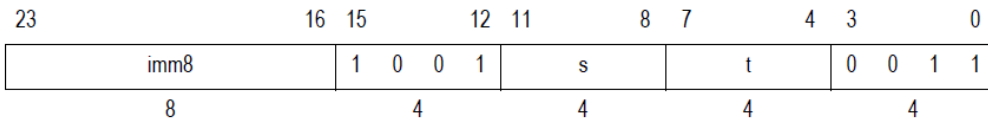
```
vAddr ← AR[s] + 4
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    MR[w] ← mem32
    AR[s] ← vAddr
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.146 LDIP—Load Double Immediate Post-Increment

Instruction Word (RRI8)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
LDIP ft, as, 0..2040
```

Description

`LDIP` is a 64-bit load from memory to the floating-point register file with base address register post-increment. The virtual address is taken from the contents of address register `as`. Sixty-four bits (eight bytes) are read from the physical address. This data is then written to floating-point register `ft`. The sum of the virtual address and an 8-bit zero-extended constant value encoded in the instruction word shifted left by three is written back to address register `as`. The increment can specify multiples of eight from zero to 2040.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the three least significant bits of the address are ignored. A reference to an address that is not 0 mod 8 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

`LDIP` calculates the increment of address register `as` using the `imm8` field of the instruction word times eight. Therefore, the machine-code increment is in terms of 64-bit (8 byte) units. However, the assembler expects a byte increment and encodes this into the instruction by dividing by eight.

Operation

```
vAddr ← AR[s]
(mem64, error) ← Load64(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
```

```

else
    FR[t] ← mem64
    AS[s] ← vAddr + (021||imm8||03)
endif

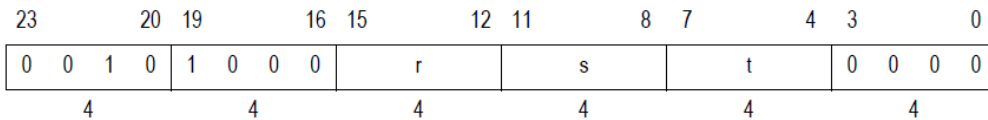
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.147 LDX—Load Double Indexed

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
LDX fr, as, at
```

Description

LDX is a 64-bit load from memory to the floating-point register file. It forms a virtual address by adding the contents of address register `as` and the contents of address register `at`. Sixty-four bits (eight bytes) are read from the physical address. This data is then written to floating-point register `fr`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the three least significant bits of the address are ignored. A reference to an address that is not 0 mod 8 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

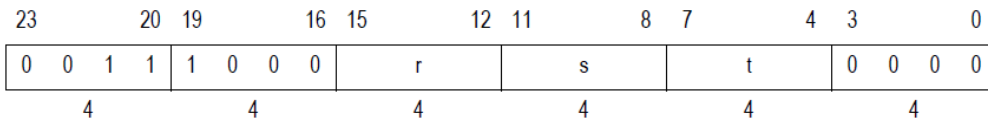
```
vAddr ← AR[s] + (AR[t])
(mem64, error) ← Load64(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    FR[r] ← mem64
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.148 LDXP—Load Double Indexed Post-Increment

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
LDXP fr, as, at
```

Description

LDXP is a 64-bit load from memory to the floating-point register file with base address register post-increment. The virtual address is taken from the contents of address register *as*. Sixty-four bits (eight bytes) are read from the physical address. This data is then written to floating-point register *fr*. The sum of the virtual address and the contents of address register *at* is written back to address register *as*.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option (*Unaligned Exception Option* on page 148), the three least significant bits of the address are ignored. A reference to an address that is not 0 mod 8 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```

vAddr ← AR[s]
(mem64, error) ← Load64(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    FR[r] ← mem64
    AR[s] ← vAddr + (AR[t])
endif

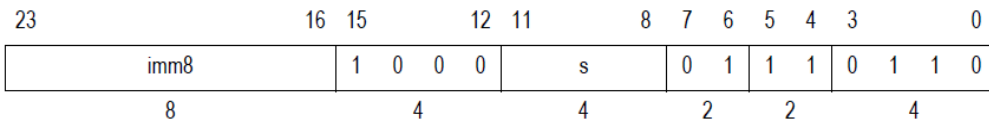
```

Exceptions

- Memory Load Group (see *Memory Load Group*)
- Coprocessor Group (see *Coprocessor Group*)

8.3.149 LOOP—Loop

Instruction Word (BR18)



Required Configuration Option

Loop Option (See *Loop Option* on page 84)

Assembler Syntax

```
LOOP as, label
```

Description

LOOP sets up a zero-overhead loop by setting the LCOUNT, LBEG, and LEND special registers, which control instruction fetch. The loop will iterate the number of times specified by address register as, with 0 causing the loop to iterate 232 times. LCOUNT, the current loop iteration counter, is loaded from the contents of address register as minus one. LEND is the loop end address and is loaded with the address of the LOOP instruction plus four, plus the zero-extended 8-bit offset encoded in the instruction (therefore, the loop code may be up to 256 bytes in length). LBEG, the loop begin address, is loaded with the address of the following instruction.

After the processor fetches an instruction that increments the `PC` to the value contained in `LEND`, and `LCOUNT` is not zero, it loads the `PC` with the contents of `LBEG` and decrements `LCOUNT`. `LOOP` is intended to be implemented with help from the instruction fetch engine of the processor, and therefore should not incur a mispredict or taken branch penalty. Branches and jumps to the address contained in `LEND` do not cause a loop back, and therefore may be used to exit the loop prematurely. Likewise, a return from a call instruction as the last instruction of the loop would not trigger loop back; this case should be avoided.

There is no mechanism to proceed to the next iteration of the loop from the middle of the loop. The compiler may insert a branch to a `NOF` placed as the last instruction of the loop to implement this function if required.

Because `LCOUNT`, `LBEG`, and `LEND` are single registers, zero-overhead loops may not be nested. Using conditional branch instructions to implement outer level loops is typically not a performance issue. Because loops cannot be nested, it is usually inappropriate to include a procedure call inside a loop (the callee might itself use a zero-overhead loop).

To simplify the implementation of zero-overhead loops, the `LBEG` address must be such that the first instruction must entirely fit within a naturally aligned four byte region or, if the fetch width is larger than four bytes, a naturally aligned region which is the size of the fetch width. Some implementations require, in addition, that the fetch width is any greater than the naturally aligned power of two region (of four bytes or larger) which is no smaller than that first instruction. When the `LOOP` instruction would not naturally be placed at such an address, the insertion of `NOF` instructions or adjustment of which instructions are 16-bit density instructions is sufficient to give it the required alignment.

The automatic loop-back when the `PC` increments to match `LEND` is disabled when `PS.EXCM` is set. This prevents non-privileged code from affecting the operation of the privileged exception vector code. Dynamic loaders need to avoid mixing new code and old register values as the combination may execute in unexpected ways.

Assembler Note

The assembler automatically aligns the `LOOP` instruction as required.

When the label is out of range, the assembler may insert a number of instructions to extend the size of the loop. Prefixing the instruction mnemonic with an underscore (`_LOOP`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
LCOUNT ← AR[s] - 1
LBEG ← nextPC
LEND ← PC + (024imm8) + 4
```

Exceptions

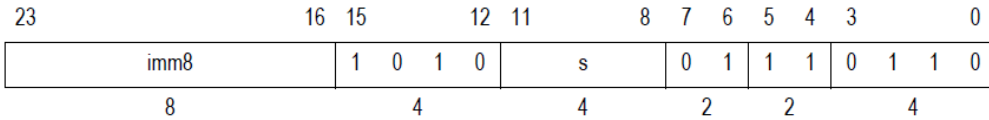
- EveryInstR Group (see [EveryInstR Group](#))

Implementation Notes

In some implementations, `LOOP` takes an extra clock for the first loop back of certain loops. In addition, certain instructions (such as `ISYNC` or a write to `LEND`) may cause an additional cycle on the following loop back.

8.3.150 LOOPGTZ—Loop if Greater Than Zero

Instruction Word (BRI8)



Required Configuration Option

Loop Option (See [Loop Option](#) on page 84)

Assembler Syntax

```
LOOPGTZ as, label
```

Description

`LOOPGTZ` sets up a zero-overhead loop by setting the `LCOUNT`, `LBEG`, and `LEND` special registers, which control instruction fetch. The loop will iterate the number of times specified by address register `as` with values ≤ 0 causing the loop to be skipped altogether by branching directly to the loop end address. `LCOUNT`, the current loop iteration counter, is loaded from the contents of address register `as` minus one. `LEND` is the loop end address and is loaded with the address of the `LOOPGTZ` instruction plus four, plus the zero-extended 8-bit offset encoded in the instruction (therefore, the loop code may be up to 256 bytes in length). `LBEG`, the loop begin address, is loaded with the address of the following instruction. `LCOUNT`, `LEND`, and `LBEG` are still loaded even when the loop is skipped.

After the processor fetches an instruction that increments the `PC` to the value contained in `LEND`, and `LCOUNT` is not zero, it loads the `PC` with the contents of `LBEG` and decrements `LCOUNT`. `LOOPGTZ` is intended to be implemented with help from the instruction fetch engine of the processor, and therefore should not incur a mispredict or taken branch penalty. Branches and jumps to the address contained in `LEND` do not cause a loop back, and therefore may be used to exit the loop prematurely. Similarly, a return from a call instruction as the last instruction of the loop would not trigger loop back; this case should be avoided.

There is no mechanism to proceed to the next iteration of the loop from the middle of the loop. The compiler may insert a branch to a `NOF` placed as the last instruction of the loop to implement this function if required.

Because `LCOUNT`, `LBEG`, and `LEND` are single registers, zero-overhead loops may not be nested. Using conditional branch instructions to implement outer level loops is typically not a performance issue. Because loops cannot be nested, it is usually inappropriate to include a procedure call inside a loop (the callee might itself use a zero-overhead loop).

To simplify the implementation of zero-overhead loops, the `LBEG` address must be such that the first instruction must entirely fit within a naturally aligned four byte region or, if the fetch width is larger than four bytes, a naturally aligned region which is the next power of two equal to or larger than the fetch width. Some implementations require, in addition, that the fetch width is any greater than the naturally aligned power of two region (of four bytes or larger) which is no smaller than that first instruction. When the `LOOP` instruction would not naturally be placed at such an address, the insertion of `NOP` instructions or adjustment of which instructions are 16-bit density instructions is sufficient to give it the required alignment.

The automatic loop-back when the `PC` increments to match `LEND` is disabled when `PS.EXCM` is set. This prevents non-privileged code from affecting the operation of the privileged exception vector code. Dynamic loaders need to avoid mixing new code and old register values as the combination may execute in unexpected ways.

Assembler Note

The assembler automatically aligns the `LOOPGTZ` instruction as required.

When the label is out of range, the assembler may insert a number of instructions to extend the size of the loop. Prefixing the instruction mnemonic with an underscore (`_LOOPGTZ`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
LCOUNT ← AR[s] - 1
LBEG ← nextPC
LEND ← PC + (024imm8) + 4
if AR[s] ≤ 032 then
    nextPC ← PC + (024imm8) + 4
endif
```

Exceptions

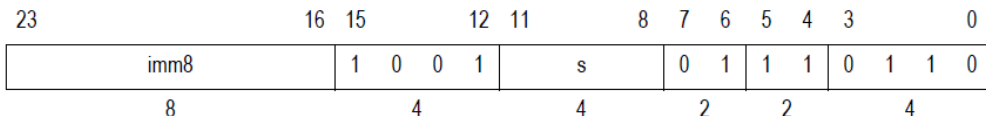
- EveryInstR Group (see [EveryInstR Group](#))

Implementation Notes

In some implementations, `LOOPGTZ` takes an extra clock for the first loop back of certain loops. In addition, certain instructions (such as `ISYNC` or a write to `LEND`) may cause an additional cycle on the following loop back.

8.3.151 LOOPNEZ—Loop if Not-Equal Zero

Instruction Word (BRI8)



Required Configuration Option

Loop Option (See [Loop Option](#) on page 84)

Assembler Syntax

```
LOOPNEZ as, label
```

Description

LOOPNEZ sets up a zero-overhead loop by setting the **LCOUNT**, **LBEG**, and **LEND** special registers, which control instruction fetch. The loop will iterate the number of times specified by address register **as** with the zero value causing the loop to be skipped altogether by branching directly to the loop end address. **LCOUNT**, the current loop iteration counter, is loaded from the contents of address register **as** minus 1. **LEND** is the loop end address and is loaded with the address of the **LOOPNEZ** instruction plus four plus the zero-extended 8-bit offset encoded in the instruction (therefore, the loop code may be up to 256 bytes in length). **LBEG** is loaded with the address of the following instruction. **LCOUNT**, **LEND**, and **LBEG** are still loaded even when the loop is skipped.

After the processor fetches an instruction that increments the **PC** to the value contained in **LEND**, and **LCOUNT** is not zero, it loads the **PC** with the contents of **LBEG** and decrements **LCOUNT**. **LOOPNEZ** is intended to be implemented with help from the instruction fetch engine of the processor, and therefore should not incur a mispredict or taken branch penalty. Branches and jumps to the address contained in **LEND** do not cause a loop back, and therefore may be used to exit the loop prematurely. Similarly a return from a call instruction as the last instruction of the loop would not trigger loop back; this case should be avoided.

There is no mechanism to proceed to the next iteration of the loop from the middle of the loop. The compiler may insert a branch to a **NOF** placed as the last instruction of the loop to implement this function if required.

Because **LCOUNT**, **LBEG**, and **LEND** are single registers, zero-overhead loops may not be nested. Using conditional branch instructions to implement outer level loops is typically not a performance issue. Because loops cannot be nested, it is usually inappropriate to include a procedure call inside a loop (the callee might itself use a zero-overhead loop).

To simplify the implementation of zero-overhead loops, the **LBEG** address must be such that the first instruction must entirely fit within a naturally aligned four byte region or, if the fetch width is larger than four bytes, a naturally aligned region which is the next power of two equal to or larger than the fetch width. Some implementations require, in addition, that the fetch width is any greater than the naturally aligned power of two region (of four bytes or larger)

which is no smaller than that first instruction. When the `LOOP` instruction would not naturally be placed at such an address, the insertion of `NOP` instructions or adjustment of which instructions are 16-bit density instructions is sufficient to give it the required alignment.

The automatic loop-back when the `PC` increments to match `LEND` is disabled when `PS.EXCM` is set. This prevents non-privileged code from affecting the operation of the privileged exception vector code. Dynamic loaders need to avoid mixing new code and old register values as the combination may execute in unexpected ways.

Assembler Note

The assembler automatically aligns the `LOOPNEZ` instruction as required.

When the label is out of range, the assembler may insert a number of instructions to extend the size of the loop. Prefixing the instruction mnemonic with an underscore (`_LOOPNEZ`) disables this feature and forces the assembler to generate an error in this case.

Operation

```
LCOUNT ← AR[s] - 1
LBEG ← nextPC
LEND ← PC + (024imm8) + 4)
if AR[s] = 032 then
    nextPC ← PC + (024imm8) + 4
endif
```

Exceptions

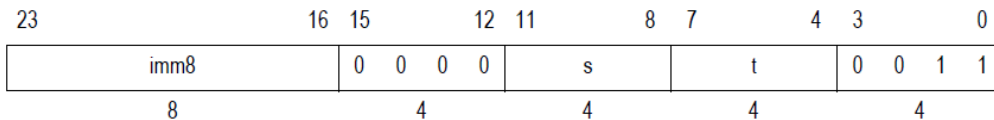
- EveryInstR Group (see [EveryInstR Group](#))

Implementation Notes

In some implementations, `LOOPNEZ` takes an extra clock for the first loop back of certain loops. In addition, certain instructions (such as `ISYNC` or a write to `LEND`) may cause an additional cycle on the following loop back.

8.3.152 LSI—Load Single Immediate

Instruction Word (RRI8)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
LSI ft, as, 0..1020
```

Description

LSI is a 32-bit load from memory to the floating-point register file. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. Thirty-two bits (four bytes) are read from the physical address. This data is then written to floating-point register `ft`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, LSI calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

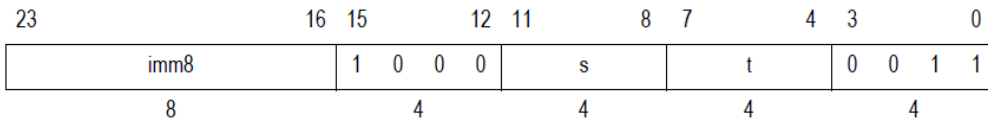
```
vAddr ← AR[s] + (022||imm8||02)
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    FR[t] ← mem32
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.153 LSIP—Load Single Immediate Post-Increment

Instruction Word (RRI8)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
LSIP ft, as, 0..1020
```

Description

LSIP is a 32-bit load from memory to the floating-point register file with base address register post-increment. The virtual address is taken from the contents of address register `as`. Thirty-two bits (four bytes) are read from the physical address. This data is then written to floating-point register `ft` and the virtual address is written back to address register `as`. The sum of the virtual address and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two is written back to address register `as`. The increment can specify multiples of four from zero to 1020

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, LSIP calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

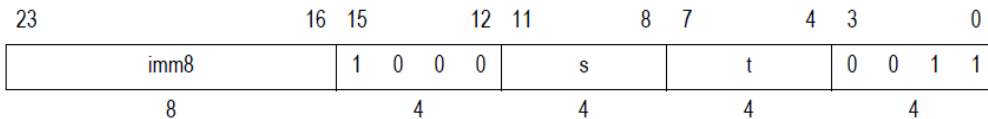
```
vAddr ← AR[s]
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    FR[t] ← mem32
    AS[s] ← vAddr + (022||imm8||02)
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.154 LSIU—Load Single Immediate Update

Instruction Word (RR18)



Required Configuration Option

Floating-Point 2000 Coprocessor Option (See [Floating-Point 2000 Coprocessor Option](#))

Assembler Syntax

```
LSIU ft, as, 0..1020
```

Description

LSIU is a 32-bit load from memory to the floating-point register file with base address register update. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. Thirty-two bits (four bytes) are read from the physical address. This data is then written to floating-point register `ft` and the virtual address is written back to address register `as`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent

memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, LSIU calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

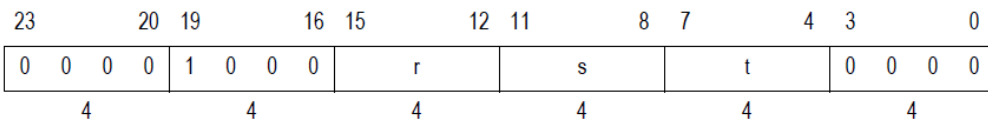
```
vAddr ← AR[s] + (022||imm8||02)
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    FR[t] ← mem32
    AS[s] ← vAddr
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.155 LSX—Load Single Indexed

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
LSX fr, as, at
```

Description

LSX is a 32-bit load from memory to the floating-point register file. It forms a virtual address by adding the contents of address register `as` and the contents of address register `at`. Thirty-two bits (four bytes) are read from the physical address. This data is then written to floating-point register `fr`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```

vAddr ← AR[s] + (AR[t])
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    FR[r] ← mem32
endif

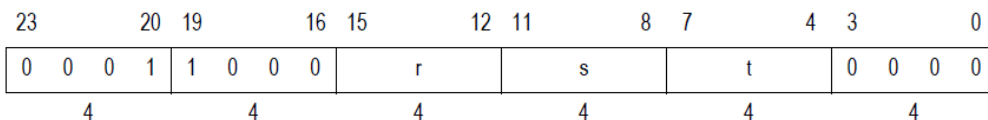
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.156 LSXP—Load Single Indexed Post-Increment

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
LSXP fr, as, at
```

Description

LSXP is a 32-bit load from memory to the floating-point register file with base address register post-increment. The virtual address is taken from the contents of address register `as`. Thirty-two bits (four bytes) are read from the physical address. This data is then written to floating-point register `fr`. The sum of the virtual address and the contents of address register `at` is written back to address register `as`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

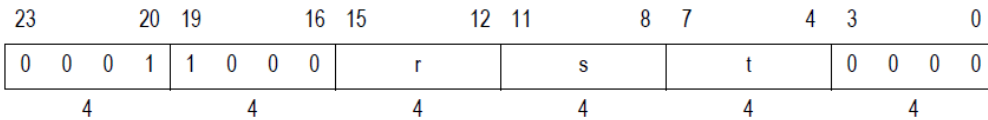
```
vAddr ← AR[s]
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    FR[r] ← mem32
    AR[s] ← vAddr + (AR[t])
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.157 LSXU—Load Single Indexed Update

Instruction Word (RRR)



Required Configuration Option

Floating-Point 2000 Coprocessor Option (See [Floating-Point 2000 Coprocessor Option](#))

Assembler Syntax

```
LSXU fr, as, at
```

Description

LSXU is a 32-bit load from memory to the floating-point register file with base address register update. It forms a virtual address by adding the contents of address register `as` and the contents of address register `at`. Thirty-two bits (four bytes) are read from the physical address. This data is then written to floating-point register `fr` and the virtual address is written back to address register `as`.

If the Region Translation Option ([Unaligned Exception Option](#) on page 148) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

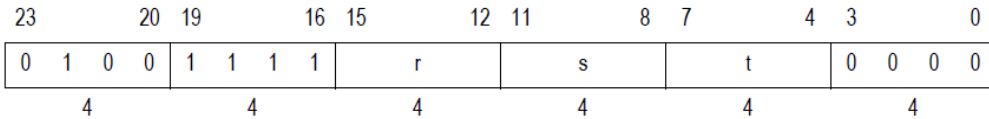
```
vAddr ← AR[s] + (AR[t])
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    FR[r] ← mem32
    AR[s] ← vAddr
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.158 MADD.D—Multiply and Add Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MADD.D fr, fs, ft
```

Description

Using IEEE754 double-precision arithmetic, `MADD.D` multiplies the contents of floating-point registers `fs` and `ft`, adds the product to the contents of floating-point register `fr`, and then writes the sum back to floating-point register `fr`. The computation is performed with no intermediate round.

Operation

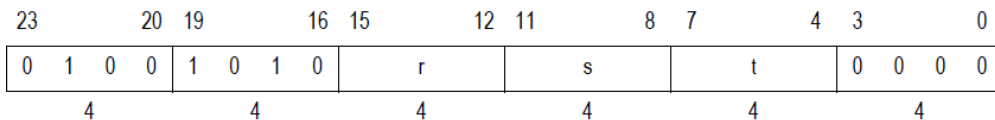
```
FR[r] ← FR[r] +D (FR[s] ×D FR[t]) (×D does not round)  
FSR[StatusFlags: VOUI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.159 MADD.S—Multiply and Add Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MADD.S fr, fs, ft
```

Description

Using IEEE754 single-precision arithmetic, `MADD.S` multiplies the contents of floating-point registers `fs` and `ft`, adds the product to the contents of floating-point register `fr`, and then writes the sum back to floating-point register `fr`. The computation is performed with no intermediate round.

Operation

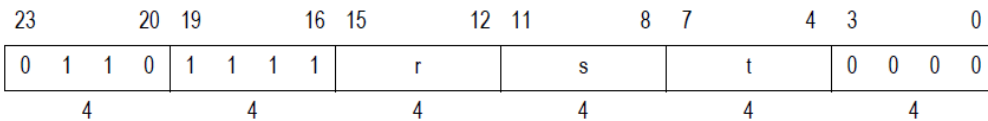
```
FR[r] ← FR[r] +s (FR[s] ×s FR[t]) (×s does not round)  
FSR[StatusFlags: VOUI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.160 MADDN.D—Multiply and Add Double Round Nearest

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MADDN.D fr, fs, ft
```

Description

Using IEEE754 double-precision arithmetic, `MADDN.D` multiplies the contents of floating-point registers `fs` and `ft`, adds the product to the contents of floating-point register `fr`, and then writes the sum back to floating-point register `fr`. The computation is performed with no intermediate round. Unlike the `MADD.D` instruction, this instruction does its final round in round-to-nearest mode regardless of the `FCR` register and sets no flags.

Operation

```
FR[r] ← FR[r] +D (FR[s] ×D FR[t]) (×D does not round, +D round-to-nearest)
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.161 MADDN.S—Multiply and Add Single Round Nearest

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	1	0	1	0	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MADDN.S fr, fs, ft
```

Description

Using IEEE754 single-precision arithmetic, `MADDN.S` multiplies the contents of floating-point registers `fs` and `ft`, adds the product to the contents of floating-point register `fr`, and then writes the sum back to floating-point register `fr`. The computation is performed with no intermediate round. Unlike the `MADD.S` instruction, this instruction does its final round in round-to-nearest mode regardless of the `FCR` register and sets no flags.

Operation

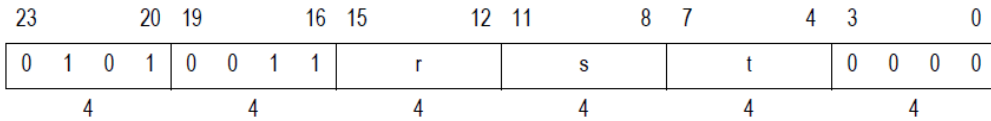
```
FR[r] ← FR[r] +S (FR[s] ×S FR[t]) (×S does not round, +S round-to-nearest)
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.162 MAX—Maximum Value

Instruction Word (RRR)



Required Configuration Option

Miscellaneous Operations Option (See [Miscellaneous Operations Option](#) on page 94)

Assembler Syntax

```
MAX ar, as, at
```

Description

MAX computes the maximum of the two's complement contents of address registers `as` and `at` and writes the result to address register `ar`.

Operation

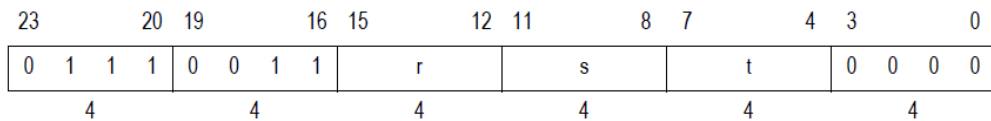
```
AR[r] ← if AR[s] < AR[t] then AR[t] else AR[s]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.163 MAXU—Maximum Value Unsigned

Instruction Word (RRR)



Required Configuration Option

Miscellaneous Operations Option (See [Miscellaneous Operations Option](#) on page 94)

Assembler Syntax

```
MAXU ar, as, at
```

Description

MAXU computes the maximum of the unsigned contents of address registers `as` and `at` and writes the result to address register `ar`.

Operation

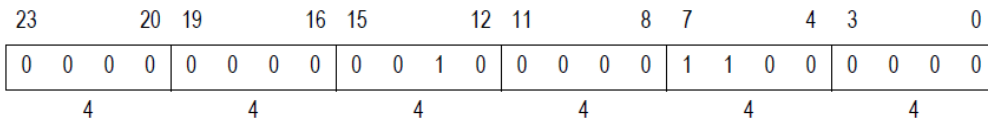
```
AR[r] ← if (0!AR[s]) < (0!AR[t]) then AR[t] else AR[s]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.164 MEMW—Memory Wait

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
MEMW
```

Description

MEMW ensures that all previous load, store, acquire, release, prefetch, and cache instructions along with any writebacks caused by previous cache instructions perform before performing any subsequent load, store, acquire, release, prefetch, or cache instructions. MEMW is intended to implement the `volatile` attribute of languages such as C and C++. The compiler should separate all `volatile` loads and stores with a MEMW instruction. ISYNC should be used to cause instruction fetches to wait as MEMW will have no effect on them.

On processor/system implementations that always reference memory in program order, MEMW may be a no-op. Implementations that reorder load, store, or cache instructions, or which perform merging of stores (for example, in a write buffer) must order such memory references so that all memory references executed before MEMW are performed before any memory references that are executed after MEMW.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation's `memw` function.

Operation

```
memw()
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.165 MIN—Minimum Value

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	0	0	0	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Miscellaneous Operations Option (See [Miscellaneous Operations Option](#) on page 94)

Assembler Syntax

```
MIN ar, as, at
```

Description

MIN computes the minimum of the two's complement contents of address registers `as` and `at` and writes the result to address register `ar`.

Operation

```
AR[r] ← if AR[s] < AR[t] then AR[s] else AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.166 MINU—Minimum Value Unsigned

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	1	0	0	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Miscellaneous Operations Option (See [Miscellaneous Operations Option](#) on page 94)

Assembler Syntax

```
MINU ar, as, at
```

Description

MINU computes the minimum of the unsigned contents of address registers `as` and `at`, and writes the result to address register `ar`.

Operation

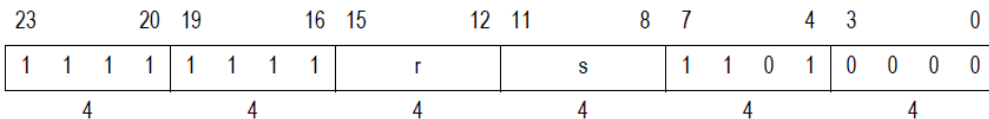
```
AR[r] ← if (0!AR[s]) < (0!AR[t]) then AR[s] else AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.167 MKDADJ.D—Make Divide Adjust Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MKDADJ.D fr, fs
```

Description

MKDADJ.D uses the double-precision values in floating-point registers `fs` and `fr` to create a pair of sign and exponent combinations specifically designed for an IEEE divide sequence. One sign and exponent combination is placed in the sign and exponent of `fr` while the other is placed in the upper mantissa of `fr`. These combinations are consumed by later `ADDEXP.D` and `ADDEXPM.D` instructions from which the combinations propagate to a `DIVN.D` instruction which does the necessary adjustments to a divide sequence result. This instruction is not

intended for use anywhere but in a divide sequence. For more on the divide sequence (see [Divide and Square Root Sequences](#) on page 110).

Operation

```
FR[r] ← divide_adjust(FR[s],FR[r])
FSR[StatusFlags: VZ] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.168 MKDADJ.S—Make Divide Adjust Single

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
1	1	1	1	1	0	1	0	r	s	1	1	0	1	0	0	0	0
4				4				4				4					

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MKDADJ.S fr, fs
```

Description

MKDADJ.S uses the single-precision values in floating-point registers *fs* and *fr* to create a pair of sign and exponent combinations specifically designed for an IEEE divide sequence. One sign and exponent combination is placed in the sign and exponent of *fr* while the other is placed in the upper mantissa of *fr*. These combinations are consumed by later `ADDEXP.S` and `ADDEXPM.S` instructions from which the combinations propagate to a `DIVN.S` instruction which does the necessary adjustments to a divide sequence result. This instruction is not intended for use anywhere but in a divide sequence. For more on the divide sequence (see [Divide and Square Root Sequences](#) on page 110).

Operation

```
FR[r] ← divide_adjust(FR[s],FR[r])
FSR[StatusFlags: VZ] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.169 MKSADJ.D—Make Square Root Adjust Double

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0
1	1	1	1	1	1	1	1	1	0	0	0
4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MKSADJ.D fr, fs
```

Description

MKSADJ.D uses the double-precision value in floating-point register `fs` to create a pair of sign and exponent combinations specifically designed for an IEEE square root sequence. One sign and exponent combination is placed in the sign and exponent of `fr` while the other is placed in the upper mantissa of `fr`. These combinations are consumed by later `ADDEXP.D` and `ADDEXPM.D` instructions from which the combinations propagate to a `DIVN.D` instruction which does the necessary adjustments to a square root sequence result. This instruction is not intended for use anywhere but in a square root sequence. For more on the square root sequence (see [Divide and Square Root Sequences](#) on page 110).

Operation

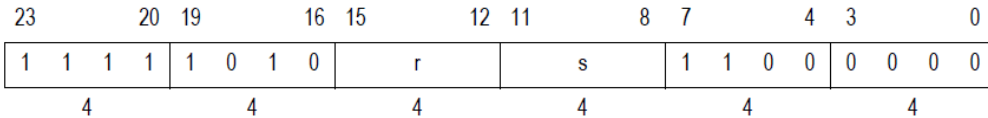
```
FR[r] ← square_root_adjust(FR[s])  
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.170 MKSADJ.S—Make Square Root Adjust Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MKSADJ.S fr, fs
```

Description

MKSADJ.S uses the single-precision value in floating-point register *fs* to create a pair of sign and exponent combinations specifically designed for an IEEE square root sequence. One sign and exponent combination is placed in the sign and exponent of *fr* while the other is placed in the upper mantissa of *fr*. These combinations are consumed by later ADDEXP.S and ADDEXPM.S instructions from which the combinations propagate to a DIVN.S instruction which does the necessary adjustments to a square root sequence result. This instruction is not intended for use anywhere but in a square root sequence. For more on the square root sequence (see [Divide and Square Root Sequences](#) on page 110).

Operation

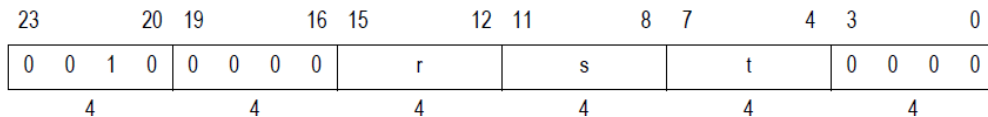
```
FR[r] ← square_root_adjust(FR[s])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.171 MOV—Move

Instruction Word (RRR)



Required Configuration Option

Assembler Macro

Assembler Syntax

```
MOV ar, as
```

Description

MOV is an assembler macro that uses the OR instruction ([Assembler Syntax](#)) to move the contents of address register `as` to address register `ar`. The assembler input

```
MOV ar, as
```

expands into

```
OR ar, as, as
```

Assembler Note

The assembler may convert MOV instructions to MOV.N when the Code Density Option is enabled. Prefixing the MOV instruction with an underscore (`_MOV`) disables this optimization and forces the assembler to generate the OR form of the instruction.

Operation

```
AR[r] ← AR[s]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.172 MOV.D—Move Double

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	1	1	1	1	1	1	1	0	0	0	0			
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MOV.D fr, fs
```

Description

MOV.D moves the contents of floating-point register *fs* to floating-point register *fr*. The move is non-arithmetic; no floating-point exceptions are raised. The function is identical to the MOV.S instruction.

Operation

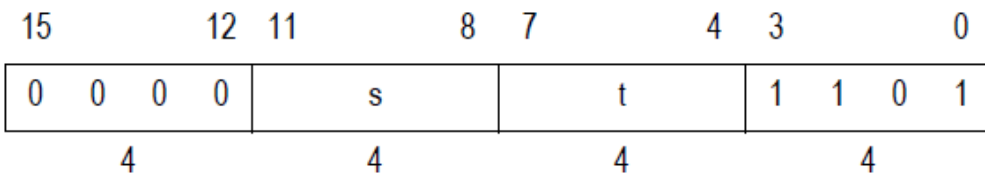
```
FR[r] ← FR[s]
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.173 MOV.N—Narrow Move

Instruction Word (RRR)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

```
MOV.N at, as
```

Description

MOV.N is similar in function to the assembler macro MOV, but has a 16-bit encoding. MOV.N moves the contents of address register *as* to address register *at*.

Assembler Note

The assembler may convert `MOV.N` instructions to `MOV`. Prefixing the `MOV.N` instruction with an underscore (`_MOV.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

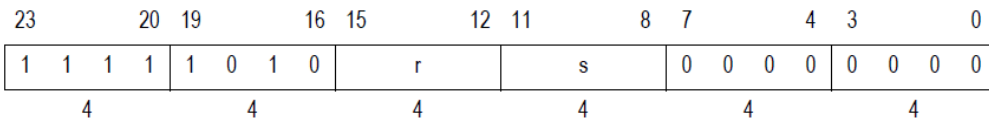
```
AR[t] ← AR[s]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.174 MOV.S—Move Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MOV.S fr, fs
```

Description

`MOV.S` moves the contents of floating-point register `fs` to floating-point register `fr`. The move is non-arithmetic; no floating-point exceptions are raised. The function is identical to the `MOV.D` instruction.

Operation

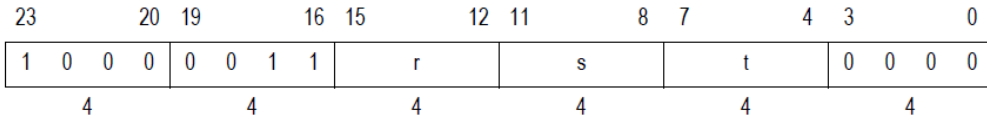
```
FR[r] ← FR[s]
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.175 MOVEQZ—Move if Equal to Zero

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
MOVEQZ ar, as, at
```

Description

MOVEQZ performs a conditional move if equal to zero. If the contents of address register `at` are zero, then the processor sets address register `ar` to the contents of address register `as`. Otherwise, MOVEQZ performs no operation and leaves address register `ar` unchanged.

The inverse of MOVEQZ is MOVNEZ.

Operation

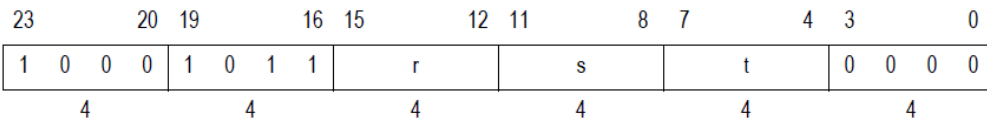
```
if AR[t] = 032 then
    AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.176 MOVEQZ.D—Move Double if Equal to Zero

Instruction Word (RRR)



Required Configuration Option

Assembler Macro

Assembler Syntax

```
MOVEQZ.D fr, fs, at
```

Description

`MOVEQZ.D` is an assembler macro that uses the `MOVEQZ.S` instruction ([Assembler Syntax](#)) to move the contents of floating-point register `fs` to floating-point register `fr`, if address register `at` contains zero. The assembler input

```
MOVEQZ.D fr, fs, at
```

expands into

```
MOVEQZ.S fr, fs, at
```

`MOVEQZ.D` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVEQZ.D` is `MOVNEZ.D`.

Operation

```
if AR[t] = 032 then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(Coprocessor0Disabled) if Coprocessor Context Option

8.3.177 MOVEQZ.S—Move Single if Equal to Zero

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	0	0	1	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MOVEQZ.S fr, fs, at
```

Description

`MOVEQZ.S` is a conditional move between floating-point registers based on the value in an address register. If address register `at` contains zero, the contents of floating-point register `fs` are written to floating-point register `fr`. `MOVEQZ.S` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVEQZ.S` is `MOVNEZ.S`.

Operation

```
if AR[t] = 032 then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.178 MOVF—Move if False

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	0	0	0	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
MOVF ar, as, bt
```

Description

`MOVF` moves the contents of address register `as` to address register `ar` if Boolean register `bt` is false. Address register `ar` is left unchanged if Boolean register `bt` is true.

The inverse of `MOVF` is `MOVT`.

Operation

```
if not BRt then
    AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.179 MOVF.D—Move Double if False

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	0	0	1	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Assembler Macro

Assembler Syntax

```
MOVF.D fr, fs, bt
```

Description

MOVF.D is an assembler macro that uses the MOVF.S instruction ([Assembler Syntax](#)) to move the contents of floating-point register `fs` to floating-point register `fr`, if Boolean register `bt` contains zero. The assembler input

```
MOVF.D fr, fs, bt
```

expands into

```
MOVF.S fr, fs, bt
```

MOVF.D is non-arithmetic; no floating-point exceptions are raised.

The inverse of MOVF.D is MOVTF.D.

Operation

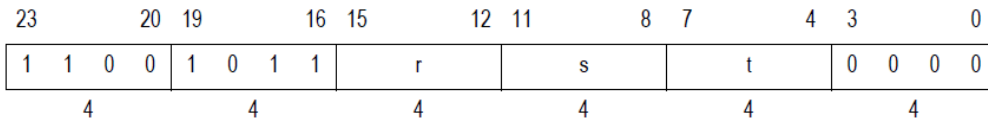
```
if not BRt then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.180 MOVF.S—Move Single if False

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MOVF.S fr, fs, bt
```

Description

MOVF.S is a conditional move between floating-point registers based on the value in a Boolean register. If Boolean register *bt* contains zero, the contents of floating-point register *fs* are written to floating-point register *fr*. MOVF.S is non-arithmetic; no floating-point exceptions are raised.

The inverse of MOVF.S is MOV_T.S.

Operation

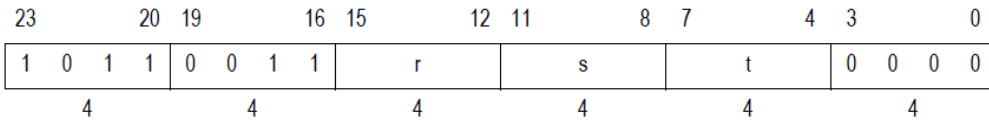
```
if not BRt then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.181 MOVGEZ—Move if Greater Than or Equal to Zero

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
MOVGEZ ar, as, at
```

Description

MOVGEZ performs a conditional move if greater than or equal to zero. If the contents of address register `at` are greater than or equal to zero (that is, the most significant bit is clear), then the processor sets address register `ar` to the contents of address register `as`. Otherwise, MOVGEZ performs no operation and leaves address register `ar` unchanged.

The inverse of MOVGEZ is MOVLTZ.

Operation

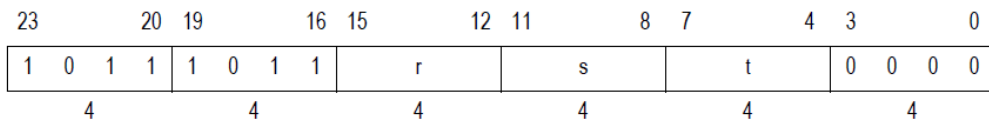
```
if AR[t]31 = 0 then
    AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.182 MOVGEZ.D—Move Double if Greater Than or Eq Zero

Instruction Word (RRR)



Required Configuration Option

Assembler Macro

Assembler Syntax

```
MOVGEZ.D fr, fs, at
```

Description

`MOVGEZ.D` is an assembler macro that uses the `MOVGEZ.S` instruction ([Assembler Syntax](#)) to move the contents of floating-point register `fs` to floating-point register `fr`, if address register `at` is greater than or equal to zero (that is, the most significant bit is clear). The assembler input

```
MOVGEZ.D fr, fs, at
```

expands into

```
MOVGEZ.S fr, fs, at
```

`MOVGEZ.D` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVGEZ.D` is `MOVLTZ.D`.

Operation

```
if AR[t]31 = 0 then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.183 MOVGEZ.S—Move Single if Greater Than or Eq Zero

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	1	1	1	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MOVGEZ.S fr, fs, at
```

Description

`MOVGEZ.S` is a conditional move between floating-point registers based on the value in an address register. If the contents of address register `at` is greater than or equal to zero (that is, the most significant bit is clear), the contents of floating-point register `fs` are written to floating-point register `fr`. `MOVGEZ.S` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVGEZ.S` is `MOVLTZ.S`.

Operation

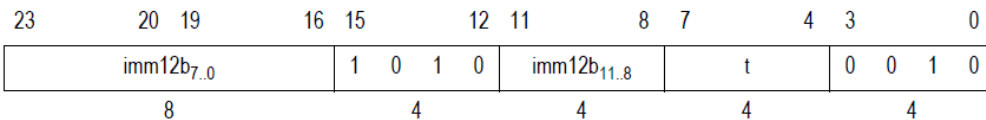
```
if AR[t]31 = 0 then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.184 MOVI—Move Immediate

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
MOVI at, -2048..2047
```

Description

`MOVI` sets address register `at` to a constant in the range `-2048..2047` encoded in the instruction word. The constant is stored in two non-contiguous fields of the instruction word.

The processor decodes the constant specification by concatenating the two fields and sign-extending the 12-bit value.

Assembler Note

The assembler will convert `MOVI` instructions into a literal load when given an immediate operand that evaluates to a value outside the range $-2048..2047$. The assembler will convert `MOVI` instructions to `MOVI.N` when the Code Density Option is enabled and the immediate operand falls within the available range. Prefixing the `MOVI` instruction with an underscore (`_MOVI`) disables these features and forces the assembler to generate an error for the first case and the wide form of the instruction for the second case.

Operation

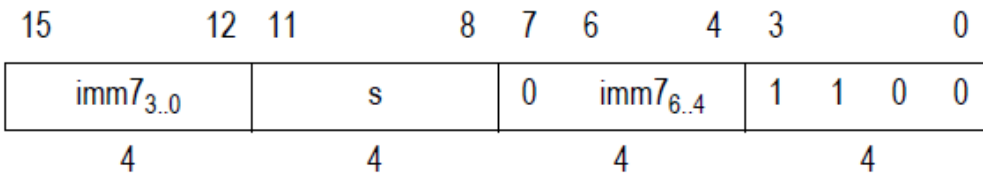
```
AR[t] ← imm121120imm12
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.185 MOVI.N—Narrow Move Immediate

Instruction Word (RI7)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

```
MOVI.N as, -32..95
```

Description

`MOVI.N` is similar to `MOVI`, but has a 16-bit encoding and supports a smaller range of constant values encoded in the instruction word.

`MOVI.N` sets address register `as` to a constant in the range $-32..95$ encoded in the instruction word. The constant is stored in two non-contiguous fields of the instruction word. The range is asymmetric around zero because positive constants are more frequent than negative constants. The processor decodes the constant specification by concatenating the

two fields and sign-extending the 7-bit value with the logical and of its two most significant bits.

Assembler Note

The assembler may convert `MOVI.N` instructions to `MOVI`. Prefixing the `MOVI.N` instruction with an underscore (`_MOVI.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

```
AR[s] ← (imm76 and imm75)25imm7
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.186 MOVL TZ—Move if Less Than Zero

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	1	0	0	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
MOVL TZ ar, as, at
```

Description

`MOVL TZ` performs a conditional move if less than zero. If the contents of address register `at` are less than zero (that is, the most significant bit is set), then the processor sets address register `ar` to the contents of address register `as`. Otherwise, `MOVL TZ` performs no operation and leaves address register `ar` unchanged.

The inverse of `MOVL TZ` is `MOVGEZ`.

Operation

```
if AR[t]31 ≠ 0 then
```

```

    AR[r] ← AR[s]
endif

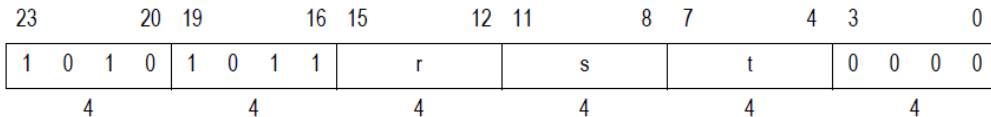
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.187 MOVLZ.D—Move Double if Less Than Zero

Instruction Word (RRR)



Required Configuration Option

Assembler Macro

Assembler Syntax

```

MOVLZ.D fr, fs, at

```

Description

MOVLZ.D is an assembler macro that uses the MOVLZ.S instruction ([Assembler Syntax](#)) to move the contents of floating-point register `fs` to floating-point register `fr`, if address register `at` is less than zero (that is, the most significant bit is set). The assembler input

```

MOVLZ.D fr, fs, at

```

expands into

```

MOVLZ.S fr, fs, at

```

MOVLZ.D is non-arithmetic; no floating-point exceptions are raised.

The inverse of MOVLZ.D is MOVGEZ.D.

Operation

```

if AR[t]31 ≠ 0 then
    FR[r] ← FR[s]
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.188 MOVLTZ.S—Move Single if Less Than Zero

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	1	0	1	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MOVLTZ.S fr, fs, at
```

Description

MOVLTZ.S is a conditional move between floating-point registers based on the value in an address register. If the contents of address register `at` is less than zero (that is, the most significant bit is set), the contents of floating-point register `fs` are written to floating-point register `fr`. MOVLTZ.S is non-arithmetic; no floating-point exceptions are raised.

The inverse of MOVLTZ.S is MOVGEZ.S.

Operation

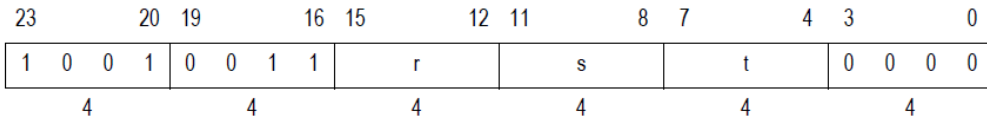
```
if AR[t]31 ≠ 0 then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.189 MOVNEZ—Move if Not-Equal to Zero

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
MOVNEZ ar, as, at
```

Description

MOVNEZ performs a conditional move if not equal to zero. If the contents of address register `at` are non-zero, then the processor sets address register `ar` to the contents of address register `as`. Otherwise, MOVNEZ performs no operation and leaves address register `ar` unchanged.

The inverse of MOVNEZ is MOVEQZ.

Operation

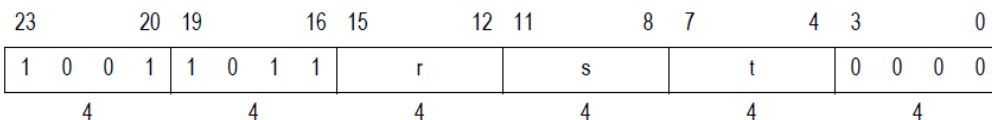
```
if AR[t] ≠ 032 then
    AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.190 MOVNEZ.D—Move Double if Not Equal to Zero

Instruction Word (RRR)



Required Configuration Option

Assembler Macro

Assembler Syntax

```
MOVNEZ.D fr, fs, at
```

Description

`MOVNEZ.D` is an assembler macro that uses the `MOVNEZ.S` instruction ([Assembler Syntax](#)) to move the contents of floating-point register `fs` to floating-point register `fr`, if the contents of address register `at` is non-zero. The assembler input

```
MOVNEZ.D fr, fs, at
```

expands into

```
MOVNEZ.S fr, fs, at
```

`MOVNEZ.D` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVNEZ.D` is `MOVEQZ.D`.

Operation

```
if AR[t] ≠ 032 then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.191 MOVNEZ.S—Move Single if Not Equal to Zero

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	0	1	1	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MOVNEZ.S fr, fs, at
```

Description

`MOVNEZ.S` is a conditional move between floating-point registers based on the value in an address register. If the contents of address register `at` is non-zero, the contents of floating-point register `fs` are written to floating-point register `fr`. `MOVNEZ.S` is non-arithmetic; no floating-point exceptions are raised.

The inverse of `MOVNEZ.S` is `MOVEQZ.S`.

Operation

```
if AR[t] ≠ 032 then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.192 MOVSP—Move to Stack Pointer

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	0	0	0	0	0	0	0	1	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
MOVSP at, as
```

Description

`MOVSP` provides an atomic window check and register-to-register move. If the caller's registers are present in the register file, this instruction simply moves the contents of address register

as to address register at. If the caller's registers are not present, MOVSP raises an Alloca exception under the Windowed Register Option.

MOVSP is typically used to perform variable-size stack frame allocation. The Xtensa Windowed Register ABI specifies that some of the caller's registers may be stored just below the callee's stack pointer. When the stack frame is extended, these values may need to be moved. Under the Windowed Register Option this is handled by raising an Alloca exception so that the registers can be moved with interrupts and exceptions disabled. The Xtensa ABI also requires that the caller's return address be in a0 when MOVSP is executed.

Operation

```

if Windowed Register Option & WindowStarttWindowBase-0011..WindowBase-0001 = 03 then
    Exception (AllocaCause)
else
    AR[t] ← AR[s]
endif

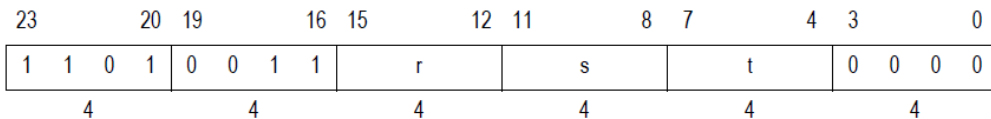
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(AllocaCause) if Windowed Register Option

8.3.193 MOV_T—Move if True

Instruction Word (RRR)



Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
MOVT ar, as, bt
```

Description

MOV_T moves the contents of address register as to address register ar if Boolean register bt is true. Address register ar is left unchanged if Boolean register bt is false.

The inverse of MOV_T is MOV_F.

Operation

```
if BRt then
    AR[r] ← AR[s]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.194 MOV_{T.D}—Move Double if True

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	0	1	1	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Assembler Macro

Assembler Syntax

```
MOVT.D fr, fs, bt
```

Description

MOV_{T.D} is an assembler macro that uses the MOV_{T.S} instruction ([Assembler Syntax](#)) to move the contents of floating-point register `fs` to floating-point register `fr`, if Boolean register `bt` is set. The assembler input

```
MOVT.D fr, fs, bt
```

expands into

```
MOVT.S fr, fs, bt
```

MOV_{T.D} is non-arithmetic; no floating-point exceptions are raised.

The inverse of MOV_{T.D} is MOV_{F.D}.

Operation

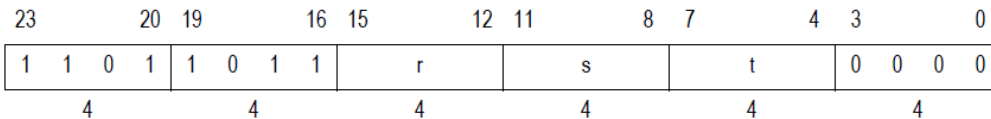
```
if BRt then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.195 MOV_{T.S}—Move Single if True

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MOVT.S fr, fs, bt
```

Description

MOV_{T.S} is a conditional move between floating-point registers based on the value in a Boolean register. If Boolean register *bt* is set, the contents of floating-point register *fs* are written to floating-point register *fr*. MOV_{T.S} is non-arithmetic; no floating-point exceptions are raised.

The inverse of MOV_{T.S} is MOV_{F.S}.

Operation

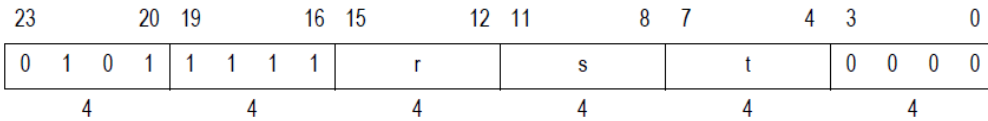
```
if BRt then
    FR[r] ← FR[s]
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.196 MSUB.D—Multiply and Subtract Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MSUB.D fr, fs, ft
```

Description

Using IEEE754 double-precision arithmetic, `MSUB.D` multiplies the contents of floating-point registers `fs` and `ft`, subtracts the product from the contents of floating-point register `fr`, and then writes the difference back to floating-point register `fr`. The computation is performed with no intermediate round.

Operation

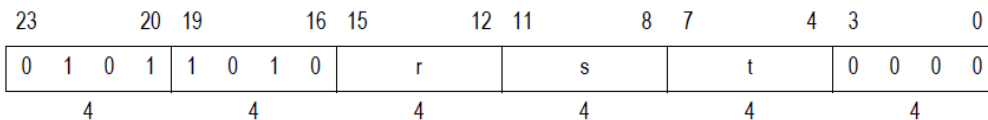
```
FR[r] ← FR[r] -D (FR[s] ×D FR[t]) (×D does not round)  
FSR[StatusFlags: VOUI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.197 MSUB.S—Multiply and Subtract Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MSUB.S fr, fs, ft
```

Description

Using IEEE754 single-precision arithmetic, `MSUB.S` multiplies the contents of floating-point registers `fs` and `ft`, subtracts the product from the contents of floating-point register `fr`, and then writes the difference back to floating-point register `fr`. The computation is performed with no intermediate round.

Operation

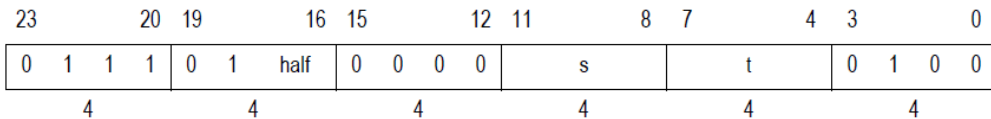
```
FR[r] ← FR[r] -s (FR[s] ×s FR[t]) (×s does not round)  
FSR[StatusFlags: VOUI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.198 MUL.AA.*—Signed Multiply

Instruction Word (RRR)



Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MUL.AA.* as, at
```

Where `*` expands as follows:

```
MUL.AA.LL - for (half=0)  
MUL.AA.HL - for (half=1)  
MUL.AA.LH - for (half=2)  
MUL.AA.HH - for (half=3)
```

Description

`MUL.AA.*` performs a two's complement multiply of half of each of the address registers `as` and `at`, producing a 32-bit result. The result is sign-extended to 40 bits and written to the MAC16 accumulator.

Operation

```
m1 ← if half0 then AR[s]31..16 else AR[s]15..0
m2 ← if half1 then AR[t]31..16 else AR[t]15..0
ACC ← (m11524∥m1) × (m21524∥m2)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.199 MUL.AD.*—Signed Multiply

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0								
0	0	1	1	0	1	half	0	0	0	0	s	0	y	0	0	0	1	0	0
4				4				4				4							

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MUL.AD.* as, my
```

Where `*` expands as follows:

```
MUL.AD.LL - for (half=0)
MUL.AD.HL - for (half=1)
MUL.AD.LH - for (half=2)
MUL.AD.HH - for (half=3)
```

Description

`MUL.AD.*` performs a two's complement multiply of half of address register `as` and half of MAC16 register `my`, producing a 32-bit result. The result is sign-extended to 40 bits and written to the MAC16 accumulator. The `my` operand can designate either MAC16 register `m2` or `m3`.

Operation

```
m1 ← if half0 then AR[s]31..16 else AR[s]15..0
m2 ← if half1 then MR[1|y]31..16 else MR[1|y]15..0
ACC ← (m11524||m1) × (m21524||m2)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.200 MUL.DA.*—Signed Multiply

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	1	0	0	1	half	0	x	0	0	0	0	0	0	0
4				4				4				4			

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MUL.DA.* mx, at
```

Where * expands as follows:

```
MUL.DA.LL - for (half=0)
MUL.DA.HL - for (half=1)
MUL.DA.LH - for (half=2)
MUL.DA.HH - for (half=3)
```

Description

MUL.DA.* performs a two's complement multiply of half of MAC16 register `mx` and half of address register `at`, producing a 32-bit result. The result is sign-extended to 40 bits and written to the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`.

Operation

```
m1 ← if half0 then MR[0|x]31..16 else MR[0|x]15..0
```



```

m2 ← if half1 then AR[t]31..16 else AR[t]15..0
ACC ← (m11524||m1) × (m21524||m2)

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.201 MUL.DD.*—Signed Multiply

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0										
0	0	1	0	0	1	half	0	x	0	0	0	0	0	y	0	0	0	1	0	0	0
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MUL.DD.* mx, my
```

Where * expands as follows:

```

MUL.DD.LL - for (half=0)
MUL.DD.HL - for (half=1)
MUL.DD.LH - for (half=2)
MUL.DD.HH - for (half=3)

```

Description

MUL.DD.* performs a two's complement multiply of half of the MAC16 registers `mx` and `my`, producing a 32-bit result. The result is sign-extended to 40 bits and written to the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`. The `my` operand can designate either MAC16 register `m2` or `m3`.

Operation

```

m1 ← if half0 then MR[0|x]31..16 else MR[0|x]15..0
m2 ← if half1 then MR[1|y]31..16 else MR[1|y]15..0
ACC ← (m11524||m1) × (m21524||m2)

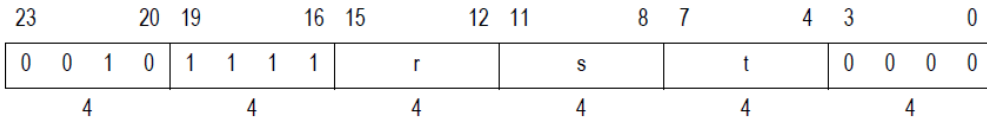
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.202 MUL.D—Multiply Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MUL.D fr, fs, ft
```

Description

MUL.D computes the IEEE754 double-precision product of the contents of floating-point registers *fs* and *ft* and writes the result to floating-point register *fr*.

Operation

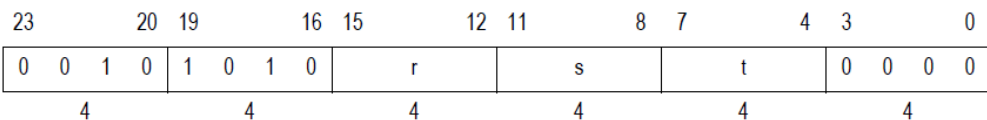
```
FR[r] ← FR[s] ×D FR[t]  
FSR[StatusFlags: VOUI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.203 MUL.S—Multiply Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
MUL.S fr, fs, ft
```

Description

MUL.S computes the IEEE754 single-precision product of the contents of floating-point registers *fs* and *ft* and writes the result to floating-point register *fr*.

Operation

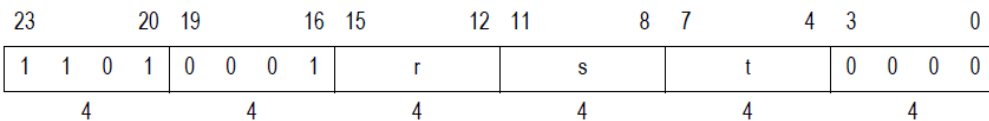
```
FR[r] ← FR[s] ×s FR[t]  
FSR[StatusFlags: VOUI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.204 MUL16S—Multiply 16-bit Signed

Instruction Word (RRR)



Required Configuration Option

16-bit Integer Multiply Option (See [16-bit Integer Multiply Option](#) on page 87)

Assembler Syntax

```
MUL16S ar, as, at
```

Description

MUL16S performs a two's complement multiplication of the least-significant 16 bits of the contents of address registers *as* and *at* and writes the 32-bit product to address register *ar*.

Operation

```
AR[r] ← (AR[s]1516∥AR[s]15..0) × (AR[t]1516∥AR[t]15..0)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.205 MUL16U—Multiply 16-bit Unsigned

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	0	0	0	0	0	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

16-bit Integer Multiply Option (See [16-bit Integer Multiply Option](#) on page 87)

Assembler Syntax

```
MUL16U ar, as, at
```

Description

MUL16U performs an unsigned multiplication of the least-significant 16 bits of the contents of address registers `as` and `at` and writes the 32-bit product to address register `ar`.

Operation

```
AR[r] ← (016 | AR[s]15..0) × (016 | AR[t]15..0)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.206 MULA.AA.*—Signed Multiply/Accumulate

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0					
0	1	1	1	1	0	half	0	0	0	0	s	t	0	1	0	0
4				4				4				4				

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULA.AA.* as, at
```

Where * expands as follows:

```
MULA.AA.LL - for (half=0)  
MULA.AA.HL - for (half=1)  
MULA.AA.LH - for (half=2)  
MULA.AA.HH - for (half=3)
```

Description

MULA.AA.* performs a two's complement multiply of half of each of the address registers *as* and *at*, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator.

Operation

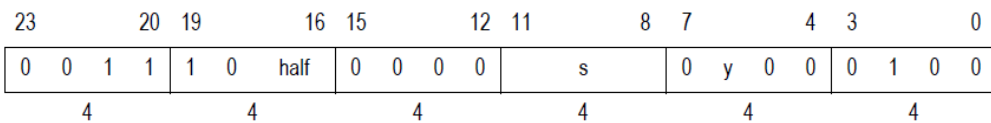
```
m1 ← if half0 then AR[s]31..16 else AR[s]15..0  
m2 ← if half1 then AR[t]31..16 else AR[t]15..0  
ACC ← ACC + (m115224∥m1) × (m215224∥m2)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.207 MULA.AD.*—Signed Multiply/Accumulate

Instruction Word (RRR)



Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULA.AD.* as, my
```

Where * expands as follows:

```
MULA.AD.LL - for (half=0)
MULA.AD.HL - for (half=1)
MULA.AD.LH - for (half=2)
MULA.AD.HH - for (half=3)
```

Description

MULA.AD.* performs a two's complement multiply of half of address register as and half of MAC16 register my , producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The my operand can designate either MAC16 register $m2$ or $m3$.

Operation

```
 $m1 \leftarrow$  if  $half_0$  then  $AR[s]_{31..16}$  else  $AR[s]_{15..0}$ 
 $m2 \leftarrow$  if  $half_1$  then  $MR[1|y]_{31..16}$  else  $MR[1|y]_{15..0}$ 
 $ACC \leftarrow ACC + (m1_{15}^{24} \# m1) \times (m2_{15}^{24} \# m2)$ 
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.208 MULA.DA.*—Signed Multiply/Accumulate

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0					
0	1	1	0	1	0	half	0	x	0	0	0	0	0	0	0	0
4				4				4				4				

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULA.DA.* mx, at
```

Where * expands as follows:

```
MULA.DA.LL - for (half=0)
MULA.DA.HL - for (half=1)
```

```
MULA.DA.LH - for (half=2)
MULA.DA.HH - for (half=3)
```

Description

MULA.DA.* performs a two's complement multiply of half of MAC16 register `mx` and half of address register `at`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`.

Operation

```
m1 ← if half0 then MR[0!x]31..16 else MR[0!x]15..0
m2 ← if half1 then AR[t]31..16 else AR[t]15..0
ACC ← ACC + (m11524!m1) × (m21524!m2)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.209 MULA.DA.*.LDDEC—Signed Mult/Accum, Ld/Autodec

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	0	1	1	0	half	0	x	w	s	t	0	1	0	0
4				4				4				4			

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULA.DA.*.LDDEC mw, as, mx, at
```

Where * expands as follows:

```
MULA.DA.LL.LDDEC - for (half=0)
MULA.DA.HL.LDDEC - for (half=1)
MULA.DA.LH.LDDEC - for (half=2)
MULA.DA.HH.LDDEC - for (half=3)
```

Description

MULA.DA.*.LDDEC performs a parallel load and multiply/accumulate.

First, it performs a two's complement multiply of half of MAC16 register m_x and half of address register a_t , producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The m_x operand can designate either MAC16 register m_0 or m_1 .

Next, it loads MAC16 register m_w from memory using auto-decrement addressing. It forms a virtual address by subtracting 4 from the contents of address register a_s . Thirty-two bits (four bytes) are read from the physical address. This data is then written to MAC16 register m_w , and the virtual address is written back to address register a_s . The m_w operand can designate any of the four MAC16 registers.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

The MAC16 register source m_x and the MAC16 register destination m_w may be the same. In this case, the instruction uses the contents of m_x as the source operand prior to loading m_x with the load data.

Operation

```

vAddr ← AR[s] - 4
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    m1 ← if half0 then MR[0|x]31..16 else MR[0|x]15..0
    m2 ← if half1 then AR[t]31..16 else AR[t]15..0
    ACC ← ACC + (m11524 | m1) × (m21524 | m2)
    AR[s] ← vAddr
    MR[w] ← mem32
endif

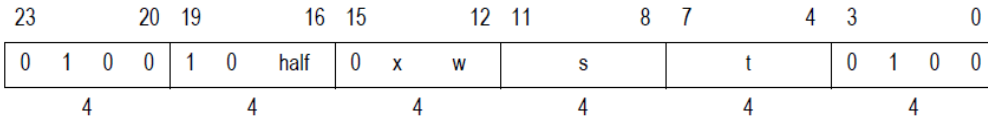
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.210 MULA.DA.*LDINC—Signed Mult/Accum, Ld/Autoinc

Instruction Word (RRR)



Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULA.DA.*.LDINC mw, as, mx, at
```

Where * expands as follows:

```
MULA.DA.LL.LDINC - for (half=0)
MULA.DA.HL.LDINC - for (half=1)
MULA.DA.LH.LDINC - for (half=2)
MULA.DA.HH.LDINC - for (half=3)
```

Description

`MULA.DA.*.LDINC` performs a parallel load and multiply/accumulate.

First, it performs a two's complement multiply of half of MAC16 register `mx` and half of address register `at`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`.

Next, it loads MAC16 register `mw` from memory using auto-increment addressing. It forms a virtual address by adding 4 to the contents of address register `as`. 32 bits (four bytes) are read from the physical address. This data is then written to MAC16 register `mw`, and the virtual address is written back to address register `as`. The `mw` operand can designate any of the four MAC16 registers.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

The MAC16 register source m_x and the MAC16 register destination m_w may be the same. In this case, the instruction uses the contents of m_x as the source operand prior to loading m_x with the load data.

Operation

```

vAddr ← AR[s] + 4
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    m1 ← if half0 then MR[0!x]31..16 else MR[0!x]15..0
    m2 ← if half1 then AR[t]31..16 else AR[t]15..0
    ACC ← ACC + (m11524!m1) × (m21524!m2)
    AR[s] ← vAddr
    MR[w] ← mem32
endif

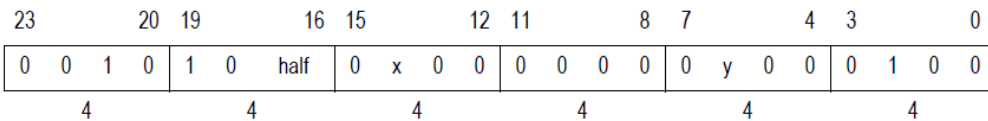
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.211 MULA.DD.*—Signed Multiply/Accumulate

Instruction Word (RRR)



Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULA.DD.* mx, my
```

Where * expands as follows:

```

MULA.DD.LL - for (half=0)
MULA.DD.HL - for (half=1)
MULA.DD.LH - for (half=2)
MULA.DD.HH - for (half=3)

```

Description

`MULA.DD.*` performs a two's complement multiply of half of each of the MAC16 registers `mx` and `my`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`. The `my` operand can designate either MAC16 register `m2` or `m3`.

Operation

```

m1 ← if half0 then MR[0|x]31..16 else MR[0|x]15..0
m2 ← if half1 then MR[1|y]31..16 else MR[1|y]15..0
ACC ← ACC + (m11524|m1) × (m21524|m2)

```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.212 MULA.DD.*.LDDEC—Signed Mult/Accum, Ld/Autodec

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0							
0	0	0	1	1	0	half	0	x	w	s	0	y	0	0	0	1	0	0
4				4				4				4						

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULA.DD.*.LDDEC mw, as, mx, my
```

Where `*` expands as follows:

```

MULA.DD.LL.LDDEC - for (half=0)
MULA.DD.HL.LDDEC - for (half=1)
MULA.DD.LH.LDDEC - for (half=2)
MULA.DD.HH.LDDEC - for (half=3)

```

Description

`MULA.DD.*.LDDEC` performs a parallel load and multiply/accumulate.

First, it performs a two's complement multiply of half of the MAC16 registers `mx` and `my`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`. The `my` operand can designate either MAC16 register `m2` or `m3`.

Next, it loads MAC16 register mw from memory using auto-decrement addressing. It forms a virtual address by subtracting 4 from the contents of address register as . Thirty-two bits (four bytes) are read from the physical address. This data is then written to MAC16 register mw , and the virtual address is written back to address register as . The mw operand can designate any of the four MAC16 registers.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

The MAC16 register destination mw may be the same as either MAC16 register source mx or my . In this case, the instruction uses the contents of mx and my as the source operands prior to loading mw with the load data.

Operation

```

vAddr ← AR[s] - 4
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    m1 ← if half0 then MR[0!x]31..16 else MR[0!x]15..0
    m2 ← if half1 then MR[1!y]31..16 else MR[1!y]15..0
    ACC ← ACC + (m11524!m1) × (m21524!m2)
    AR[s] ← vAddr
    MR[w] ← mem32
endif

```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.213 MULA.DD.*LDINC—Signed Mult/Accum, Ld/Autoinc

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0								
0	0	0	0	1	0	half	0	x	w	s	0	y	0	0	0	1	0	0	0
4				4				4				4							

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULA.DD.*.LDINC mw, as, mx, my
```

Where * expands as follows:

```
MULA.DD.LL.LDINC - for (half=0)
MULA.DD.HL.LDINC - for (half=1)
MULA.DD.LH.LDINC - for (half=2)
MULA.DD.HH.LDINC - for (half=3)
```

Description

`MULA.DD.*.LDINC` performs a parallel load and multiply/accumulate.

First, it performs a two's complement multiply of half of each of the MAC16 registers `mx` and `my`, producing a 32-bit result. The result is sign-extended to 40 bits and added to the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`. The `my` operand can designate either MAC16 register `m2` or `m3`.

Next, it loads MAC16 register `mw` from memory using auto-increment addressing. It forms a virtual address by adding 4 to the contents of address register `as`. Thirty-two bits (four bytes) are read from the physical address. This data is then written to MAC16 register `mw`, and the virtual address is written back to address register `as`. The `mw` operand can designate any of the four MAC16 registers.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

The MAC16 register destination `mw` may be the same as either MAC16 register source `mx` or `my`. In this case, the instruction uses the contents of `mx` and `my` as the source operands prior to loading `mw` with the load data.

Operation

```
vAddr ← AR[s] + 4
(mem32, error) ← Load32(vAddr)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreErrorCause)
else
    m1 ← if half0 then MR[0|x]31..16 else MR[0|x]15..0
    m2 ← if half1 then MR[1|y]31..16 else MR[1|y]15..0
    ACC ← ACC + (m11524|m1) × (m21524|m2)
    AR[s] ← vAddr
    MR[w] ← mem32
endif
```

Exceptions

- Memory Load Group (see [Memory Load Group](#))

8.3.214 MULL—Multiply Low

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	0	0	0	0	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

32-bit Integer Multiply Option (See [32-bit Integer Multiply Option](#) on page 88)

Assembler Syntax

```
MULL ar, as, at
```

Description

MULL performs a 32-bit multiplication of the contents of address registers `as` and `at`, and writes the least significant 32 bits of the product to address register `ar`. Because the least significant product bits are unaffected by the multiplicand and multiplier sign, MULL is useful for both signed and unsigned multiplication.

Operation

```
AR[r] ← AR[s] × AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.215 MULS.AA.*—Signed Multiply/Subtract

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0					
0	1	1	1	1	1	half	0	0	0	0	s	t	0	1	0	0
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULS.AA.* as, at
```

Where * expands as follows:

```
MULS.AA.LL - for (half=0)
MULS.AA.HL - for (half=1)
MULS.AA.LH - for (half=2)
MULS.AA.HH - for (half=3)
```

Description

MULS.AA.* performs a two's complement multiply of half of each of the address registers `as` and `at`, producing a 32-bit result. The result is sign-extended to 40 bits and subtracted from the contents of the MAC16 accumulator.

Operation

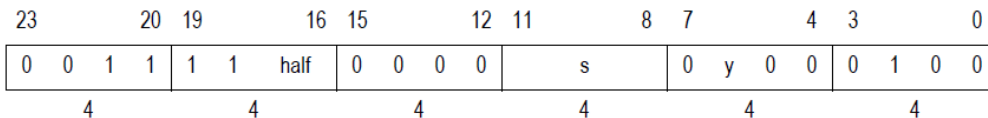
```
m1 ← if half0 then AR[s]31..16 else AR[s]15..0
m2 ← if half1 then AR[t]31..16 else AR[t]15..0
ACC ← ACC - (m11524∥m1) × (m21524∥m2)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.216 MULS.AD.*—Signed Multiply/Subtract

Instruction Word (RRR)



Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULS.AD.* as, my
```

Where * expands as follows:

```
MULS.AD.LL - for (half=0)
MULS.AD.HL - for (half=1)
MULS.AD.LH - for (half=2)
MULS.AD.HH - for (half=3)
```

Description

`MULS.AD.*` performs a two's complement multiply of half of address register `as` and half of MAC16 register `my`, producing a 32-bit result. The result is sign-extended to 40 bits and subtracted from the contents of the MAC16 accumulator. The `my` operand can designate either MAC16 register `m2` or `m3`.

Operation

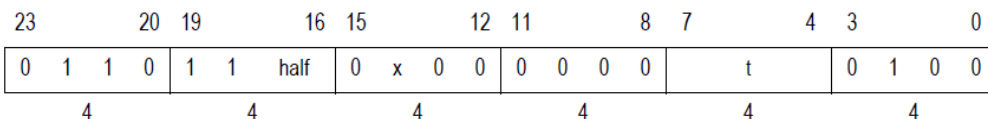
```
m1 ← if half0 then AR[s]31..16 else AR[s]15..0
m2 ← if half1 then MR[1|y]31..16 else MR[1|y]15..0
ACC ← ACC - (m115224∥m1) × (m215224∥m2)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.217 MULS.DA.*—Signed Multiply/Subtract

Instruction Word (RRR)



Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULS.DA.* mx, at
```

Where * expands as follows:

```
MULS.DA.LL - for (half=0)
MULS.DA.HL - for (half=1)
MULS.DA.LH - for (half=2)
MULS.DA.HH - for (half=3)
```

Description

`MULS.DA.*` performs a two's complement multiply of half of MAC16 register `mx` and half of address register `at`, producing a 32-bit result. The result is sign-extended to 40 bits and subtracted from the contents of the MAC16 accumulator. The `mx` operand can designate either MAC16 register `m0` or `m1`.

Operation

```
m1 ← if half0 then MR[0!x]31..16 else MR[0!x]15..0
m2 ← if half1 then AR[t]31..16 else AR[t]15..0
ACC ← ACC - (m115224!m1) × (m215224!m2)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.218 MULS.DD.*—Signed Multiply/Subtract

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0									
0	0	1	0	1	1	half	0	x	0	0	0	0	0	0	0	0	0	0	0	0
4				4				4				4								

Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
MULS.DD.* mx, my
```

Where * expands as follows:

```
MULS.DD.LL - for (half=0)  
MULS.DD.HL - for (half=1)  
MULS.DD.LH - for (half=2)  
MULS.DD.HH - for (half=3)
```

Description

MULS.DD.* performs a two's complement multiply of half of each of MAC16 registers m_x and m_y , producing a 32-bit result. The result is sign-extended to 40 bits and subtracted from the contents of the MAC16 accumulator. The m_x operand can designate either MAC16 register m_0 or m_1 . The m_y operand can designate either MAC16 register m_2 or m_3 .

Operation

```
m1 ← if half0 then MR[0|x]31..16 else MR[0|x]15..0  
m2 ← if half1 then MR[1|y]31..16 else MR[1|y]15..0  
ACC ← ACC - (m11524∥m1) × (m21524∥m2)
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.219 MULSH—Multiply Signed High

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	1	1	0	0	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

32-bit Integer Multiply Option (See [32-bit Integer Divide Option](#) on page 90)

Assembler Syntax

```
MULSH ar, as, at
```

Description

MULSH performs a 32-bit two's complement multiplication of the contents of address registers *as* and *at* and writes the most significant 32 bits of the product to address register *ar*.

Operation

```

$$tp \leftarrow (AR[s]_{31:32} \mid AR[s]) \times (AR[t]_{31:32} \mid AR[t])$$

$$AR[r] \leftarrow tp_{63..32}$$

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.220 MULUH—Multiply Unsigned High

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	1	0	0	0	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

32-bit Integer Multiply Option (See [32-bit Integer Multiply Option](#) on page 88)

Assembler Syntax

```
MULUH ar, as, at
```

Description

MULUH performs an unsigned multiplication of the contents of address registers *as* and *at*, and writes the most significant 32 bits of the product to address register *ar*.

Operation

```

$$tp \leftarrow (0^{32} \mid AR[s]) \times (0^{32} \mid AR[t])$$

$$AR[r] \leftarrow tp_{63..32}$$

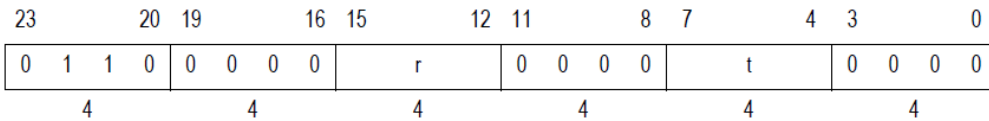
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.221 NEG—Negate

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
NEG ar, at
```

Description

NEG calculates the two's complement negation of the contents of address register `at` and writes it to address register `ar`. Arithmetic overflow is not detected.

Operation

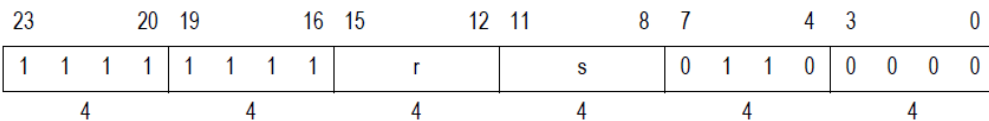
```
AR[r] ← 0 - AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.222 NEG.D—Negate Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
NEG.D fr, fs
```

Description

NEG.D negates the double-precision value of the contents of floating-point register fr_s and writes the result to floating-point register fr_r .

Operation

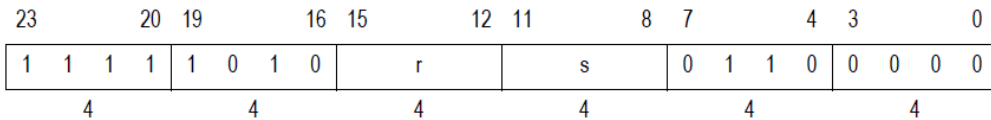
```
FR[r] ← -D FR[s]
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.223 NEG.S—Negate Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
NEG.S fr, fs
```

Description

NEG.S negates the single-precision value of the contents of floating-point register fr_s and writes the result to floating-point register fr_r .

Operation

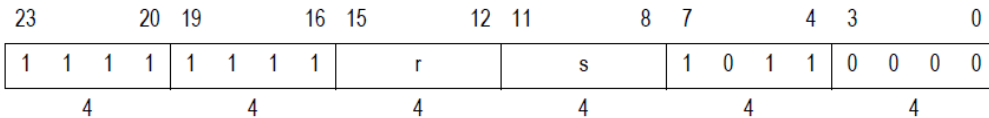
```
FR[r] ← -S FR[s]
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.224 NEXP01.D—Narrow Exponent Range Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
NEXP01.D fr, fs
```

Description

`NEXP01.D` narrows the exponent range of the double-precision number in floating-point register `fs` by multiplying or dividing by a power of 4.0, inverts the sign bit, and places the result in floating-point register `fr`. The power of 4.0 is chosen so that the magnitude of the resulting number is greater than or equal to 1.0 and less than 4.0. Denormal arguments are normalized first. NaN, Infinity, and Zero result in special values.

`NEXP01.D` is used in divide and square root algorithms (see [Divide and Square Root Sequences](#) on page 110) and is not intended for use anywhere else.

Operation

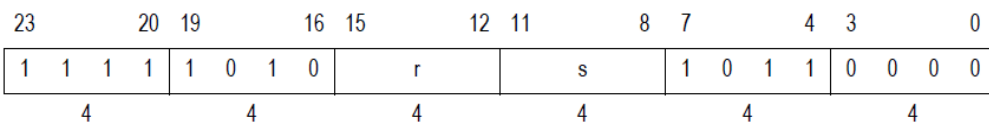
```
if (FR[s]62..52 = 111) then
    FR[r] ← (not FR[s]63)#0#110#FR[s]51..0
elseif (FR[s]62..0 = 063) then
    FR[r] ← (not FR[s]63)#1#062
else
    N ← TRUNC (LOG2 (ABS (NORMALIZE (FR[s]))) ÷D 2.0)
    FR[r] ← -D (FR[s] ÷D POW(4.0,N))
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.225 NEXP01.S—Narrow Exponent Range Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
NEXP01.S fr, fs
```

Description

NEXP01.S narrows the exponent range of the single-precision number in floating-point register `fs` by multiplying or dividing by a power of 4.0, inverts the sign bit, and places the result in floating-point register `fr`. The power of 4.0 is chosen so that the magnitude of the resulting number is greater than or equal to 1.0 and less than 4.0. Denormal arguments are normalized first. NaN, Infinity, and Zero result in special values.

NEXP01.S is used in divide and square root algorithms (see [Divide and Square Root Sequences](#) on page 110) and is not intended for use anywhere else.

Operation

```
if (FR[s]30..23 = 18) then
    FR[r] ← (not FR[s]31) # 0117 # FR[s]22..0
elsif (FR[s]30..0 = 031) then
    FR[r] ← (not FR[s]31) # 11030
else
    N ← TRUNC (LOG2 (ABS (NORMALIZE ((FR[s]))) ÷D 2.0))
    FR[r] ← -D (FR[s] ÷D POW(4.0,N))
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.226 NOP—No-Operation

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	
4				4				4				4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

NOP

Description

This instruction performs no operation. It is typically used for instruction alignment. `NOP` is a 24-bit instruction. For a 16-bit version, see `NOP.N`.

Assembler Note

The assembler may convert `NOP` instructions to `NOP.N` when the Code Density Option is enabled. Prefixing the `NOP` instruction with an underscore (`_NOP`) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

none

Exceptions

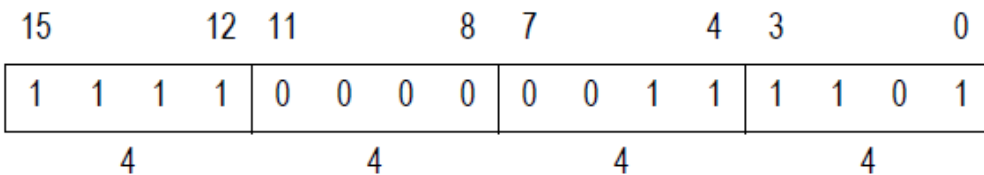
- EveryInst Group (see [EveryInst Group](#))

Implementation Notes

In some implementations `NOP` is not an instruction but only an assembler macro that uses the instruction “OR `An`, `An`, `An`” (with `An` a convenient register).

8.3.227 `NOP.N`—Narrow No-Operation

Instruction Word (RRRN)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

`NOP.N`

Description

This instruction performs no operation. It is typically used for instruction alignment. `NOP.N` is a 16-bit instruction. For a 24-bit version, see `NOP`.

Assembler Note

The assembler may convert `NOP.N` instructions to `NOP`. Prefixing the `NOP.N` instruction with an underscore (`_NOP.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

```
none
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.228 NSA—Normalization Shift Amount

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	1	0	0	0	0	0	0	1	1	1	0	s	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Miscellaneous Operations Option (See [Miscellaneous Operations Option](#) on page 94)

Assembler Syntax

```
NSA at, as
```

Description

`NSA` calculates the left shift amount that will normalize the two's complement contents of address register `as` and writes this amount (in the range 0 to 31) to address register `at`. If `as` contains 0 or -1, `NSA` returns 31. Using `SSL` and `SLL` to shift `as` left by the `NSA` result yields the smallest value for which bits 31 and 30 differ unless `as` contains 0.

Operation

```
sign ← AR[s]31  
if AR[s]30..0 = sign31 then
```

```

    AR[t] ← 31
else
    b4 ← AR[s]30..16 = sign15
    t3 ← if b4 then AR[s]15..0 else AR[s]31..16
    b3 ← t315..8 = sign8
    t2 ← if b3 then t37..0 else t315..8
    b2 ← t27..4 = sign4
    t1 ← if b2 then t23..0 else t27..4
    b1 ← t13..2 = sign2
    b0 ← if b1 then t11 = sign else t13 = sign
    AR[t] ← 027 | ((b4|b3|b2|b1|b0) - 1)
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.229 NSAU—Normalization Shift Amount Unsigned

Instruction Word (RRR)

	23		20	19		16	15		12	11		8	7		4	3		0
	0	1	0	0	0	0	0	1	1	1	1	s	t	0	0	0	0	0
	4		4		4		4		4		4		4		4		4	

Required Configuration Option

Miscellaneous Operations Option (See [Miscellaneous Operations Option](#) on page 94)

Assembler Syntax

```
NSAU at, as
```

Description

NSAU calculates the left shift amount that will normalize the unsigned contents of address register `as` and writes this amount (in the range 0 to 32) to address register `at`. If `as` contains 0, NSAU returns 32. Using `SSL` and `SLL` to shift `as` left by the NSAU result yields the smallest value for which bit 31 is set, unless `as` contains 0.

Operation

```

if AR[s] = 032 then
    AR[t] ← 32
else
    b4 ← AR[s]31..16 = 016
    t3 ← if b4 then AR[s]15..0 else AR[s]31..16
    b3 ← t315..8 = 08
    t2 ← if b3 then t37..0 else t315..8
    b2 ← t27..4 = 04
    t1 ← if b2 then t23..0 else t27..4

```

```

    b1 ← tI3..2 = 02
    b0 ← if b1 then tI1 = 0 else tI3 = 0
    AR[t] ← 027 | b4 | b3 | b2 | b1 | b0
endif

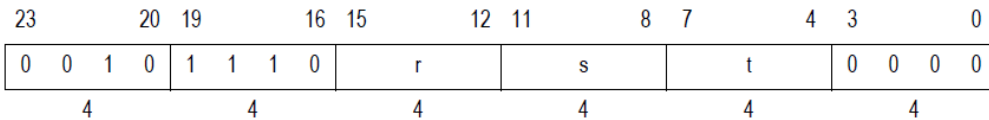
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.230 *OEQ.D*—Compare Double Equal

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
OEQ.D br, fs, ft
```

Description

OEQ.D compares the double-precision values in floating-point registers *fs* and *ft* for IEEE754 equality. If the values are ordered and equal then Boolean register *br* is set to 1, otherwise *br* is set to 0. IEEE754 specifies that +0 and -0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN. Like most floating-point instructions *OEQ.D* sets the Invalid Operation flag if either input is a Signalling NaN.

Operation

```

BRr ← not isNaND(FR[s]) and not isNaND(FR[t])
      and (FR[s] =D FR[t])
FSR[StatusFlags: V] ← Or in update

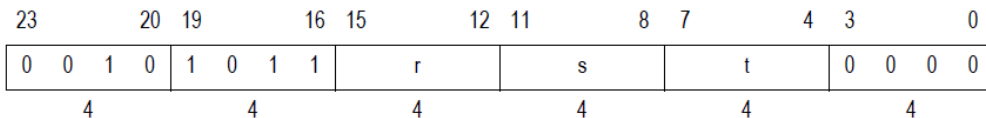
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.231 *OEQ.S*—Compare Single Equal

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
OEQ.S br, fs, ft
```

Description

OEQ.S compares the single-precision values in floating-point registers *fs* and *ft* for IEEE754 equality. If the values are ordered and equal then Boolean register *br* is set to 1, otherwise *br* is set to 0. IEEE754 specifies that +0 and -0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN. Like most floating-point instructions OEQ.S sets the Invalid Operation flag if either input is a Signalling NaN.

Operation

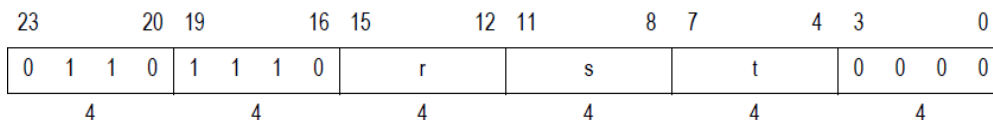
```
BRr ← not isNaNs(FR[s]) and not isNaNs(FR[t])
        and (FR[s] =s FR[t])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.232 OLE.D—Compare Double Ord & Less Than or Equal

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
OLE.D br, fs, ft
```

Description

`OLE.D` compares the double-precision values in floating-point registers `fs` and `ft`. If the contents of `fs` are ordered with, and less than or equal to the contents of `ft`, then Boolean register `br` is set to 1, otherwise `br` is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN. `OLE.D` sets the Invalid Operation flag if either input is a NaN of any kind.

Operation

```
BRr ← not isNaND(FR[s]) and not isNaND(FR[t])  
          and (FR[s] ≤D FR[t])  
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.233 OLE.S—Compare Single Ord & Less Than or Equal

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	1	0	1	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
OLE.S br, fs, ft
```

Description

`OLE.S` compares the single-precision values in floating-point registers `fs` and `ft`. If the contents of `fs` are ordered with, and less than or equal to the contents of `ft`, then Boolean register `br` is set to 1, otherwise `br` is set to 0. According to IEEE754, +0 and -0 compare as

equal. IEEE754 floating-point values are ordered if neither is a NaN. `OLT.S` sets the Invalid Operation flag if either input is a NaN of any kind.

Operation

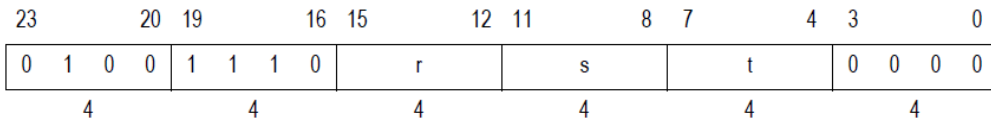
```
BRr ← not isNaNs(FR[s]) and not isNaNs(FR[t])
        and (FR[s] ≤s FR[t])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.234 OLT.D—Compare Double Ordered and Less Than

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
OLT.D br, fs, ft
```

Description

`OLT.D` compares the double-precision values in floating-point registers `fs` and `ft`. If the contents of `fs` are ordered with and less than the contents of `ft` then Boolean register `br` is set to 1, otherwise `br` is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN. `OLT.D` sets the Invalid Operation flag if either input is a NaN of any kind.

Operation

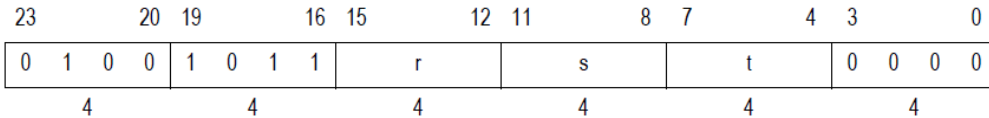
```
BRr ← not isNaND(FR[s]) and not isNaND(FR[t])
        and (FR[s] <D FR[t])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.235 OLT.S—Compare Single Ordered and Less Than

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
OLT.S br, fs, ft
```

Description

OLT.S compares the single-precision values in floating-point registers `fs` and `ft`. If the contents of `fs` are ordered with and less than the contents of `ft` then Boolean register `br` is set to 1, otherwise `br` is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are ordered if neither is a NaN. OLT.S sets the Invalid Operation flag if either input is a NaN of any kind.

Operation

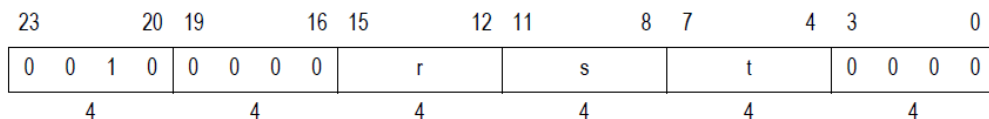
```
BRr ← not isNaNs(FR[s]) and not isNaNs(FR[t])
        and (FR[s] <s FR[t])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.236 OR—Bitwise Logical Or

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
OR ar, as, at
```

Description

OR calculates the bitwise logical or of address registers `as` and `at`. The result is written to address register `ar`.

Operation

```
AR[r] ← AR[s] or AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.237 ORB—Boolean Or

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	0	1	0	0	0	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
ORB br, bs, bt
```

Description

ORB performs the logical or of Boolean registers `bs` and `bt`, and writes the result to Boolean register `br`.

When the sense of one of the source Booleans is inverted (0 → true, 1 → false), use `ORB.C`. When the sense of both of the source Booleans is inverted, use `ORB.D` and an inverted test of the result.

Operation

$$BR_r \leftarrow BR_s \text{ or } BR_t$$

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.238 ORBC—Boolean Or with Complement

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0			
0	0	1	1	0	0	1	0	r	s	t	0	0	0	0
4				4				4		4		4		

Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

$$\text{ORBC } br, bs, bt$$

Description

ORBC performs the logical or of Boolean register `bs` with the logical complement of Boolean register `bt` and writes the result to Boolean register `br`.

Operation

$$BR_r \leftarrow BR_s \text{ or not } BR_t$$

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.239 TLB—PDTLB Probe Data

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	1	0	1	0	0	0	0	1	1	0	1	s	t	0	0	0	0
4				4				4		4		4					

Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
PDTLB at, as
```

Description

`PDTLB` searches the data TLB for an entry that translates the virtual address in address register `as` and writes the way and index of that entry to address register `at`. If no entry matches, zero is written to the hit bit of `at`. The value written to `at` is implementation-specific, but in all implementations a value with the hit bit set is suitable as an input to the `IDTLB` or `WDTLB` instructions. See [Options for Memory Protection and Translation](#) on page 183 for information on the result register formats for specific memory protection and translation options. Even though `CRING` is required to be zero for the instruction to run, `PS.Ring` is used in the TLB lookup so that a probe may check access for lower privileges.

`PDTLB` is a privileged instruction.

Operation

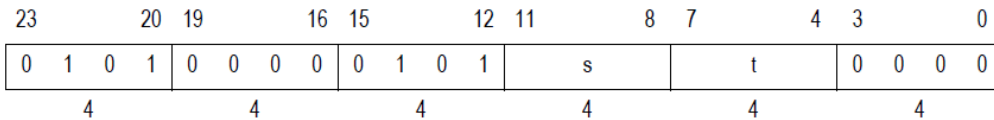
```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    (match, vpn, ei, wi) ← ProbeDataTLB(AR[s], PS.Ring)
    if match > 1 then
        EXCVADDR ← AR[s]
        Exception (LoadStoreTLBMultiHit)
    else
        AR[t] ← PackDataTLBEntrySpec(match, vpn, ei, wi)
    endif
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(LoadStoreTLBMultiHitCause) if Region Protection Option or MMU Option
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.240 PITLB—Probe Instruction TLB

Instruction Word (RRR)



Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
PITLB at, as
```

Description

PITLB searches the Instruction TLB for an entry that translates the virtual address in address register *as* and writes the way and index of that entry to address register *at*. If no entry matches, zero is written to the hit bit of *at*. The value written to *at* is implementation-specific, but in all implementations a value with the hit bit set is suitable as an input to the IITLB or WITLB instructions. See [Options for Memory Protection and Translation](#) on page 183 for information on the result register formats for specific memory protection and translation options. Even though CRING is required to be zero for the instruction to run, PS.Ring is used in the TLB lookup so that a probe may check access for lower privileges as well.

PITLB is a privileged instruction.

Operation

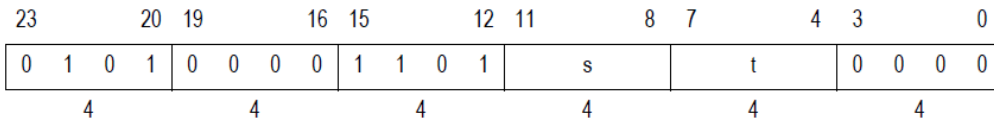
```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    (match, vpn, ei, wi) ← ProbeInstTLB(AR[s],PS.Ring)
    if match > 1 then
        EXCVADDR ← AR[s]
        Exception (InstructionFetchTLBMultiHit)
    else
        AR[t] ← PackInstTLBEntrySpec(match, vpn, ei, wi)
    endif
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.241 PPTLB—Probe Protection TLB

Instruction Word (RRR)



Required Configuration Option

Memory Protection Unit Option ([Memory Protection Unit Option](#) on page 205)

Assembler Syntax

```
PPTLB at, as
```

Description

PPTLB searches the Protection TLB for a Foreground Segment or Background Segment that provides protection information for the virtual address in address register *as* and writes that information and which location it came from to address register *at*. The value written to *at* is implementation-specific. See [Formats for Probing Memory Protection Unit Option TLB Entries](#) on page 213 for information on the result register format.

PPTLB is a privileged instruction.

Operation

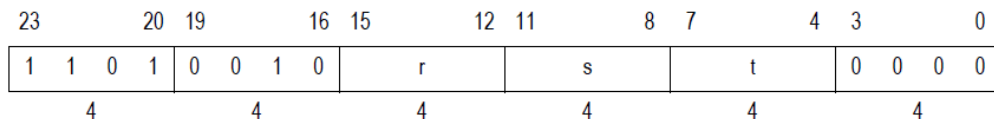
```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    AR[t] ← PackProtectionTLBEntrySpec
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(LoadStoreTLBMultiHitCause) if Region Protection Option or Memory Protection Unit Option or MMU Option
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.242 QUOS—Quotient Signed

Instruction Word (RRR)



Required Configuration Option

32-bit Integer Divide Option (See [32-bit Integer Divide Option](#) on page 90)

Assembler Syntax

```
QUOS ar, as, at
```

Description

QUOS performs a 32-bit two's complement division of the contents of address register *as* by the contents of address register *at* and writes the quotient to address register *ar*. The ambiguity which exists when either address register *as* or address register *at* is negative is resolved by requiring the product of the quotient and address register *at* to be smaller in absolute value than the address register *as*. If the contents of address register *at* are zero, QUOS raises an Integer Divide by Zero exception instead of writing a result. Overflow (-2147483648 divided by -1) is not detected.

Operation

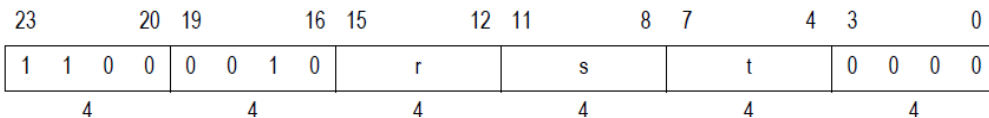
```
if AR[t] = 032 then
    Exception (IntegerDivideByZero)
else
    AR[r] ← AR[s] quo AR[t]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
-
- GenExcep(IntegerDivideByZeroCause) if 32-bit Integer Divide Option

8.3.243 QUOU—Quotient Unsigned

Instruction Word (RRR)



Required Configuration Option

32-bit Integer Divide Option (See [32-bit Integer Divide Option](#) on page 90)

Assembler Syntax

```
QUOU ar, as, at
```

Description

QUOU performs a 32-bit unsigned division of the contents of address register `as` by the contents of address register `at` and writes the quotient to address register `ar`. If the contents of address register `at` are zero, QUOU raises an Integer Divide by Zero exception instead of writing a result.

Operation

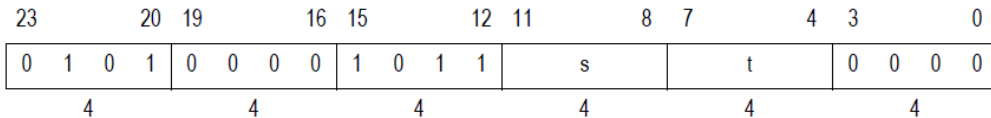
```
if AR[t] = 032 then
    Exception (IntegerDivideByZero)
else
    tq ← (0#AR[s]) quo (0#AR[t])
    AR[r] ← tq31..0
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(IntegerDivideByZeroCause) if 32-bit Integer Divide Option

8.3.244 RDTLB0—Read Data TLB Entry Virtual

Instruction Word (RRR)



Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
RDTLB0 at, as
```

Description

RDTLB0 reads the data TLB entry specified by the contents of address register `as` and writes the Virtual Page Number (VPN) and address space ID (ASID) to address register `at`. See [Options for Memory Protection and Translation](#) on page 183 for information on the address and result register formats for specific memory protection and translation options.

RDTLB0 is a privileged instruction.

Operation

```
AR[t] ← RDTLB0(AR[s])
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.245 RDTLB1—Read Data TLB Entry Translation

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	1	0	1	0	0	0	0	1	1	1	1	s	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
RDTLB1 at, as
```

Description

RDTLB1 reads the data TLB entry specified by the contents of address register `as` and writes the Physical Page Number (PPN) and cache attribute (CA) to address register `at`. See [Options for Memory Protection and Translation](#) on page 183> for information on the address and result register formats for specific memory protection and translation options.

RDTLB1 is a privileged instruction.

Operation

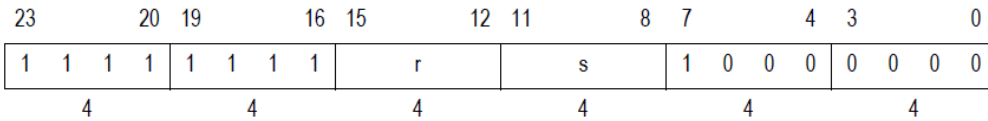
```
AR[t] ← RDTLB1(AR[s])
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.246 RECIP0.D—Reciprocal Begin Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
RECIP0.D fr, fs
```

Description

RECIP0.D is the first step of a Newton-Raphson reciprocal computation. A rough approximation of the reciprocal of the argument in `fs` is computed by table lookup and placed in `fr`. IEEE flags are set for the reciprocal operation. The approximation is accurate enough that three Newton-Raphson steps are sufficient for an accuracy better than 1-ulp. This instruction is not intended for use anywhere but in a reciprocal sequence. For more on how to use RECIP0.D see [Divide and Square Root Sequences](#) on page 110.

Operation

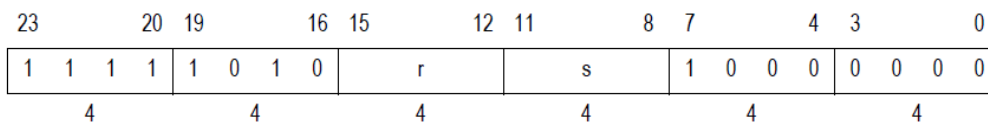
```
FR[r] ← reciprocal_approximation(FR[s])  
FSR[StatusFlags: VZOU] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.247 RECIP0.S—Reciprocal Begin Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
RECIP0.S fr, fs
```

Description

RECIP0.S is the first step of a Newton-Raphson reciprocal computation. A rough approximation of the reciprocal of the argument in `fs` is computed by table lookup and placed in `fr`. IEEE flags are set for the reciprocal operation. The approximation is accurate enough that two Newton-Raphson steps are sufficient for an accuracy better than 1-ulp. This instruction is not intended for use anywhere but in a reciprocal sequence. For more on how to use RECIP0.S see [Divide and Square Root Sequences](#) on page 110.

Operation

```
FR[r] ← reciprocal_approximation(FR[s])  
FSR[StatusFlags: VZOU] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.248 REMS—Remainder Signed

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	1	1	0	0	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

32-bit Integer Divide Option (See [32-bit Integer Divide Option](#) on page 90)

Assembler Syntax

```
REMS ar, as, at
```

Description

REMS performs a 32-bit two's complement division of the contents of address register `as` by the contents of address register `at` and writes the remainder to address register `ar`. The ambiguity which exists when either address register `as` or address register `at` is negative is resolved by requiring the remainder to have the same sign as address register `as`. If the

contents of address register `at` are zero, `REMS` raises an Integer Divide by Zero exception instead of writing a result.

Operation

```
if AR[t] = 032 then
    Exception (IntegerDivideByZero)
else
    AR[r] ← AR[s] rem AR[t]
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(IntegerDivideByZeroCause) if 32-bit Integer Divide Option

8.3.249 REMU—Remainder Unsigned

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0			
1	1	1	0	0	0	1	0	r	s	t	0	0	0	0
4				4				4		4		4		

Required Configuration Option

32-bit Integer Divide Option (See [32-bit Integer Divide Option](#) on page 90)

Assembler Syntax

```
REMU ar, as, at
```

Description

`REMU` performs a 32-bit unsigned division of the contents of address register `as` by the contents of address register `at` and writes the remainder to address register `ar`. If the contents of address register `at` are zero, `REMU` raises an Integer Divide by Zero exception instead of writing a result.

Operation

```
if AR[t] = 032 then
    Exception (IntegerDivideByZero)
else
    tr ← (0|AR[s]) rem (0|AR[t])
    AR[r] ← tr31..0
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(IntegerDivideByZeroCause) if 32-bit Integer Divide Option

8.3.250 RER—Read External Register

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0		
0	1	0	0	0	0	0	0	0	0	0	0		
4				4				4		4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
RER at, as
```

Description

RER reads one of a set of "External Registers". It is in some ways similar to the RSR.* instruction except that the registers being read are not defined by the Xtensa ISA and are conceptually outside the processor core. They are read through processor ports.

Address register `as` is used to determine which register is to be read and the result is placed in address register `at`. When no External Register is addressed by the value in address register `as`, the result in address register `at` is undefined. The entire address space is reserved for use by Cadence. RER and WER are managed by the processor core so that the requests appear on the processor ports in program order. External logic is responsible for extending that order to the registers themselves.

In older implementations, RER is a privileged instruction while in newer implementations, parts of the address space can be privileged as determined by the ERACCESS Special Register ([page 340](#)).

Operation

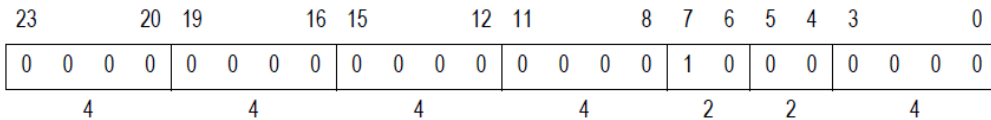
```
if (CRING ≠ 0 & PrivilegedAddressRegion) then
    Exception (ExternalRegPrivilegeCause)
else
    Read External Register as defined outside the processor.
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.251 RET—Non-Windowed Return

Instruction Word (CALLX)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
RET
```

Description

RET returns from a routine called by CALL0 or CALLX0. It is equivalent to the instruction

```
JX A0
```

RET exists as a separate instruction because some Xtensa ISA implementations may realize performance advantages from treating this operation as a special case.

Assembler Note

The assembler may convert RET instructions to RET.N when the Code Density Option is enabled. Prefixing the RET instruction with an underscore (`_RET`) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

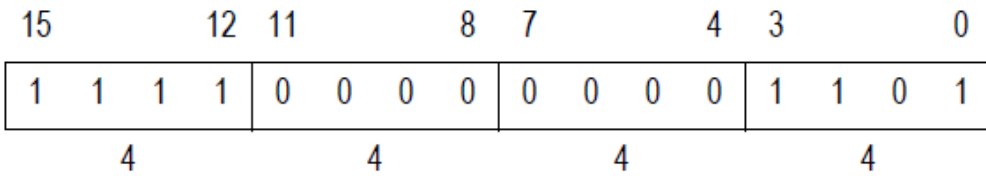
```
nextPC ← AR[0]
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.252 RET.N—Narrow Non-Windowed Return

Instruction Word (RRRN)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

```
RET.N
```

Description

RET.N is the same as RET in a 16-bit encoding. RET returns from a routine called by CALL0 or CALLX0.

Assembler Note

The assembler may convert RET.N instructions to RET. Prefixing the RET.N instruction with an underscore (`_RET.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

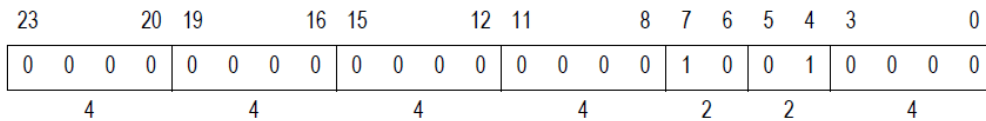
```
nextPC ← AR[0]
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.253 RETW—Windowed Return

Instruction Word (CALLX)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

RETW

Description

RETW returns from a subroutine called by CALL4, CALL8, CALL12, CALLX4, CALLX8, or CALLX12, and that had ENTRY as its first instruction.

Under the Windowed Register Option, RETW uses bits 29..0 of address register a0 as the low 30 bits of the return address and bits 31..30 of the address of the RETW as the high two bits of the return address. Bits 31..30 of a0 are used as the caller's window increment.

Under the Windowed Register Option, RETW subtracts the window increment from WindowBase to return to the caller's registers. It then checks the WindowStart bit for this WindowBase. If it is set, then the caller's registers still reside in the register file, and RETW completes by clearing its own WindowStart bit, jumping to the return address, and, in some implementations, setting PS.CALLINC to bits 31..30 of a0. If the WindowStart bit is clear, then the caller's registers have been stored into the stack, so RETW signals one of window underflow's 4, 8, or 12, based on the size of the caller's window increment. The underflow handler is invoked with WindowBase decremented, a minor exception to the rule that instructions aborted by an exception have no side effects to the operating state of the processor. The processor stores the previous value of WindowBase in PS.OWB so that it can be restored by RFWU.

Under the Windowed Register Option, the window underflow handler is expected to restore the caller's registers, set the caller's WindowStart bit, and then return (see RFWU) to re-execute the RETW, which will then complete.

Under the Windowed Register Option, the operation of this instruction is undefined if $AR[0]_{31..30}$ is 0², if PS.WOE is 0, if PS.EXCM is 1, or if the first set bit among $[WindowStart_{WindowBase-1}, WindowStart_{WindowBase-2}, WindowStart_{WindowBase-3}]$ is anything other than $WindowStart_{WindowBase-n}$, where n is $AR[0]_{31..30}$. (If none of the three bits is set, an underflow exception will be raised as described above, but if the wrong first one is set, the state is not legal.) Some implementations raise an illegal instruction exception in these cases as a debugging aid.

Assembler Note

The assembler may convert RETW instructions to RETW.N when the Code Density Option is enabled. Prefixing the RETW instruction with an underscore (RETW) disables this optimization and forces the assembler to generate the wide form of the instruction.

Operation

```

n ← AR[0]31..30
nextPC ← PC31..30 | AR[0]29..0
owb ← WindowBase
m ← if WindowStartWindowBase-4'b0001 then 2'b01
    elsif WindowStartWindowBase-4'b0010 then 2'b10
    elsif WindowStartWindowBase-4'b0011 then 2'b11
    else 2'b00
if n=2'b00 | (m≠2'b00 & m≠n) | PS.WOE=0 | PS.EXCM=1 then
    -- undefined operation
    -- may raise illegal instruction exception
else
    if WindowStartWindowBase - (021n) ≠ 0 then
        WindowStartowb ← 0
    else
        -- Underflow exception
        PS.EXCM ← 1
        EPC[1] ← PC
        PS.OWB ← owb
        nextPC ← if n = 2'b01 then WindowUnderflow4
            else if n = 2'b10 then WindowUnderflow8
            else WindowUnderflow12
    endif
    WindowBase ← WindowBase - (021n)
    PS.CALLINC ← n -- in some implementations
endif
endif

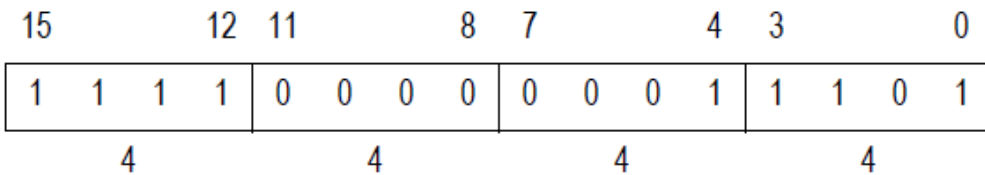
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- WindowUnderExcep

8.3.254 RETW.N—Narrow Windowed Return

Instruction Word (RRRN)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82) and Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
RETW.N
```

Description

RETW.N is the same as RETW in a 16-bit encoding.

Assembler Note

The assembler may convert `RETW,N` instructions to `RETW`. Prefixing the `RETW,N` instruction with an underscore (`_RETW,N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

Operation

```
n ← AR[0]31..30
nextPC ← PC31..30 | AR[0]29..0
owb ← WindowBase
m ← if WindowStartWindowBase-4'b0001 then 2'b01
    elsif WindowStartWindowBase-4'b0010 then 2'b10
    elsif WindowStartWindowBase-4'b0011 then 2'b11
    else 2'b00
if n=2'b00 | (m≠2'b00 & m≠n) | PS.WOE=0 | PS.EXCM=1 then
    -- undefined operation
    -- may raise illegal instruction exception
else
    WindowBase ← WindowBase - (02 | n)
    if WindowStartWindowBase ≠ 0 then
        WindowStartowb ← 0
    else
        -- Underflow exception
        PS.EXCM ← 1
        EPC[1] ← PC
        PS.OWB ← owb
        nextPC ← if n = 2'b01 then WindowUnderflow4
                 else if n = 2'b10 then WindowUnderflow8
                 else WindowUnderflow12
    endif
    PS.CALLINC ← n -- in some implementations
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- WindowUnderExcep

8.3.255 RFDD—Return from Debug and Dispatch

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0					
1	1	1	1	0	0	0	1	1	1	0	0	0	0	0	0	0
4				4				4				4				

Required Configuration Option

Debug Option (See [Debug Option](#) on page 256) and OCD, Implementation-Specific

Assembler Syntax

RFDD

Description

This instruction is used only in On-Chip Debug Mode and exists only in some implementations. It is an illegal instruction when the processor is not in On-Chip Debug Mode. See the *Xtensa Debug Guide* for a description of its operation.

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2

8.3.256 RFDE—Return from Double Exception

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0
0	0	0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

Exception Option 2 (See [Exception Option 2](#) on page 126)

Assembler Syntax

RFDE

Description

RFDE returns from an exception that went to the double exception vector (that is, an exception raised while the processor was executing with `PS.EXCM` set). It is similar to RFE, but `PS.EXCM` is not cleared, and `DEPC`, if it exists, is used instead of `EPC[1]`. RFDE simply jumps to the exception PC. `PS.UM` and `PS.WOE` are left unchanged.

RFDE is a privileged instruction.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
elseif NDEPC=1 then
    nextPC ← DEPC
else
```

```

nextPC ← EPC[1]
endif

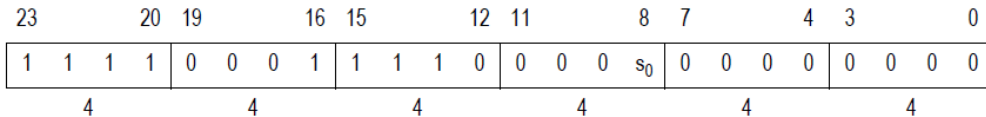
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.257 RFDO—Return from Debug Operation

Instruction Word (RRR)



Required Configuration Option

Debug Option (See [Debug Option](#) on page 256) and also OCD, Implementation-Specific

Assembler Syntax

```
RFDO
```

Description

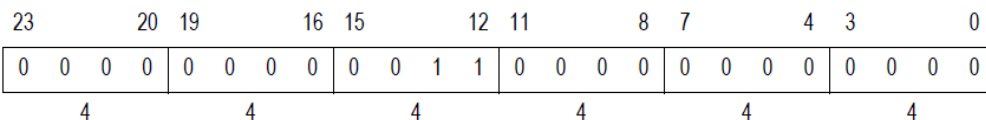
This instruction is used only in On-Chip Debug Mode and exists only in some implementations. It is an illegal instruction when the processor is not in On-Chip Debug Mode. See the *Xtensa Debug Guide* for a description of its operation.

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- GenExcep(IllegalInstructionCause)

8.3.258 RFE—Return from Exception

Instruction Word (RRR)



Required Configuration Option

Exception Option 2 (See [Exception Option 2](#) on page 126)

Assembler Syntax

```
RFE
```

Description

RFE returns from either the UserExceptionVector or the KernelExceptionVector. RFE sets PS.EXCM back to 0, and then jumps to the address in EPC[1]. PS.UM and PS.WOE are left unchanged.

RFE is a privileged instruction.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    PS.EXCM ← 0
    nextPC ← EPC[1]
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.259 RFI—Return from High-Priority Interrupt

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	0	0	0	0	0	0	0	0	1	0	0	0	0		
4				4				4				4			

Required Configuration Option

High-Priority Interrupt Option (See [High-Priority Interrupt Option](#) on page 157)

Assembler Syntax

```
RFI 0..15
```

Description

RFI returns from a high-priority interrupt. It restores the PS from EPS[level] and jumps to the address in EPC[level]. Level is given as a constant 2..(NLEVEL+NNMI) in the instruction

word. The operation of this opcode when level is 0 or 1 or greater than (NLEVEL+NNMI) is undefined.

RFI is a privileged instruction.

Operation

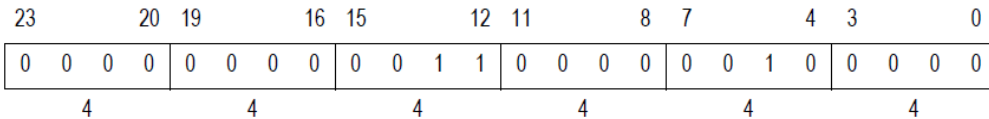
```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    nextPC ← EPC[level]
    PS ← EPS[level]
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.260 RFME—Return from Memory Error

Instruction Word (RRR)



Required Configuration Option

Memory ECC/Parity Option (See [Memory ECC/Parity Option](#) on page 168)

Assembler Syntax

```
RFME
```

Description

RFME returns from a memory error exception. It restores the PS from MEPS and jumps to the address in MEPC. In addition, the MEME bit of the MESR register is cleared.

RFME is a privileged instruction.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    nextPC ← MEPC
```

```

PS ← MEPS
MESR.MEME ← 0
endif

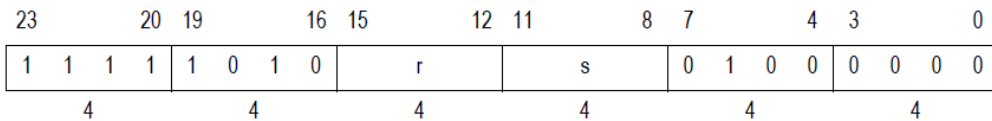
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.261 RFR—Move FR to AR

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
RFR ar, fs
```

Description

RFR moves the contents of floating-point register `fs` to address register `ar`. The move is non-arithmetic; no floating-point exceptions are raised. When floating point support is double-precision, this instruction moves the lower half of the floating point register.

Operation

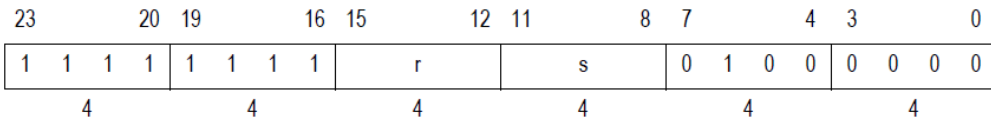
```
AR[r] ← FR[s]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.262 RFRD—Move FR to AR Upper

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
RFRD ar, fs
```

Description

RFRD moves the upper half of the contents of floating-point register `fs` to address register `ar`. The move is non-arithmetic; no floating-point exceptions are raised. The lower half of the register can be moved using the RFR instruction.

Operation

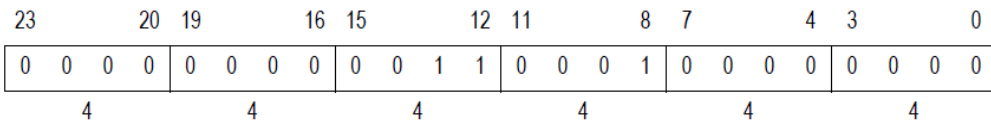
```
AR[r] ← Upper(FR[s])
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.263 RFUE—Return from User-Mode Exception

Instruction Word (RRR)



Required Configuration Option

Exclusive Access Option (Xtensa Exception Architecture 1 Only)

Assembler Syntax

```
RFUE
```

Description

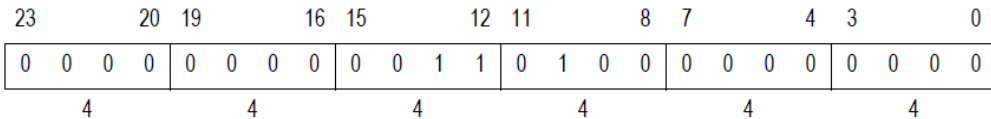
RFUE exists only in Xtensa Exception Architecture 1 (see [Xtensa Exception Architecture 1](#)). It is an illegal instruction in current Xtensa implementations.

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2

8.3.264 RFWO—Return from Window Overflow

Instruction Word (RRR)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
RFWO
```

Description

RFWO returns from an exception that went to one of the three window overflow vectors. It sets PS.EXCM back to 0, clears the WindowStart bit of the registers that were spilled, restores WindowBase from PS.OWB, and then jumps to the address in EPC[1]. PS.UM is left unchanged.

RFWO is a privileged instruction.

Operation

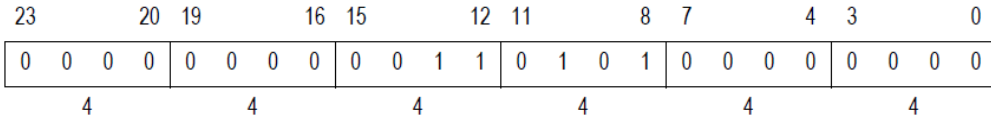
```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    PS.EXCM ← 0
    nextPC ← EPC[1]
    WindowStartWindowBase ← 0
    WindowBase ← PS.OWB
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.265 RFWU—Return From Window Underflow

Instruction Word (RRR)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
RFWU
```

Description

RFWU returns from an exception that went to one of the three window underflow vectors. It sets PS.EXCM back to 0, sets the WindowStart bit of the registers that were reloaded, restores WindowBase from PS.OWB, and then jumps to the address in EPC[1]. PS.UM is left unchanged.

RFWU is a privileged instruction.

Operation

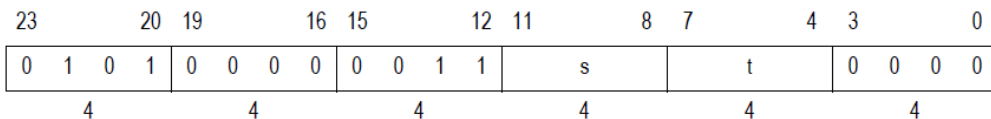
```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    PS.EXCM ← 0
    nextPC ← EPC[1]
    WindowStartWindowBase ← 1
    WindowBase ← PS.OWB
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.266 RITLB0—Read Instruction TLB Entry Virtual

Instruction Word (RRR)



Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
RITLBO at, as
```

Description

RITLBO reads the instruction TLB entry specified by the contents of address register *as* and writes the Virtual Page Number (VPN) and address space ID (ASID) to address register *at*. See [Options for Memory Protection and Translation](#) on page 183 for information on the address and result register formats for specific memory protection and translation options.

RITLBO is a privileged instruction.

Operation

```
AR[t] ← RITLBO(AR[s])
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.267 RITLB1—Read Instruction TLB Entry Translation

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	1	0	1	0	0	0	0	0	1	1	1	s	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
RITLB1 at, as
```

Description

RITLB1 reads the instruction TLB entry specified by the contents of address register `as` and writes the Physical Page Number (PPN) and cache attribute (CA) to address register `at`. See [Options for Memory Protection and Translation](#) on page 183 for information on the address and result register formats for specific memory protection and translation options.

RITLB1 is a privileged instruction.

Operation

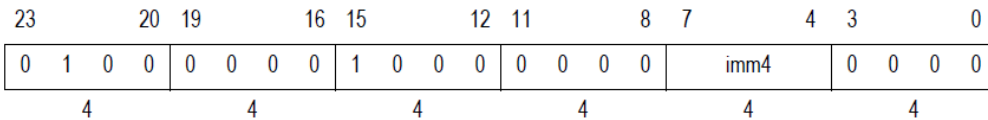
```
AR[t] ← RITLB1(AR[s])
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.268 ROTW—Rotate Window

Instruction Word (RRR)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
ROTW -8..7
```

Description

Under the Windowed Register Option, ROTW adds a constant to `WindowBase`, thereby moving the current window into the register file. ROTW is intended for use in exception handlers and context switch code.

ROTW is a privileged instruction.

Operation

```
if Windowed Register Option & CRING ≠ 0 then
    Exception (PrivilegedCause)
elseif Exception Option & PS.Ring ≠ 0 then
    Exc(User executes privileged instruction)
else
```

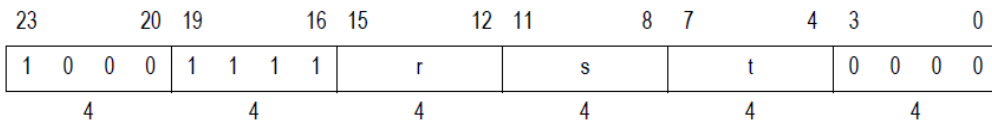
```
WindowBase - WindowBase + imm4 if Windowed Register Option
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.269 ROUND.D—Round Double to Fixed

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ROUND.D ar, fs, 0..15
```

Description

`ROUND.D` converts the contents of floating-point register `fs` from double-precision to signed integer format, rounding toward the nearest. The double-precision value is first scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 2.0, 4.0, ..., 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0` and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $\geq 2^{31} - 0.5$), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (scaled argument $< -2^{31} - 0.5$) or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

```
AR[r] ← roundD(FR[s] ×D powD(2.0,t))
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.270 ROUND.S—Round Single to Fixed

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	0	0	0	1	0	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ROUND.S ar, fs, 0..15
```

Description

`ROUND.S` converts the contents of floating-point register `fs` from single-precision to signed integer format, rounding toward the nearest. The single-precision value is first scaled by a power of two constant value encoded in the `t` field, with `0..15` representing 1.0, 2.0, 4.0, ..., 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0` and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $\geq 2^{31} - 0.5$), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (scaled argument $< -2^{31} - 0.5$) or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

```
AR[r] ← rounds(FR[s] ×s pows(2.0,t))  
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.271 RPTLB0—Read Protection TLB Entry Address

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	1	0	1	0	0	0	0	1	0	1	1	s	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Memory Protection Unit Option ([Memory Protection Unit Option](#) on page 205)

Assembler Syntax

```
RPTLB0 at, as
```

Description

RPTLB0 reads the Protection TLB segment specified by the contents of address register `as` and places the result in address register `at`. See [Formats for Reading Memory Protection Unit Option TLB Entries](#) on page 212 for information on address and result register formats.

RPTLB0 is a privileged instruction.

Operation

```
AR[t] ← RPTLB0(AR[s])
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.272 RPTLB1—Read Protection TLB Entry Info

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	1	0	1	0	0	0	0	1	1	1	1	s	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Memory Protection Unit Option ([Memory Protection Unit Option](#) on page 205)

Assembler Syntax

```
RPTLB1 at, as
```

Description

RPTLB1 reads the Protection TLB segment specified by the contents of address register `as` and places the result in address register `at`. See [Formats for Reading Memory Protection Unit Option TLB Entries](#) on page 212 for information on address and result register formats.

RPTLB1 is a privileged instruction.

Operation

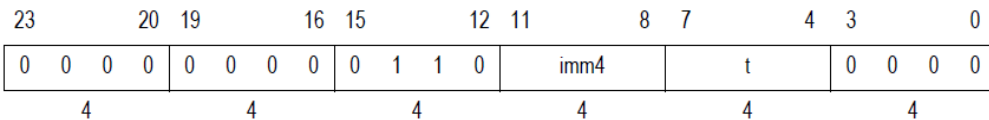
```
AR[t] ← RPTLB1(AR[s])
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.273 RSIL—Read and Set Interrupt Level

Instruction Word (RRR)



Required Configuration Option

Interrupt Option (See [Interrupt Option](#) on page 151)

Assembler Syntax

```
RSIL at, 0..15
```

Description

RSIL first reads the PS Special Register (described in [PS Register Fields](#), PS Register Fields), writes this value to address register `at`, and then sets `PS.INTLEVEL` to a constant in the range `0..15` encoded in the instruction word. Interrupts at and below the `PS.INTLEVEL` level are disabled.

A `WSR.PS` or `XSR.PS` followed by an `RSIL` should be separated with an `ESYNC` to guarantee the value written is read back.

On some Xtensa ISA implementations the latency of `RSIL` is greater than one cycle, and so it is advantageous to schedule uses of the `RSIL` result later.

`RSIL` is typically used as follows:

```
RSIL a2, newlevel
code to be executed at newlevel
WSR.PS a2
```

The instruction following the `RSIL` is guaranteed to be executed at the new interrupt level specified in `PS.INTLEVEL`, therefore it is not necessary to insert one of the `SYNC` instructions to force the interrupt level change to take effect.

`RSIL` is a privileged instruction.

Operation

```

if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    AR[t] ← PS
    PS.INTLEVEL ← s
endif

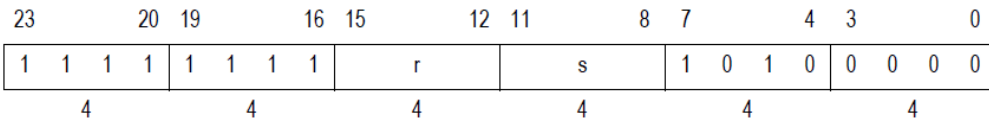
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.274 `RSQRT0.D`—Reciprocal Sqrt Begin Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
RSQRT0.D fr, fs
```

Description

`RSQRT0.D` is the first step of a Newton-Raphson reciprocal square-root computation. A rough approximation of the reciprocal square-root of the argument in `fs` is computed by table lookup and placed in `fr`. IEEE flags are set for the reciprocal square-root operation. The approximation is accurate enough that three Newton-Raphson steps are sufficient for an accuracy better than 2-ulps. This instruction is not intended for use anywhere but in a reciprocal square root sequence. For more on how to use `RSQRT0.D` see [Divide and Square Root Sequences](#) on page 110.

Operation

```
FR[r] ← reciprocal_square_root_approximation(FR[s])
FSR[StatusFlags: VZI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.275 RSQRT0.S—Reciprocal Sqrt Begin Single

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
1	1	1	1	1	0	1	0	r	s	1	0	1	0	0	0	0	0
4				4				4				4					

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
RSQRT0.S fr, fs
```

Description

`RSQRT0.S` is the first step of a Newton-Raphson reciprocal square-root computation. A rough approximation of the reciprocal square-root of the argument in `fs` is computed by table lookup and placed in `fr`. IEEE flags are set for the reciprocal square-root operation. The approximation is accurate enough that two Newton-Raphson steps are sufficient for an accuracy better than 2-ulps. This instruction is not intended for use anywhere but in a reciprocal square root sequence. For more on how to use `RSQRT0.S` see [Divide and Square Root Sequences](#) on page 110.

Operation

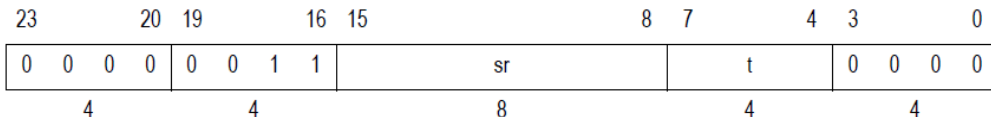
```
FR[r] ← reciprocal_square_root_approximation(FR[s])
FSR[StatusFlags: VZI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.276 RSR.*—Read Special Register

Instruction Word (RSR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
RSR.* at  
RSR at, *  
RSR at, 0..255
```

Description

RSR.* reads the Special Registers that are described in [Processor Control Instructions](#) on page 70. See [Special Registers](#) on page 272 for more detailed information on the operation of this instruction for each Special Register.

The contents of the Special Register designated by the 8-bit `sr` field of the instruction word are written to address register `at`. The name of the Special Register is used in place of the “*” in the assembler syntax above and the translation is made to the 8-bit `sr` field by the assembler.

RSR is an assembler macro for RSR.* that provides compatibility with the older versions of the instruction containing either the name or the number of the Special Register.

A `WSR.*` followed by an `RSR.*` to the same register should be separated with `ESYNC` to guarantee the value written is read back. On some Xtensa ISA implementations, the latency of `RSR.*` is greater than one cycle, and so it is advantageous to schedule other instructions before instructions that use the `RSR.*` result.

RSR.* with Special Register numbers ≥ 64 is privileged. An `RSR.*` for an unconfigured register generally will raise an illegal instruction exception.

Operation

```
if sr ≥ 64 and CRING ≠ 0 then  
    Exception (PrivilegedCause)  
else  
    see the Tables in Special Registers on page 272  
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.277 RSYNC—Register Read Synchronize

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0
0	0	0	0	0	0	0	0	0	0	0	0
4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
RSYNC
```

Description

RSYNC waits for all previously fetched WSR.* instructions to be performed before interpreting the register fields of the next instruction. This operation is also performed as part of ISYNC. ESYNC and DSYNC are performed as part of this instruction.

This instruction is appropriate after WSR.WindowBase, WSR.WindowStart, WSR.PS, WSR.CPENABLE, or WSR.EPS* instructions before using their results. See the Special Register Tables in [Special Registers](#) on page 272 for a complete description of the uses of the RSYNC instruction.

Because the instruction execution pipeline is implementation-specific, the operation section below specifies only a call to the implementation's `rsync` function.

Operation

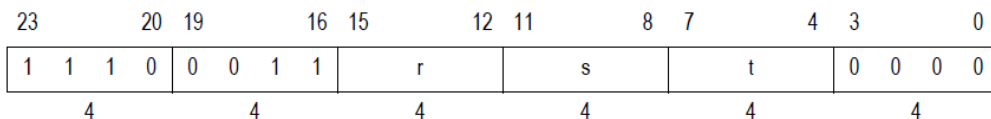
```
rsync()
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.278 RUR.*—Read User Register

Instruction Word (RRR)



Required Configuration Option

No Option - instructions created from the TIE language (See [Coprocessor Context Switch](#) on page 150)

Assembler Syntax

```
RUR.* ar
RUR ar, *
```

Description

RUR.* reads TIE state that has been grouped into 32-bit quantities by the TIE `user_register` statement. The name in the `user_register` statement replaces the “*” in the instruction name and causes the correct register number to be placed in the `st` field of the encoded instruction. The contents of the TIE `user_register` designated by the 8-bit number $16*s+t$ are written to address register `ar`. Here `s` and `t` are the numbers corresponding to the respective fields of the instruction word.

RUR is an assembler macro for RUR.*, which provides compatibility with the older version of the instruction.

Operation

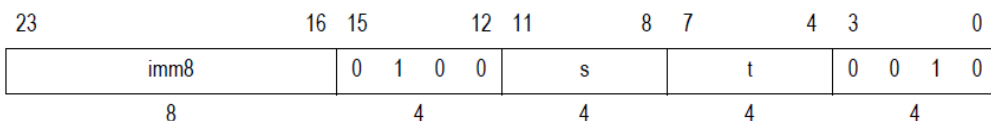
```
AR[r] ← user_register[st]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(Coprocessor*Disabled) if Exception Option 2 and Coprocessor Context Option

8.3.279 S8I—Store 8-bit

Instruction Word (RR18)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
S8I at, as, 0..255
```

Description

S8I is an 8-bit store from address register `at` to memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word. Therefore, the offset has a range from 0 to 255. Eight bits (1 byte) from the least significant quarter of address register `at` are written to memory at the physical address.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Operation

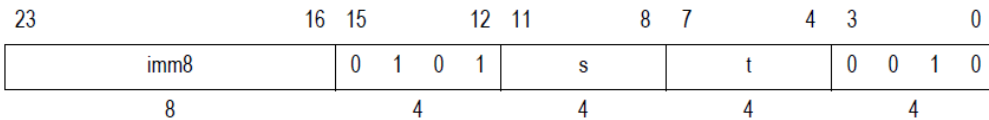
```
vAddr ← AR[s] + (024 | imm8)  
Store8 (vAddr, AR[t]7..0)
```

Exceptions

- Memory Group (see [Memory Group](#))
- GenExcep(StoreProhibitedCause) if Region Protection Option or MMU Option
- DebugExcep(DBREAK) if Debug Option

8.3.280 S16I—Store 16-bit

Instruction Word (RR18)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
S16I at, as, 0..510
```

Description

S16I is a 16-bit store from address register `at` to memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by one. Therefore, the offset can specify multiples of two from zero to 510. Sixteen bits (two bytes) from the least significant half of the register are written to memory at the physical address.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the least significant bit of the address is ignored. A reference to an odd address produces the same result as a reference to the address, minus one. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, S16I calculates the sum of address register `as` and the `imm8` field of the instruction word times two. Therefore, the machine-code offset is in terms of 16-bit (2 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by two.

Operation

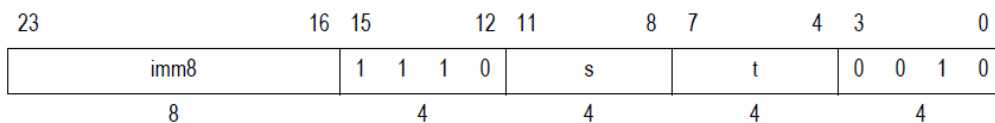
```
vAddr ← AR[s] + (023imm810)
Store16 (vAddr, AR[t]15..0)
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))

8.3.281 S32C1I—Store 32-bit Compare Conditional

Instruction Word (RR18)



Required Configuration Option

Conditional Store Option (See [Conditional Store Option](#) on page 118)

Assembler Syntax

```
s32c1i at, as, 0..1020
```

Description

`S32C1I` is a conditional store instruction intended for updating synchronization variables in memory shared between multiple processors. It may also be used to atomically update variables shared between different interrupt levels or other pairs of processes on a single processor. `S32C1I` attempts to store the contents of address register `at` to the virtual address formed by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. If the old contents of memory at the physical address equals the contents of the `SCOMPARE1` Special Register, the new data is written; otherwise the memory is left unchanged. In either case, the value read from the location is written to address register `at`. In some implementations, under unusual circumstances, the bitwise not of `SCOMPARE1` may be returned when memory is left unchanged instead of the current value of the memory location (see [S32C1I Modification](#)). The memory read, compare, and write may take place in the processor or the memory system, depending on the Xtensa ISA implementation, as long as these operations exclude other writes to this location. See [Conditional Store Option](#) on page 118 for more information on where the atomic operation takes place.

From a memory ordering point of view, the atomic pair of accesses has the characteristics of both an acquire and a release. That is, the atomic pair of accesses does not begin until all previous loads, stores, acquires, and releases have performed. The atomic pair must perform before any following load, store, acquire, or release may begin.

If the Region Translation Option () or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

`S32C1I` does both a load and a store when the store is successful. However, memory protection tests check for store capability and the instruction may raise a `StoreProhibitedCause` exception, but will never raise a `LoadProhibitedCause` exception.

Assembler Note

To form a virtual address, `S32C1I` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-

bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```

vAddr ← AR[s] + (022imm8102)
(mem32, error) ← Store32C1 (vAddr, AR[t], SCOMPARE1)
if error then
    EXCVADDR ← vAddr
    Exception (LoadStoreError)
else
    AR[t] ← One Of (mem32, ~SCOMPARE1)
endif

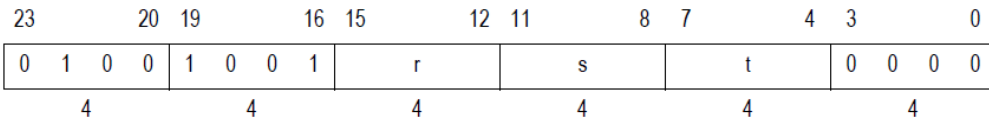
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))

8.3.282 S32E—Store 32-bit for Window Exceptions

Instruction Word (RRI4)



Required Configuration Option

Windowed Register Option (See [Windowed Register Option](#) on page 240)

Assembler Syntax

```
S32E at, as, -64..-4
```

Description

S32E is a 32-bit store instruction similar to S32I, but with semantics required by window overflow and window underflow exception handlers. In particular, memory access checking is done with PS.RING instead of CRING, and the offset used to form the virtual address is a 4-bit one-extended immediate. Therefore, the offset can specify multiples of four from -64 to -4. In configurations without the MMU Option, there is no PS.RING and S32E is similar to S32I with a negative offset.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent

memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

S32E is a privileged instruction.

In the context of special handler interface code, S32E has modified operation.

Assembler Note

To form a virtual address, S32E calculates the sum of address register `as` and the `r` field of the instruction word times four (and one extended). Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```

if Windowed Register Option & CRING ≠ 0 then
    Exception (PrivilegedCause)
elseif Exception Option & PS.Ring ≠ 0 then
    Exc(User executes privileged instruction)
else
    vAddr ← AR[s] + (126r102)
    ring ← if MMU Option then PS.RING else 0
    Store32Ring (vAddr, AR[t], ring)
endif

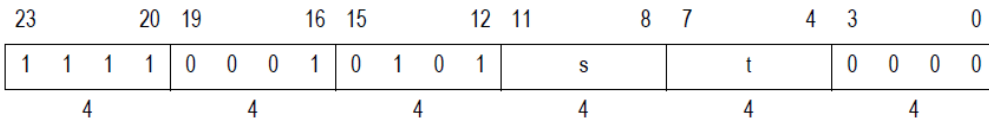
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.283 S32EX—Store 32-bit Exclusive

Instruction Word (RRR)



Required Configuration Option

Exclusive Access Option (See [Exclusive Access Option](#) on page 123)

Assembler Syntax

```
S32EX at, as
```

Description

S32EX is a conditional 32-bit store from address register `at` to memory. It uses address register `as` for its virtual address. The data to be stored is taken from the contents of address register `at`. If the physical address is marked as exclusive access, the store is completed, the exclusive mark is removed, and `ATOMCTL[8]` is set. If the physical address is not marked as exclusive access, no store to memory is done and `ATOMCTL[8]` is cleared. The previous value of `ATOMCTL[8]` is zero extended and moved to address register `at`. See [Exclusive Access Option](#) on page 123.

S32EX is intended to be followed by a GETEX instruction (see [Assembler Syntax](#)). The pair implements what is a store exclusive instruction in some architectures. The two are separated to improve interrupt latency. If both functions were done with a single instruction, the state save for an interrupt would need to wait for the memory system to acknowledge the write.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

The operation of S32EX will depend on the memory type associated with its address. It may operate entirely within a cache, by means of an ordinary external bus transaction, by means of a special external bus transaction, or by means of a series of coherent bus transactions.

Operation

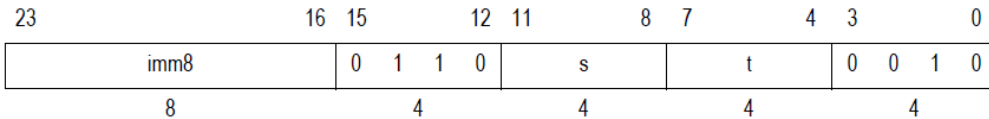
```
AR[t] ← 031||ATOMCTL8
if monitorset() then
    ATOMCTL8 ← Store32EX (AR[s], AR[t])
    clrmonitor()
else
    ATOMCTL8 ← 0
endif
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))

8.3.284 S32I—Store 32-bit

Instruction Word (RRI8)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
s32I at, as, 0..1020
```

Description

S32I is a 32-bit store from address register `at` to memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The data to be stored is taken from the contents of address register `at` and written to memory at the physical address.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

If the Instruction Memory Access Option ([Instruction Memory Access Option](#) on page 167) is configured, S32I is one of only a few memory reference instructions that can access instruction RAM.

Assembler Note

The assembler may convert S32I instructions to S32I.N when the Code Density Option is enabled and the `imm8` operand falls within the available range. Prefixing the S32I instruction with an underscore (`_S32I`) disables this optimization and forces the assembler to generate the wide form of the instruction.

To form a virtual address, `S32I` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

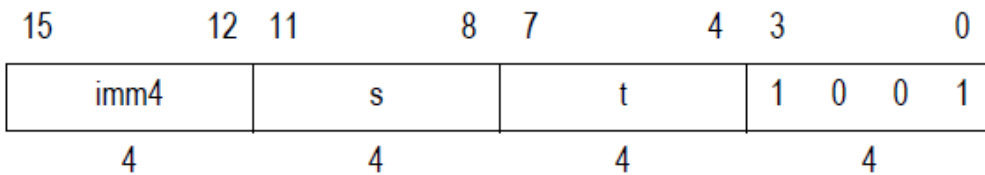
```
vAddr ← AR[s] + (022||imm8||02)
Store32 (vAddr, AR[t])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))

8.3.285 S32I.N—Narrow Store 32-bit

Instruction Word (RRRN)



Required Configuration Option

Code Density Option (See [Code Density Option](#) on page 82)

Assembler Syntax

```
S32I.N at, as, 0..60
```

Description

`S32I.N` is similar to `S32I`, but has a 16-bit encoding and supports a smaller range of offset values encoded in the instruction word.

`S32I.N` is a 32-bit store to memory. It forms a virtual address by adding the contents of address register `as` and an 4-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 60. The data to be stored is taken from the contents of address register `at` and written to memory at the physical address.

If the Instruction Memory Access Option ([Instruction Memory Access Option](#) on page 167) is configured, `S32I.N` is one of only a few memory reference instructions that can access instruction RAM.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Options, such an access raises an exception.

Assembler Note

The assembler may convert `S32I.N` instructions to `S32I`. Prefixing the `S32I.N` instruction with an underscore (`_S32I.N`) disables this optimization and forces the assembler to generate the narrow form of the instruction.

To form a virtual address, `S32I.N` calculates the sum of address register `as` and the `imm4` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

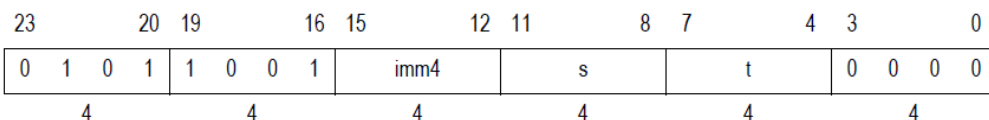
```
vAddr ← AR[s] + (026||imm4||02)
Store32 (vAddr, AR[t])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))

8.3.286 S32NB—Store 32-bit Non-Buffered

Instruction Word (RRI4)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
S32NB at, as, 0..60
```

Description

`S32NB` is a 32-bit store from address register `at` to memory. It forms a virtual address by adding the contents of address register `as` and a 4-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 60. The data to be stored is taken from the contents of address register `at` and written to memory at the physical address.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

`S32NB` provides the same functionality as `S32I` with two exceptions. First, when its operation leaves the processor, the external transaction is marked Non-Bufferable. Second, it may not be used to write to Instruction RAM.

Assembler Note

To form a virtual address, `S32NB` calculates the sum of address register `as` and the `imm4` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

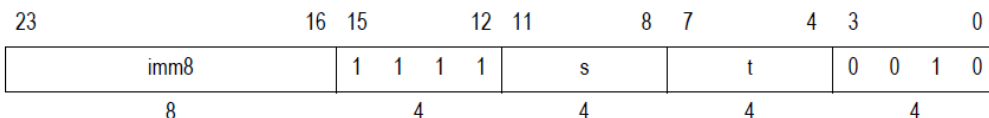
```
vAddr ← AR[s] + (026||imm4||02)
Store32 (vAddr, AR[t])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))

8.3.287 S32RI—Store 32-bit Release

Instruction Word (RRI8)



Required Configuration Option

Multiprocessor Synchronization Option (See [Multiprocessor Synchronization Option](#) on page 115)

Assembler Syntax

```
S32RI at, as, 0..1020
```

Description

`S32RI` is a store barrier and 32-bit store from address register `at` to memory. `S32RI` stores to synchronization variables, which signals that previously written data is “released” for consumption by readers of the synchronization variable. This store will not perform until all previous loads, stores, acquires, and releases have performed. This ensures that any loads of the synchronization variable that see the new value will also find all previously written data available as well.

`S32RI` forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. `S32RI` waits for previous loads, stores, acquires, and releases to be performed, and then the data to be stored is taken from the contents of address register `at` and written to memory at the physical address. Because the method of waiting is implementation dependent, this is indicated in the operation section below by the implementation function `release`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, `S32RI` calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

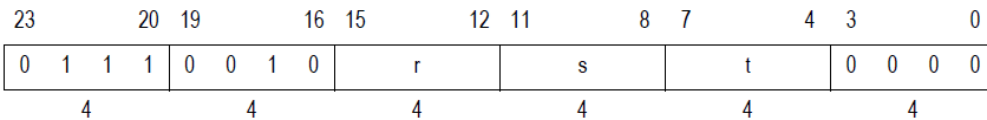
```
vAddr ← AR[s] + (022||imm8|02)
release()
Store32 (vAddr, AR[t])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))

8.3.288 SALT—Set AR if Less Than

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SALT ar, as, at
```

Description

The `SALT` instruction exists to improve the performance of a magnitude comparison. If address register `as` considered as a signed integer is less than address register `at` considered as a signed integer, then address register `ar` is set to `0x1`. Otherwise address register `ar` is set to `0x0`.

By reversing the position of the `as` and `at` registers and/or considering the result in the opposite sense, all four conditions of less-than, greater-than, less-than-or-equal, and greater-than-or-equal can be tested.

Operation

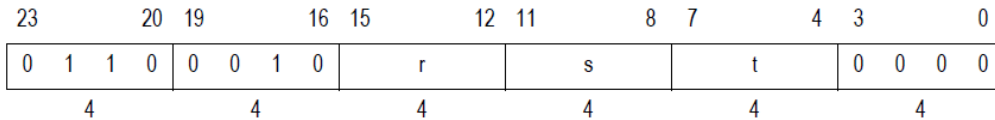
```
if AR[s] < AR[t] then
  AR[r] ← 031||1
else
  AR[r] ← 032
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.289 SALTU—Set AR if Less Than Unsigned

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SALTU ar, as, at
```

Description

The SALTU instruction exists to improve the performance of an unsigned magnitude comparison. If address register `as` considered as an unsigned integer is less than address register `at` considered as an unsigned integer, then address register `ar` is set to `0x1`. Otherwise address register `ar` is set to `0x0`.

By reversing the position of the `as` and `at` registers and/or considering the result in the opposite sense, all four conditions of less-than, greater-than, less-than-or-equal, and greater-than-or-equal can be tested.

Operation

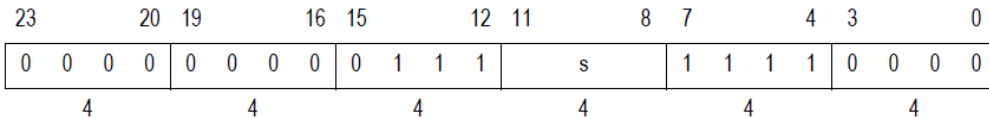
```
if AR[s] <u AR[t] then
    AR[r] ← 03111
else
    AR[r] ← 032
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.290 SDDR32.P—Store from DDR Register

Instruction Word (RRR)



Required Configuration Option

Debug Option (See [Debug Option](#) on page 256) and OCD, Implementation-Specific

Assembler Syntax

```
SDDR32.P as
```

Description

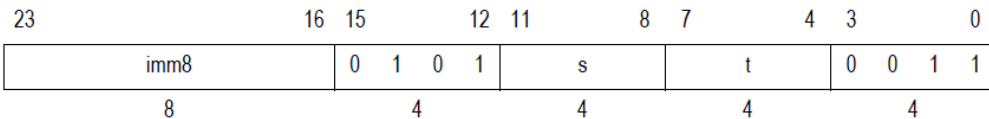
This instruction is used only in On-Chip Debug Mode and exists only in some implementations. It is an illegal instruction when the processor is not in On-Chip Debug Mode. See the *Xtensa Debug Guide* for a description of its operation.

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2

8.3.291 SDI—Store Double Immediate

Instruction Word (RR18)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SDI ft, as, 0..2040
```

Description

SDI is a 64-bit store from floating-point register `ft` to memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by three. Therefore, the offset can specify multiples of eight from zero to 2040. The data to be stored is taken from the contents of floating-point register `ft` and written to memory at the physical address.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the three least significant bits of the address are ignored. A reference to an address that is not 0 mod 8 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, SDI calculates the sum of address register `as` and the `imm8` field of the instruction word times eight. Therefore, the machine-code offset is in terms of 64-bit (8 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by eight.

Operation

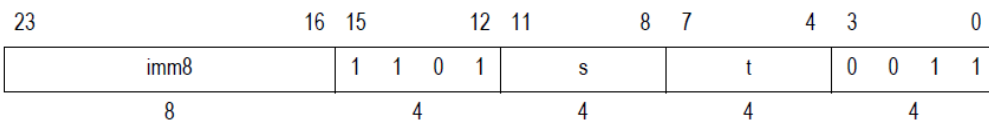
```
vAddr ← AR[s] + (021||imm8||03)
Store64 (vAddr, FR[t])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.292 SDIP—Store Double Immediate Post-Increment

Instruction Word (RRI8)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SDIP ft, as, 0..2040
```

Description

SDIP is a 64-bit store from floating-point register ft to memory with base address register post-increment. The virtual address is taken from the contents of address register as . The data to be stored is taken from the contents of floating-point register ft and written to memory at the physical address. The sum of the virtual address and an 8-bit zero-extended constant value encoded in the instruction word shifted left by three is written back to address register as . The increment can specify multiples of eight from zero to 2040.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the three least significant bits of the address are ignored. A reference to an address that is not 0 mod 8 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

SDIP calculates the increment of address register as using the $imm8$ field of the instruction word times eight. Therefore, the machine-code increment is in terms of 64-bit (8 byte) units. However, the assembler expects a byte increment and encodes this into the instruction by dividing by eight.

Operation

```
vAddr ← AR[s]
Store64 (vAddr, FR[t])
AR[s] ← vAddr + (021||imm8||03)
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.293 SDX—Store Double Indexed

Instruction Word (RRR)

	23		20	19		16	15		12	11		8	7		4	3		0
	0	1	1	0	1	0	0	0	r	s	t	0	0	0	0	0	0	0
	4		4		4		4		4		4		4		4		4	

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SDX fr, as, at
```

Description

SDX is a 64-bit store from floating-point register `fr` to memory. It forms a virtual address by adding the contents of address register `as` and the contents of address register `at`. The data to be stored is taken from the contents of floating-point register `fr` and written to memory at the physical address.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the three least significant bits of the address are ignored. A reference to an address that is not 0 mod 8 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```
vAddr ← AR[s] + (AR[t])  
Store64 (vAddr, FR[r])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.294 SDXP—Store Double Indexed Post-Increment

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	1	1	1	0	0	0	0	r	s	t	0	0	0	0
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SDXP fr, as, at
```

Description

SDXP is a 64-bit store from floating-point register `fr` to memory with base address register post-increment. The virtual address is taken from the contents of address register `as`. The data to be stored is taken from the contents of floating-point register `fr` and written to memory at the physical address. The sum of the virtual address and the contents of address register `at` is written back to address register `as`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the three least significant bits of the address are ignored. A reference to an address that is not 0 mod 8 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

```
vAddr ← AR[s]  
Store64 (vAddr, FR[r])  
AR[s] ← vAddr + (AR[t])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.295 SEXT—Sign Extend

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	0	1	0	0	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Miscellaneous Operations Option (See [Miscellaneous Operations Option](#) on page 94)

Assembler Syntax

```
SEXT ar, as, 7..22
```

Description

`SEXT` takes the contents of address register `as` and replicates the bit specified by its immediate operand (in the range 7 to 22) to the high bits and writes the result to address register `ar`. The input can be thought of as an $\text{imm}+1$ bit value with the high bits irrelevant and this instruction produces the 32-bit sign-extension of this value.

Assembler Note

The immediate values accepted by the assembler are 7 to 22. The assembler encodes these in the `t` field of the instruction using 0 to 15.

Operation

```
b ← t+7  
AR[r] ← AR[s]b31-b∣AR[s]b..0
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.296 SICT—Store Instruction Cache Tag

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
1	1	1	1	0	0	0	1	0	0	0	1	s	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Instruction Cache Test Option (See [Instruction Cache Test Option](#))

Assembler Syntax

```
SICT at, as
```

Description

`SICT` is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA and its exact method of addressing the cache may depend on the implementation.

SICT is intended for writing the RAM array that implements the instruction cache tags as part of manufacturing test.

SICT uses the contents of address register `as` to select a line in the instruction cache, and writes the contents of address register `at` to the tag associated with that line. The value written from `at` is described under Cache Tag Format in [Cache Tag Format](#) on page 163. Since SICT addresses memory differently than most memory accesses, its result is only certain to be seen by a following load if there has been a MEMW between the two.

SICT is a privileged instruction.

Operation

```

if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    index ← AR[s]x-1..z
    InstCacheTag[index] ← AR[t] // see Implementation Notes below
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```

x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)

```

The cache line specified by index `AR[s]x-1..z` in a direct-mapped cache or way `AR[s]x-1..y` and index `AR[s]y-1..z` in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does nothing. In configurations with more than one copy of Tag RAM, all copies will be written with the same value.

8.3.297 SICW—Store Instruction Cache Word

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0
1	1	1	1	0	0	0	1	0	0	1	s
4	4	4	4	4	4	4	4	4	4	4	4

Required Configuration Option

Instruction Cache Test Option (See [Instruction Cache Test Option](#))

Assembler Syntax

```
SICW at, as
```

Description

`SICW` is not part of the Xtensa Instruction Set Architecture, but is instead specific to an implementation. That is, it may not exist in all implementations of the Xtensa ISA and its exact method of addressing the cache may depend on the implementation.

`SICW` is intended for writing the RAM arrays that implement the instruction cache or other instruction related memory as part of manufacturing tests.

`SICW` uses the contents of address register `as` to select a line in the instruction cache, and writes the contents of address register `at` to the data associated with that line. The upper four bits of address register `as` may, in some implementations, be used to choose a RAM type to access. Since `SICW` addresses memory differently than most memory accesses, its result is only certain to be seen by a following load if there has been a `MEMW` between the two.

`SICW` is a privileged instruction.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    index ← AR[s]x-1..2
    InstCacheData [index] ← AR[t] // see Implementation Notes below
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))
- MemoryErrorException if Memory ECC/Parity Option

Implementation Notes

```
x ← ceil(log2(InstCacheBytes))
y ← log2(InstCacheBytes ÷ InstCacheWayCount)
z ← log2(InstCacheLineBytes)
```

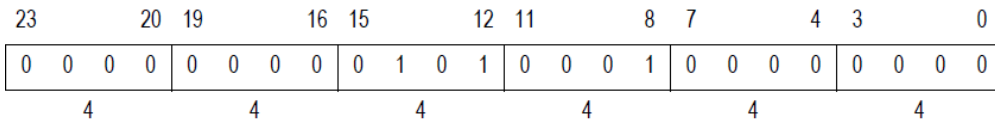
The cache line specified by index `AR[s]x-1..z` in a direct-mapped cache or way `AR[s]x-1..y` and index `AR[s]y-1..z` in a set-associative cache is the chosen line. If the specified cache way is not valid (the fourth way of a three way cache), the instruction does

nothing. Within the cache line, $AR[s]_{z-1..2}$ is used to determine which 32-bit quantity within the line is written.

The width of the instruction cache RAM may be more than 32 bits depending on the configuration. In that case, some implementations may write the same data replicated enough times to fill the entire width of the RAM.

8.3.298 SIMCALL—Simulator Call

Instruction Word (RRR)



Required Configuration Option

Xtensa Instruction Set Simulator

Assembler Syntax

```
SIMCALL
```

Description

`SIMCALL` is not implemented as a simulator call by any Xtensa processor hardware. Some older processors may raise an illegal instruction exception for this opcode while newer processors treat it as a `NOB` instruction. It is implemented by the Xtensa Instruction Set Simulator to allow simulated programs to request services of the simulator host processor. See the *Xtensa Instruction Set Simulator (ISS) User's Guide*.

The value in address register `a2` is the request code. Most codes request host system call services while others are used for special purposes such as debugging. Arguments needed by host system calls will be found in `a3` through `a7` and a return code will be stored to `a2` and an error number to `a3`.

Operation

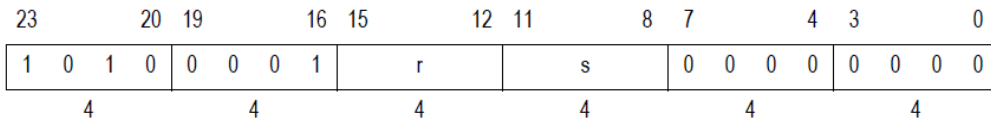
See the *Xtensa Instruction Set Simulator (ISS) User's Guide*.

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2

8.3.299 SLL—Shift Left Logical

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SLL ar, as
```

Description

SLL shifts the contents of address register `as` left by the number of bit positions specified (as 32 minus number of bit positions) in the SAR (shift amount register) and writes the result to address register `ar`. Typically the `SSL` or `SSA8L` instructions are used to specify the left shift amount by loading SAR with `32-shift`. This transformation allows SLL to be implemented in the SRC funnel shifter (which only shifts right), using the SLL data as the most significant 32 bits and zero as the least significant 32 bits. Note the result of SLL is undefined if `SAR > 32`.

Operation

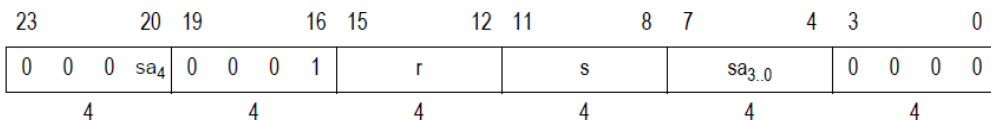
```
sa ← SAR5..0
AR[r] ← (AR[s] # 032)31+sa..sa
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.300 SLLI—Shift Left Logical Immediate

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SLLI ar, as, 1..31
```

Description

SLLI shifts the contents of address register *as* left by a constant amount in the range 1..31 encoded in the instruction. The shift amount *sa* field is split, with bits 3..0 in bits 7..4 of the instruction word and bit 4 in bit 20 of the instruction word. The shift amount is encoded as *32-shift*. When the *sa* field is 0, the result of this instruction is undefined.

Assembler Note

The shift amount is specified in the assembly language as the number of bit positions to shift left. The assembler performs the *32-shift* calculation when it assembles the instruction word. When the immediate operand evaluates to zero, the assembler converts this instruction to an OR instruction to effect a register-to-register move. To disable this transformation, prefix the mnemonic with an underscore (`_SLLI`). If *imm* evaluates to zero when the mnemonic has the underscore prefix, the assembler will emit an error.

Operation

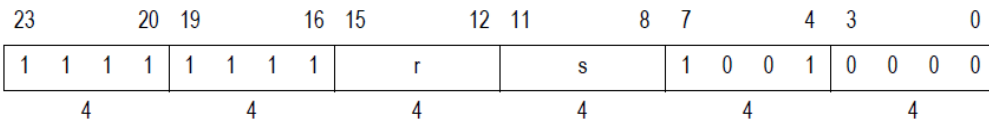
```
AR[r] ← (AR[s]!032)31+sa..sa
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.301 SQRT0.D—Square Root Begin Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SQRT0.D fr, fs
```

Description

`SQRT0.D` is the first step of a Newton-Raphson square root sequence which includes corrections to make it an IEEE compliant square root. The double-precision argument in floating-point register `fs` first has its range narrowed in the same way as the `NEXP01.D` instruction (see [Assembler Syntax](#)), but without the negation. A rough approximation of the reciprocal square root of that result is computed by table lookup and placed in `fr`. No status flags are updated. This instruction is not intended for use anywhere but in a square root sequence. For more on the IEEE exact square root sequence, see [Divide and Square Root Sequences](#) on page 110.

Operation

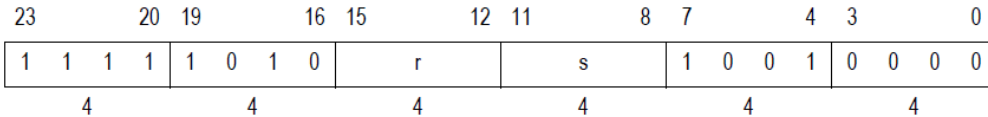
```
FR[r] ← begin_square_root_sequence(FR[s])
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.302 SQRT0.S—Square Root Begin Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SQRT0.S fr, fs
```

Description

`SQRT0.S` is the first step of a Newton-Raphson square root sequence which includes corrections to make it an IEEE compliant square root. The single-precision argument in floating-point register `fs` first has its range narrowed in the same way as the `NEXP01.S` instruction (see [Assembler Syntax](#)), but without the negation. A rough approximation of the reciprocal square root of that result is computed by table lookup and placed in `fr`. No status flags are updated. This instruction is not intended for use anywhere but in a square root sequence. For more on the IEEE exact square root sequence, see [Divide and Square Root Sequences](#) on page 110.

Operation

```
FR[r] ← begin_square_root_sequence(FR[s])
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.303 SRA—Shift Right Arithmetic

Instruction Word (RRR)

23	19	16	15	12	11	8	7	4	3	0		
1	0	1	1	0	0	0	1	r	0	0	0	0
4				4				4				

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SRA ar, at
```

Description

SRA arithmetically shifts the contents of address register *at* right, inserting the sign of *at* on the left, by the number of bit positions specified by *SAR* (shift amount register) and writes the result to address register *ar*. Typically the *SSR* or *SSA8B* instructions are used to load *SAR* with the shift amount from an address register. Note the result of SRA is undefined if *SAR* > 32.

Operation

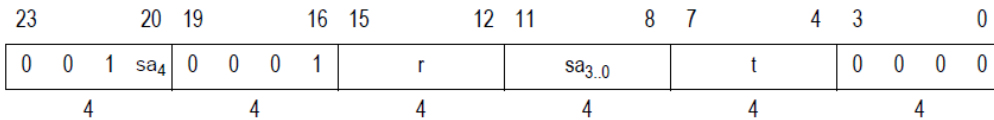
```
sa ← SAR5..0  
AR[r] ← ((AR[t]31)32 | AR[t]31+sa..sa)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.304 SRAI—Shift Right Arithmetic Immediate

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SRAI ar, at, 0..31
```

Description

SRAI arithmetically shifts the contents of address register `at` right, inserting the sign of `at` on the left, by a constant amount encoded in the instruction word in the range 0..31. The shift amount `sa` field is split, with bits 3..0 in bits 11..8 of the instruction word, and bit 4 in bit 20 of the instruction word.

Operation

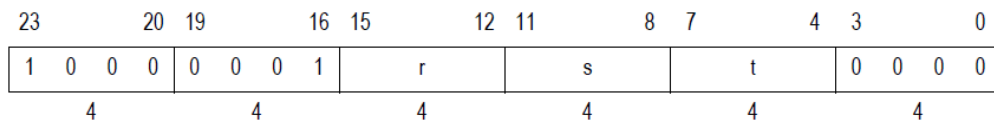
$$AR[r] \leftarrow ((AR[t]_{31})^{32} | AR[t]_{31+sa..sa})$$

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.305 SRC—Shift Right Combined

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SRC ar, as, at
```

Description

`SRC` performs a right shift of the concatenation of address registers `as` and `at` by the shift amount in `SAR`. The least significant 32 bits of the shift result are written to address register `ar`. A shift with a wider input than output is called a funnel shift. `SRC` directly performs right funnel shifts. Left funnel shifts are done by swapping the high and low operands to `SRC` and setting `SAR` to 32 minus the shift amount. The `SSL` and `SSA8B` instructions directly implement such `SAR` settings. Note the result of `SRC` is undefined if `SAR` > 32.

Operation

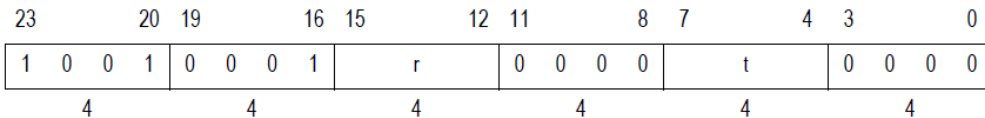
```
sa ← SAR5..0
AR[r] ← (AR[s] | AR[t])31+sa..sa
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.306 SRL—Shift Right Logical

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SRL ar, at
```

Description

`SRL` shifts the contents of address register `at` right, inserting zeros on the left, by the number of bits specified by `SAR` (shift amount register) and writes the result to address register `ar`. Typically the `SSR` or `SSA8B` instructions are used to load `SAR` with the shift amount from an address register. Note the result of `SRL` is undefined if `SAR` > 32.

Operation

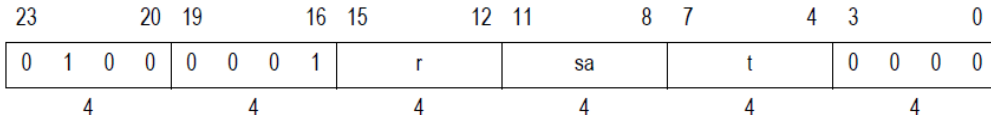
```
sa ← SAR5..0
AR[r] ← (032 | AR[t])31+sa..sa
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.307 SRLI—Shift Right Logical Immediate

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SRLI ar, at, 0..15
```

Description

SRLI shifts the contents of address register `at` right, inserting zeros on the left, by a constant amount encoded in the instruction word in the range 0..15. There is no SRLI for shifts ≥ 16 . EXTUI replaces these shifts.

Assembler Note

The assembler converts SRLI instructions with a shift amount ≥ 16 into EXTUI. Prefixing the SRLI instruction with an underscore (`_SRLI`) disables this replacement and forces the assembler to generate an error.

Operation

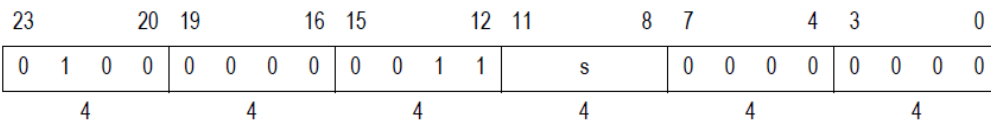
$$AR[r] \leftarrow (0^{32} \mid AR[t])_{31+sa..sa}$$

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.308 SSA8B—Set Shift Amount for BE Byte Shift

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SSA8B as
```

Description

SSA8B sets the shift amount register (*SAR*) for a left shift by multiples of eight (for example, for big-endian (BE) byte alignment). The left shift amount is the two least significant bits of address register *as* multiplied by eight. Thirty-two minus this amount is written to *SAR*. Using 32 minus the left shift amount causes a funnel right shift and swapped high and low input operands to perform a left shift. SSA8B is similar to *SSL*, except the shift amount is multiplied by eight.

SSA8B is typically used to set up for an *SRC* instruction to shift bytes. It may be used with big-endian byte ordering to extract a 32-bit value from a non-aligned byte address.

Operation

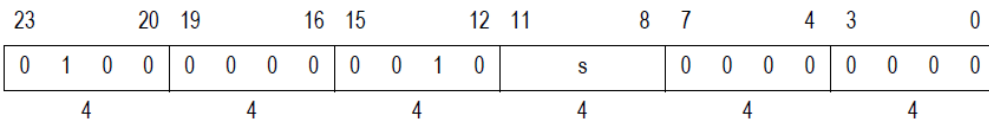
```
SAR ← 32 - (01AR[s]1..0103)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.309 SSA8L—Set Shift Amount for LE Byte Shift

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SSA8L as
```

Description

`SSA8L` sets the shift amount register (`SAR`) for a right shift by multiples of eight (for example, for little-endian (LE) byte alignment). The right shift amount is the two least significant bits of address register `as` multiplied by eight, and is written to `SAR`. `SSA8L` is similar to `SSR`, except the shift amount is multiplied by eight.

`SSA8L` is typically used to set up for an `SRC` instruction to shift bytes. It may be used with little-endian byte ordering to extract a 32-bit value from a non-aligned byte address.

Operation

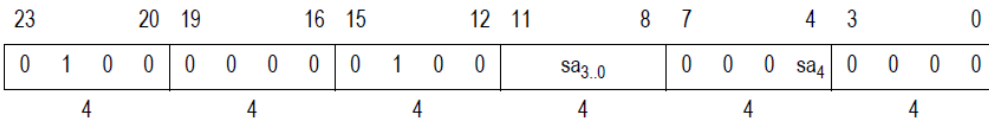
```
SAR ← 0 | AR[s]1..0 | 03
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.310 SSAI—Set Shift Amount Immediate

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SSAI 0..31
```

Description

`SSAI` sets the shift amount register (`SAR`) to a constant. The shift amount `sa` field is split, with bits `3..0` in bits `11..8` of the instruction word, and bit `4` in bit `4` of the instruction word.

Because immediate forms exist of most shifts (`SLLI`, `SRLI`, `SRAI`), this is primarily useful to set the shift amount for `SRC`.

Operation

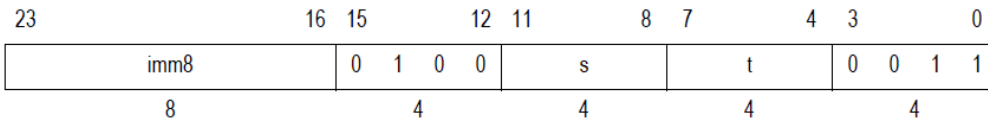
```
SAR ← 0 | sa
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))

8.3.311 SSI—Store Single Immediate

Instruction Word (RRI8)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SSI ft, as, 0..1020
```

Description

SSI is a 32-bit store from floating-point register `ft` to memory. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The data to be stored is taken from the contents of floating-point register `ft` and written to memory at the physical address.

If the Region Translation Option ([Region Translation Option](#) on page 202 or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, SSI calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

```
vAddr ← AR[s] + (022||imm8||02)
Store32 (vAddr, FR[t])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.312 SSIP—Store Single Immediate Post-Increment

Instruction Word (RRI8)

23	16	15	12	11	8	7	4	3	0		
imm8		1	1	0	0	s	t	0	0	1	1
8		4		4		4		4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SSIP ft, as, 0..1020
```

Description

SSIP is a 32-bit store from floating-point register `ft` to memory with base address register post-increment. The virtual address is taken from the contents of address register `as`. The data to be stored is taken from the contents of floating-point register `ft` and written to memory at the physical address. The sum of the virtual address and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two is written back to address register `as`. The increment can specify multiples of four from zero to 1020.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

SSIP calculates the increment of address register `as` using the `imm8` field of the instruction word times four. Therefore, the machine-code increment is in terms of 32-bit (4 byte) units. However, the assembler expects a byte increment and encodes this into the instruction by dividing by four.

Operation

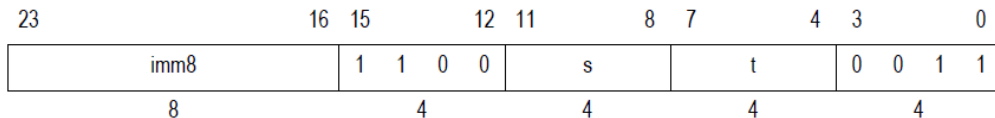
```
vAddr ← AR[s]
Store32 (vAddr, FR[t])
AR[s] ← vAddr + (022||imm8||02)
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.313 SSIU—Store Single Immediate Update

Instruction Word (RR18)



Required Configuration Option

Floating-Point 2000 Coprocessor Option (See [Floating-Point 2000 Coprocessor Option](#))

Assembler Syntax

```
SSIU ft, as, 0..1020
```

Description

SSIU is a 32-bit store from floating-point register `ft` to memory with base address register update. It forms a virtual address by adding the contents of address register `as` and an 8-bit zero-extended constant value encoded in the instruction word shifted left by two. Therefore, the offset can specify multiples of four from zero to 1020. The data to be stored is taken from the contents of floating-point register `ft` and written to memory at the physical address. The virtual address is written back to address register `as`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent

memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Assembler Note

To form a virtual address, SSIU calculates the sum of address register `as` and the `imm8` field of the instruction word times four. Therefore, the machine-code offset is in terms of 32-bit (4 byte) units. However, the assembler expects a byte offset and encodes this into the instruction by dividing by four.

Operation

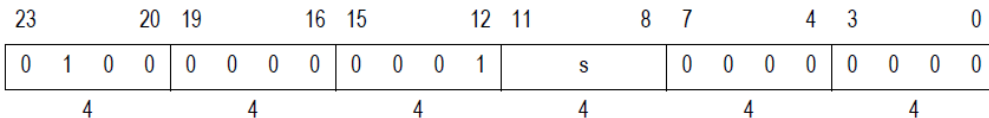
```
vAddr ← AR[s] + (022||imm8||02)
Store32 (vAddr, FR[t])
AR[s] ← vAddr
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.314 SSL—Set Shift Amount for Left Shift

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SSL as
```

Description

SSL sets the shift amount register (`SAR`) for a left shift (for example, `SLL`). The left shift amount is the 5 least significant bits of address register `as`. 32 minus this amount is written to `SAR`.

Using 32 minus the left shift amount causes a right funnel shift, and swapped high and low input operands to perform a left shift.

Operation

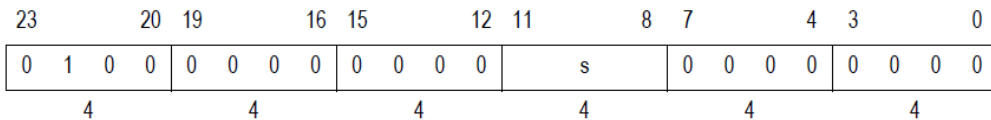
```
sa ← AR[s]4..0
SAR ← 32 - (0!sa)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.315 SSR—Set Shift Amount for Right Shift

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SSR as
```

Description

SSR sets the shift amount register (*SAR*) for a right shift (for example, *SRL*, *SRA*, or *SRC*). The least significant five bits of address register *as* are written to *SAR*. The most significant bit of *SAR* is cleared. This instruction is similar to a *WSR.SAR*, but differs in that only *AR[s]4..0* is used, instead of *AR[s]5..0*.

Operation

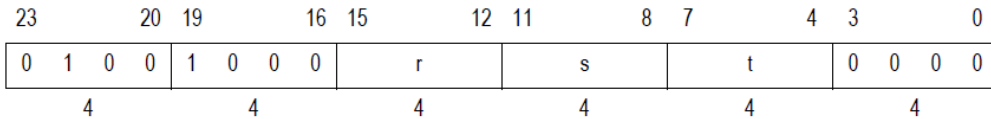
```
sa ← AR[s]4..0
SAR ← 0!sa
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.316 SSX—Store Single Indexed

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SSX fr, as, at
```

Description

SSX is a 32-bit store from floating-point register `fr` to memory. It forms a virtual address by adding the contents of address register `as` and the contents of address register `at`. The data to be stored is taken from the contents of floating-point register `fr` and written to memory at the physical address.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

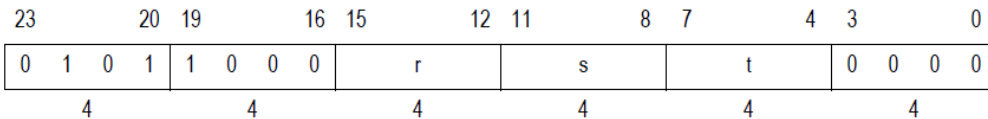
```
vAddr ← AR[s] + (AR[t])  
Store32 (vAddr, FR[r])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.317 SSXP—Store Single Indexed Post-Increment

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SSXP fr, as, at
```

Description

SSXP is a 32-bit store from floating-point register `fr` to memory with base address register post-increment. The virtual address is taken from the contents of address register `as`. The data to be stored is taken from the contents of floating-point register `fr` and written to memory at the physical address. The sum of the virtual address and the contents of address register `at` is written back to address register `as`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

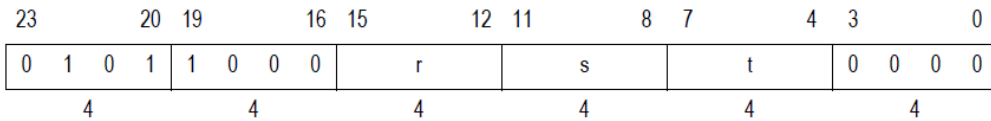
```
vAddr ← AR[s]  
Store32 (vAddr, FR[r])  
AR[s] ← vAddr + (AR[t])
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.318 SSXU—Store Single Indexed Update

Instruction Word (RRR)



Required Configuration Option

Floating-Point 2000 Coprocessor Option (See [Floating-Point 2000 Coprocessor Option](#))

Assembler Syntax

```
SSXU fr, as, at
```

Description

SSXU is a 32-bit store from floating-point register `fr` to memory with base address register update. It forms a virtual address by adding the contents of address register `as` and the contents of address register `at`. The data to be stored is taken from the contents of floating-point register `fr` and written to memory at the physical address. The virtual address is written back to address register `as`.

If the Region Translation Option ([Region Translation Option](#) on page 202) or the MMU Option ([MMU Option](#) on page 217) is enabled, the virtual address is translated to the physical address. If not, the physical address is identical to the virtual address. If the translation or memory reference encounters an error (for example, protection violation or non-existent memory), the processor raises one of several exceptions (see [The Exception Cause Register \(EXCCAUSE\) under the Exception Option 2](#) on page 135).

Without the Unaligned Exception Option ([Unaligned Exception Option](#) on page 148), the two least significant bits of the address are ignored. A reference to an address that is not 0 mod 4 produces the same result as a reference to the address with the least significant bits cleared. With the Unaligned Exception Option, such an access raises an exception.

Operation

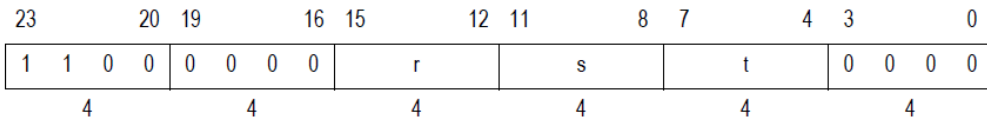
```
vAddr ← AR[s] + (AR[t])  
Store32 (vAddr, FR[r])  
AR[s] ← vAddr
```

Exceptions

- Memory Store Group (see [Memory Store Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.319 SUB—Subtract

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SUB ar, as, at
```

Description

SUB calculates the two's complement 32-bit difference of address registers `as` and `at`. The low 32 bits of the difference are written to address register `ar`. Arithmetic overflow is not detected.

Operation

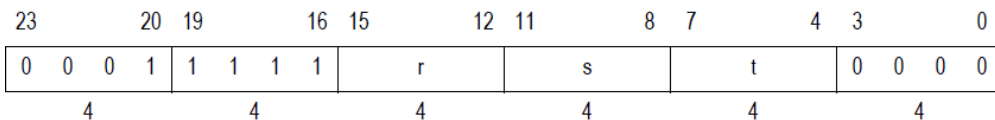
```
AR[r] ← AR[s] - AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.320 SUB.D—Subtract Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SUB.D fr, fs, ft
```

Description

`SUB.D` computes the IEEE754 double-precision difference of the contents of floating-point registers `fs` and `ft` and writes the result to floating-point register `fr`.

Operation

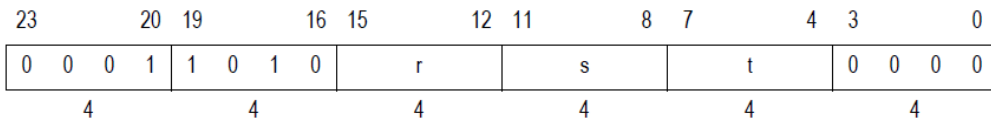
```
FR[r] ← FR[s] -D FR[t]
FSR[StatusFlags: VOI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.321 SUB.S—Subtract Single

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
SUB.S fr, fs, ft
```

Description

`SUB.S` computes the IEEE754 single-precision difference of the contents of floating-point registers `fs` and `ft` and writes the result to floating-point register `fr`.

Operation

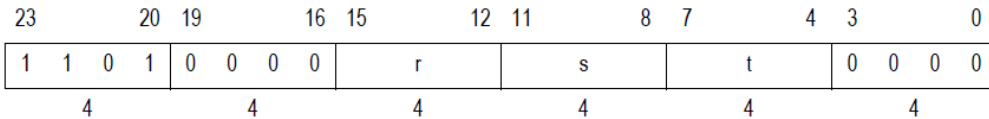
```
FR[r] ← FR[s] -S FR[t]
FSR[StatusFlags: VOI] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.322 SUBX2—Subtract with Shift by 1

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SUBX2 ar, as, at
```

Description

SUBX2 calculates the two's complement 32-bit difference of address register `as` shifted left by 1 bit and address register `at`. The low 32 bits of the difference are written to address register `ar`. Arithmetic overflow is not detected.

SUBX2 is frequently used as part of sequences to multiply by small constants.

Operation

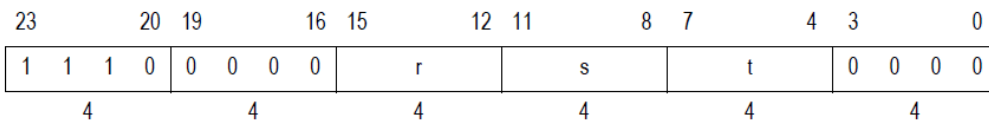
```
AR[r] ← (AR[s]₃₀..₀!0) - AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.323 SUBX4—Subtract with Shift by 2

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SUBX4 ar, as, at
```

Description

SUBX4 calculates the two's complement 32-bit difference of address register `as` shifted left by two bits and address register `at`. The low 32 bits of the difference are written to address register `ar`. Arithmetic overflow is not detected.

SUBX4 is frequently used as part of sequences to multiply by small constants.

Operation

$$AR[r] \leftarrow (AR[s]_{29..0} \ll 2) - AR[t]$$

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.324 SUBX8—Subtract with Shift by 3

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	1	1	0	0	0	0	r	s	t	0	0	0	0	
4				4				4		4		4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
SUBX8 ar, as, at
```

Description

SUBX8 calculates the two's complement 32-bit difference of address register `as` shifted left by three bits and address register `at`. The low 32 bits of the difference are written to address register `ar`. Arithmetic overflow is not detected.

SUBX8 is frequently used as part of sequences to multiply by small constants.

Operation

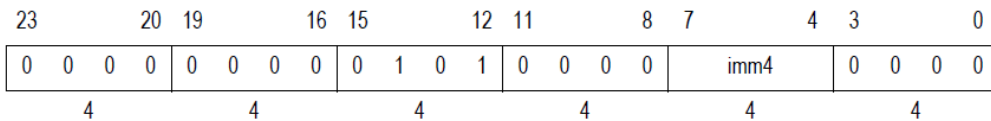
```
AR[r] ← (AR[s]28..0!03) - AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.325 SYSCALL—System Call

Instruction Word (RRR)



Required Configuration Option

Exception Option 2 (See [Exception Option 2](#) on page 126)

Assembler Syntax

```
SYSCALL  
SYSCALL imm4
```

Description

When executed, the `SYSCALL` instruction raises a system-call exception. Under the Exception Option 2 it redirects to an exception vector (see [Exception Option 2](#) on page 126) with `EPC[1]` containing the address of the `SYSCALL` and `ICOUNT` is not incremented. Since, in either case, a `SYSCALL` instruction never completes, the system call handler should add 3 to the appropriate `EPC` before returning from the exception to continue execution.

The program may pass parameters to the system-call handler in the registers. Under the Exception Option 2 there are no bits in `SYSCALL` instruction reserved for this purpose, the instruction does not take an argument, and the `imm4` field must be zero. See [System Calls](#) on page 698 “System Calls” for a description of software conventions for system call parameters.

Operation

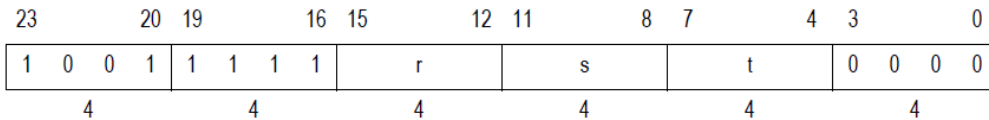
```
Exception (SyscallCause) if Exception Option 2
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- GenExcep(SyscallCause) if Exception Option 2

8.3.326 TRUNC.D—Truncate Double to Fixed

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
TRUNC.D ar, fs, 0..15
```

Description

TRUNC.D converts the contents of floating-point register `fs` from double-precision to signed integer format, rounding toward 0. The double-precision value is first scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 2.0, 4.0, ..., 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0`, and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $\geq 2^{31}$), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (scaled argument $\leq -2^{31} - 1$) or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

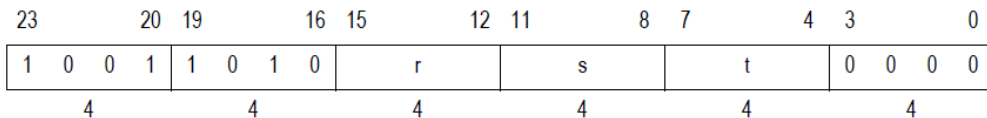
```
AR[r] ← truncD(FR[s] ×D powD(2.0,t))
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.327 TRUNC.S—Truncate Single to Fixed

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
TRUNC.S ar, fs, 0..15
```

Description

`TRUNC.S` converts the contents of floating-point register `fs` from single-precision to signed integer format, rounding toward 0. The single-precision value is first scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 2.0, 4.0, ..., 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0`, and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $\geq 2^{31}$), positive infinity, or NaN, `32'h7fffffff` is returned; for negative overflow (scaled argument $\leq -2^{31} - 1$) or negative infinity, `32'h80000000` is returned. The result is written to address register `ar`.

Operation

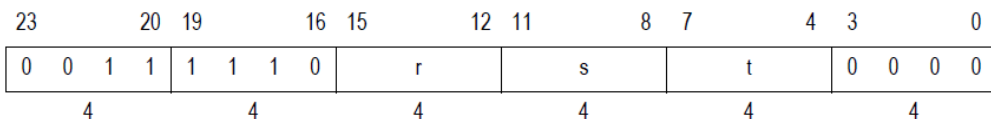
```
AR[r] ← truncs(FR[s] ×s pows(2.0,t))
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.328 UEQ.D—Compare Double Unordered or Equal

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
UEQ.D br, fs, ft
```

Description

`UEQ.D` compares the double-precision values in floating-point registers `fs` and `ft`. If the values are equal or unordered then Boolean register `br` is set to 1, otherwise `br` is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are unordered if either of them is a NaN. Like most floating-point instructions `UEQ.D` sets the Invalid Operation flag if either input is a Signalling NaN.

Operation

```
BRr ← isNaND(FR[s]) or isNaND(FR[t]) or (FR[s] =D FR[t])  
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.329 UEQ.S—Compare Single Unordered or Equal

Instruction Word (RRR)

23	19	16	15	12	11	8	7	4	3	0					
0	0	1	1	1	0	1	1	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
UEQ.S br, fs, ft
```

Description

`UEQ.S` compares the single-precision values in floating-point registers `fs` and `ft`. If the values are equal or unordered then Boolean register `br` is set to 1, otherwise `br` is set to 0. According to IEEE754, +0 and -0 compare as equal. IEEE754 floating-point values are unordered if either of them is a NaN. Like most floating-point instructions `UEQ.S` sets the Invalid Operation flag if either input is a Signalling NaN.

Operation

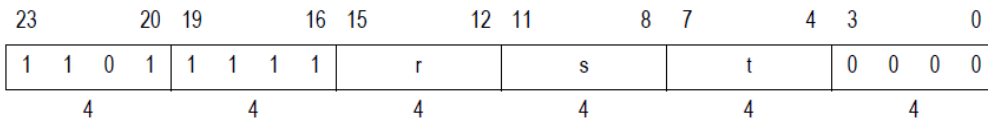
```
BRr ← isNaNS(FR[s]) or isNaNS(FR[t]) or (FR[s] =S FR[t])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.330 UFLOAT.D—Convert Unsigned Fixed to Double

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
UFLOAT.D fr, as, 0..15
```

Description

UFLOAT.D converts the contents of address register `as` from unsigned integer to double-precision format. The converted integer value is then scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 0.5, 0.25, ..., $1.0 \div_D 32768.0$. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0`, and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. The result is written to floating-point register `fr`.

Operation

```
FR[r] ← ufloatD(AR[s] ×D powD(2.0, -t))
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.331 UFLOAT.S—Convert Unsigned Fixed to Single

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
1	1	0	1	1	0	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
UFLOAT.S fr, as, 0..15
```

Description

UFLOAT.S converts the contents of address register `as` from unsigned integer to single-precision format, rounding according to the current rounding mode. The converted integer value is then scaled by a power of two constant value encoded in the `t` field, with 0..15 representing 1.0, 0.5, 0.25, ..., $1.0 \div_s 32768.0$. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for `t=0`, and moves to the left as `t` increases until for `t=15` there are 15 fractional bits represented in the fixed point number. The result is written to floating-point register `fr`.

Operation

```
FR[r] ← ufloats(AR[s]) ×s pows(2.0, -t)  
FSR[StatusFlags: I] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.332 ULE.D—Compare Double Unord or Less Than or Equal

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	1	1	1	1	1	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ULE.D br, fs, ft
```

Description

ULE.D compares the double-precision values in floating-point registers *fs* and *ft*. If the contents of *fs* are less than or equal to or unordered with the contents of *ft*, then Boolean register *br* is set to 1, otherwise *br* is set to 0. IEEE754 specifies that +0 and -0 compare as equal. IEEE754 floating-point values are unordered if either of them is a NaN. Like most floating-point instructions ULE.D sets the Invalid Operation flag if either input is a Signalling NaN.

Operation

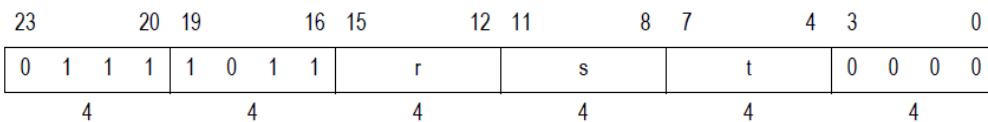
```
BRr ← isNaND(FR[s]) or isNaND(FR[t]) or (FR[s] ≤D FR[t])  
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.333 ULE.S—Compare Single Unord or Less Than or Equal

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ULE.S br, fs, ft
```

Description

ULE.S compares the single-precision values in floating-point registers *fs* and *ft*. If the contents of *fs* are less than or equal to or unordered with the contents of *ft*, then Boolean

register `br` is set to 1, otherwise `br` is set to 0. IEEE754 specifies that +0 and -0 compare as equal. IEEE754 floating-point values are unordered if either of them is a NaN. Like most floating-point instructions `ULE.S` sets the Invalid Operation flag if either input is a Signalling NaN.

Operation

```
BRr ← isNaNS(FR[s]) or isNaNS(FR[t]) or (FR[s] ≤S FR[t])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.334 ULT.D—Compare Double Unordered or Less Than

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	1	0	1	1	1	0	r	s	t	0	0	0			
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ULT.D br, fs, ft
```

Description

`ULT.D` compares the double-precision values in floating-point registers `fs` and `ft`. If the contents of `fs` are less than or unordered with the contents of `ft`, then Boolean register `br` is set to 1, otherwise `br` is set to 0. IEEE754 specifies that +0 and -0 compare as equal. IEEE754 floating-point values are unordered if either of them is a NaN. Like most floating-point instructions `ULT.D` sets the Invalid Operation flag if either input is a Signaling NaN.

Operation

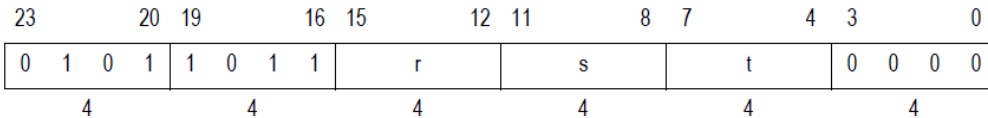
```
BRr ← isNaND(FR[s]) or isNaND(FR[t]) or (FR[s] <D FR[t])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.335 ULT.S—Compare Single Unordered or Less Than

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
ULT.S br, fs, ft
```

Description

ULT.S compares the single-precision values in floating-point registers `fs` and `ft`. If the contents of `fs` are less than or unordered with the contents of `ft`, then Boolean register `br` is set to 1, otherwise `br` is set to 0. IEEE754 specifies that +0 and -0 compare as equal. IEEE754 floating-point values are unordered if either of them is a NaN. Like most floating-point instructions ULT.S sets the Invalid Operation flag if either input is a Signaling NaN.

Operation

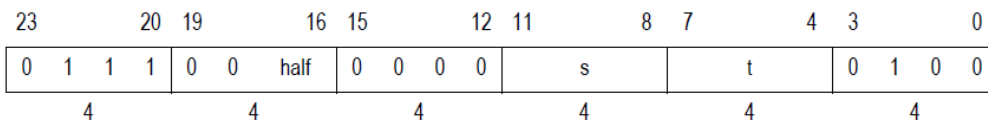
```
BRr ← isNaNs(FR[s]) or isNaNs(FR[t]) or (FR[s] <_s FR[t])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.336 UMUL.AA.*—Unsigned Multiply

Instruction Word (RRR)



Required Configuration Option

MAC16 Option (See [MAC16 Option](#) on page 91)

Assembler Syntax

```
UMUL.AA.* as, at
```

Where * expands as follows:

```
UMUL.AA.LL - for (half=0)  
UMUL.AA.HL - for (half=1)  
UMUL.AA.LH - for (half=2)  
UMUL.AA.HH - for (half=3)
```

Description

UMUL.AA.* performs an unsigned multiply of half of each of the address registers *as* and *at*, producing a 32-bit result. The result is zero-extended to 40 bits and written to the MAC16 accumulator.

Operation

```
m1 ← if half0 then AR[s]31..16 else AR[s]15..0  
m2 ← if half1 then AR[t]31..16 else AR[t]15..0  
ACC ← (024||m1) × (024||m2)
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.337 UN.D—Compare Double Unordered

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	0	0	1	1	1	0	r	s	t	0	0	0	0		
4				4				4				4			

Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
UN.D br, fs, ft
```


Description

`UN.D` sets Boolean register `br` to 1 if the double-precision values in either floating-point register `fs` or `ft` is a IEEE754 NaN; otherwise `br` is set to 0. Like most floating-point instructions `UN.D` sets the Invalid Operation flag if either input is a Signalling NaN.

Operation

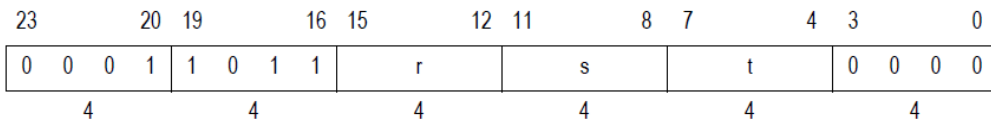
```
BRr ← isNaND(FR[s]) or isNaND(FR[t])
FSR[StatusFlags: V] ← Or in update
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.338 UN.S—Compare Single Unordered

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
UN.S br, fs, ft
```

Description

`UN.S` sets Boolean register `br` to 1 if the single-precision values in either floating-point register `fs` or `ft` is a IEEE754 NaN; otherwise `br` is set to 0. Like most floating-point instructions `UN.S` sets the Invalid Operation flag if either input is a Signaling NaN.

Operation

```
BRr ← isNaNS(FR[s]) or isNaNS(FR[t])
FSR[StatusFlags: V] ← Or in update
```

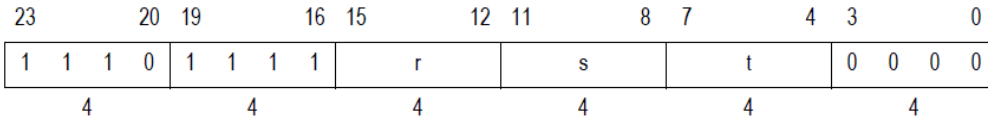
Exceptions

- EveryInst Group (see [EveryInst Group](#))

- Coprocessor Group (see [Coprocessor Group](#))

8.3.339 UTRUNC.D—Truncate Double to Fixed Unsigned

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
UTRUNC.D ar, fs, 0..15
```

Description

UTRUNC.D converts the contents of floating-point register *fs* from double-precision to unsigned integer format, rounding toward 0. The double-precision value is first scaled by a power of two constant value encoded in the *t* field, with 0..15 representing 1.0, 2.0, 4.0, ..., 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for *t*=0, and moves to the left as *t* increases until for *t*=15 there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $\geq 2^{32}$), positive infinity, or NaN, 32'hffffffff is returned; for negative numbers or negative infinity, the UTRUNC.D instruction returns exactly the same answer as the TRUNC.D instruction. The result is written to address register *ar*.

Operation

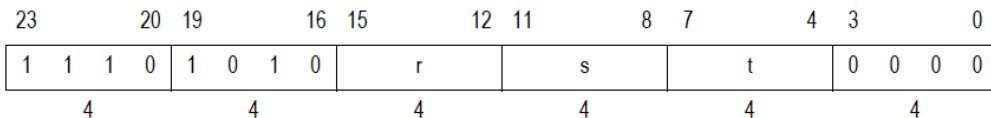
```
AR[r] ← utruncD(FR[s] ×D powD(2.0,t))
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.340 UTRUNC.S—Truncate Single to Fixed Unsigned

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
UTRUNC.S ar, fs, 0..15
```

Description

UTRUNC.S converts the contents of floating-point register *fs* from single-precision to unsigned integer format, rounding toward 0. The single-precision value is first scaled by a power of two constant value encoded in the *t* field, with 0..15 representing 1.0, 2.0, 4.0, ..., 32768.0. The scaling allows for a fixed point notation where the binary point is at the right end of the integer for *t*=0, and moves to the left as *t* increases until for *t*=15 there are 15 fractional bits represented in the fixed point number. For positive overflow (scaled argument $\geq 2^{32}$), positive infinity, or NaN, `32'hffffffff` is returned; for negative numbers or negative infinity, the UTRUNC.S instruction returns exactly the same answer as the TRUNC.S instruction. The result is written to address register *ar*.

Operation

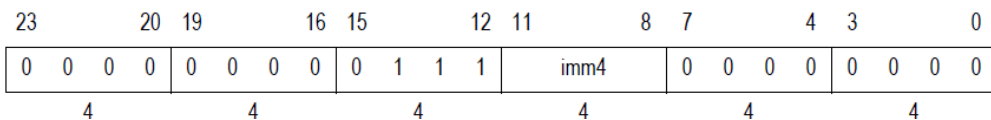
```
AR[r] ← utruncs(FR[s] ×s pows(2.0,t))
FSR[StatusFlags: VI] ← Or in update
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.341 WAITI—Wait for Interrupt

Instruction Word (RRR)



Required Configuration Option

Interrupt Option (See [Interrupt Option](#) on page 151)

Assembler Syntax

```
WAITI 0..15
```

Description

`WAITI` modifies interrupt masking and then, on some Xtensa ISA implementations, suspends processor operation until an interrupt occurs. `WAITI` is typically used in an idle loop to reduce power consumption. `CCOUNT` continues to increment during suspended operation, and a `CCOMPARE` interrupt will wake the processor.

Under the Interrupt Option (and Exception Option 2), the method of modifying interrupt masking is to set the interrupt level in `PS.INTLEVEL` to `imm4`.

When an interrupt is taken during suspended operation, `EPC` will have the address of the instruction following `WAITI`. An implementation is not required to enter suspended operation and may leave suspended operation and continue execution at the following instruction at any time. Usually, therefore, the `WAITI` instruction should be within a loop.

The combination of modifying interrupt masking and suspending operation avoids a race condition where an interrupt between the interrupt masking modification and the suspension of operation would be ignored until a second interrupt occurred.

`WAITI` is a privileged instruction.

Operation

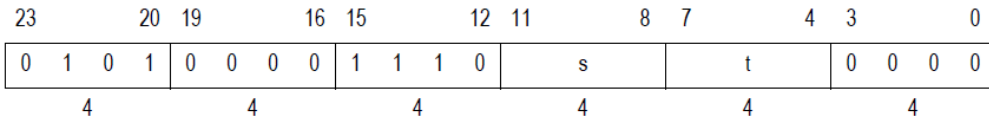
```
if Interrupt Option & CRING ≠ 0 then
    Exception (PrivilegedCause)
elseif Exception Option and PS.Ring ≠ 0 then
    Exc(User executes privileged instruction)
elseif Interrupt Option then
    PS.INTLEVEL ← imm4
elseif Exception Option then
    PS.DI ← 0
endif
```

Exceptions

- EveryInst Group (see [EveryInst Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.342 WDTLB—Write Data TLB Entry

Instruction Word (RRR)



Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
WDTLB at, as
```

Description

WDTLB uses the contents of address register *as* to specify a data TLB entry and writes the contents of address register *at* into that entry. See [Options for Memory Protection and Translation](#) on page 183 for information on the address and result register formats for specific memory protection and translation options. The point at which the data TLB write is effected is implementation-specific. Any translation that would be affected by this write before the execution of a *DSYNC* instruction is therefore undefined.

WDTLB is a privileged instruction.

Operation

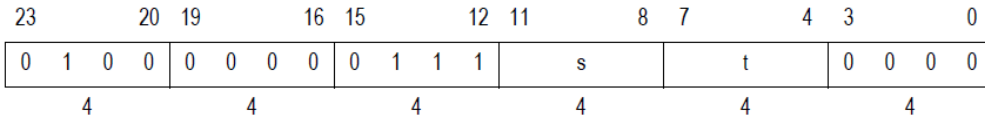
```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    (vpn, ei, wi) ← SplitDataTLBEntrySpec(AR[s])
    (ppn, sr, ring, ca) ← SplitDataEntry(wi, AR[t])
    DataTLB[wi][ei].ASID ← ASID(ring)
    DataTLB[wi][ei].VPN ← vpn
    DataTLB[wi][ei].PPN ← ppn
    DataTLB[wi][ei].SR ← sr
    DataTLB[wi][ei].CA ← ca
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.343 WER—Write External Register

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
WER at, as
```

Description

`WER` writes one of a set of "External Registers". It is in some ways similar to the `WSR.*` instruction except that the registers being written are not defined by the Xtensa ISA and are conceptually outside the processor core. They are written through processor ports.

Address register `as` is used to determine which register is to be written and address register `at` provides the write data. When no External Register is addressed by the value in address register `as`, no write occurs. The entire address space is reserved for use by Cadence. `RER` and `WER` are managed by the processor core so that the requests appear on the processor ports in program order. External logic is responsible for extending that order to the registers themselves.

In older implementations, `WER` is a privileged instruction while in newer implementations, parts of the address space can be privileged as determined by the `ERACCESS` Special Register ([page 340](#)).

Operation

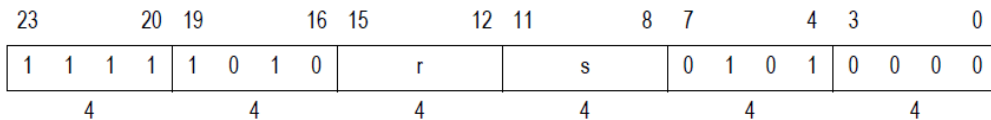
```
if (CRING ≠ 0 & PrivilegedAddressRegion) then
    Exception (ExternalRegPrivilegeCause)
else
    Write External Register as defined outside the processor.
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.344 WFR—Move AR to FR

Instruction Word (RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
WFR fr, as
```

Description

WFR moves the contents of address register *as* to floating-point register *fr*. The move is non-arithmetic; no floating-point exceptions are raised. When double-precision floating-point is supported, the move is to the lower half of the floating-point register.

Operation

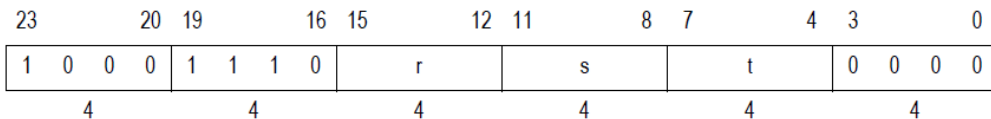
```
FR[r] ← AR[s]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.345 WFRD—Move AR to FR Double

Instruction Word(RRR)



Required Configuration Option

Floating-Point Coprocessor Option (See [Floating-Point Coprocessor Option](#) on page 99)

Assembler Syntax

```
WFRD fr, as, at
```

Description

WFRD moves the contents of address register *as* concatenated with the contents of address register *at* to floating-point register *fr*. The move is non-arithmetic; no floating-point exceptions are raised.

Operation

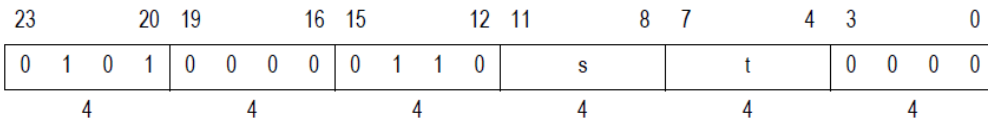
```
FR[r] ← AR[s]||AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Coprocessor Group (see [Coprocessor Group](#))

8.3.346 WITLB—Write Instruction TLB Entry

Instruction Word (RRR)



Required Configuration Option

Region Protection Option (see [Region Protection Option](#) on page 196) or MMU Option (see [MMU Option](#) on page 217)

Assembler Syntax

```
WITLB at, as
```

Description

WITLB uses the contents of address register *as* to specify an instruction TLB entry and writes the contents of address register *at* into that entry. See [Options for Memory Protection and Translation](#) on page 183 for information on the address and result register formats for specific memory protection and translation options. The point at which the instruction TLB write is effected is implementation-specific. Any translation that would be affected by this write before the execution of an `ISYNC` instruction is therefore undefined.

WITLB is a privileged instruction.

Operation

```
if CRING ≠ 0 then
    Exception (PrivilegedCause)
```



```

else
  (vpn, ei, wi) ← SplitInstTLBEntrySpec(AR[s])
  (ppn, sr, ring, ca) ← SplitInstEntry(wi, AR[t])
  InstTLB[wi][ei].ASID ← ASID(ring)
  InstTLB[wi][ei].VPN ← vpn
  InstTLB[wi][ei].PPN ← ppn
  InstTLB[wi][ei].SR ← sr
  InstTLB[wi][ei].CA ← ca
endif

```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.347 WPTLB—Write Protection TLB Entry

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0						
0	1	0	1	0	0	0	0	1	1	1	0	s	t	0	0	0	0
4				4				4				4					

Required Configuration Option

Memory Protection Unit Option ([Memory Protection Unit Option](#) on page 205)

Assembler Syntax

```
WPTLB at, as
```

Description

WPTLB uses the contents of address registers `as` and `at` to specify a protection TLB entry and the information to be written to it. See [Formats for Writing Memory Protection Unit Option TLB Entries](#) on page 211 for information on the register formats. The point at which the TLB write is effected is implementation-specific. Any translation for instruction access that would be affected by this write before the execution of an `ISYNC` instruction is therefore undefined.

WPTLB is a privileged instruction.

Operation

```

if CRING ≠ 0 then
  Exception (PrivilegedCause)
else
  ProtectionTLBWrite(AR[s], AR[t])
endif

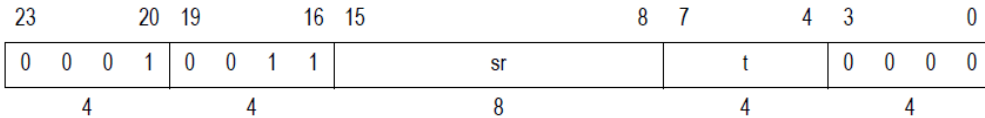
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.348 WSR.*—Write Special Register

Instruction Word (RRR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
WSR.* at
WSR at, *
WSR at, 0..255
```

Description

WSR.* writes the special registers that are described in [Processor Control Instructions](#) on page 70. See [Special Registers](#) on page 272 for more detailed information on the operation of this instruction for each Special Register.

The contents of address register `at` are written to the special register designated by the 8-bit `sr` field of the instruction word. The name of the Special Register is used in place of the '*' in the assembler syntax above and the translation is made to the 8-bit `sr` field by the assembler.

WSR is an assembler macro for WSR.* that provides compatibility with the older versions of the instruction containing either the name or the number of the Special Register.

The point at which WSR.* to certain registers affects subsequent instructions is not always defined (SAR and ACC are exceptions). In these cases, the Special Register Tables in [Special Registers](#) on page 272 explain how to ensure the effects are seen by a particular point in the instruction stream (typically involving the use of one of the ISYNC, RSYNC, ESYNC, or DSYNC instructions). A WSR.* followed by an RSR.* to the same register should be separated with an ESYNC to guarantee the value written is read back. A WSR.PS followed by RSIL also requires an ESYNC.

WSR.* with Special Register numbers ≥ 64 is privileged. A WSR.* for an unconfigured register generally will raise an illegal instruction exception.

Operation

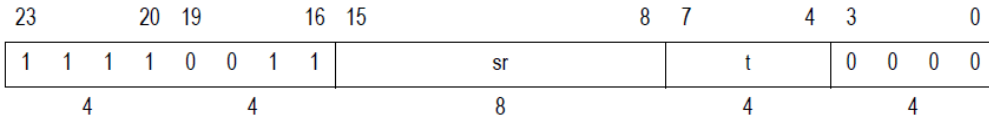
```
if sr > 64 and CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    see the Special Register Tables in Special Registers on page 272 on page 300
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2
- Privileged Instruction Group (see [Privileged Instruction Group](#))

8.3.349 WUR.*—Write User Register

Instruction Word (RSR)



Required Configuration Option

No Option - instructions created from the TIE language (See [Coprocesor Context Switch](#) on page 150)

Assembler Syntax

```
WUR.* at
WUR at,*
```

Description

WUR.* writes TIE state that has been grouped into 32-bit quantities by the TIE user_register statement. The name in the user_register statement replaces the “*” in the instruction name and causes the correct register number to be placed in the st field of the encoded instruction. The contents of address register at are written to the TIE user_register designated by the 8-bit sr field of the instruction word.

WUR is an assembler macro for WUR.* that provides compatibility with the older version of the instruction.

Operation

```
user_register[sr] ← AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(Coprocessor*Disabled) if Exception Option 2 and Coprocessor Context Option

8.3.350 XOR—Bitwise Logical Exclusive Or

Instruction Word (RRR)

23	20	19	16	15	12	11	8	7	4	3	0				
0	0	1	1	0	0	0	0	r	s	t	0	0	0	0	
4				4				4				4			

Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77)

Assembler Syntax

```
XOR ar, as, at
```

Description

XOR calculates the bitwise logical exclusive or of address registers `as` and `at`. The result is written to address register `ar`.

Operation

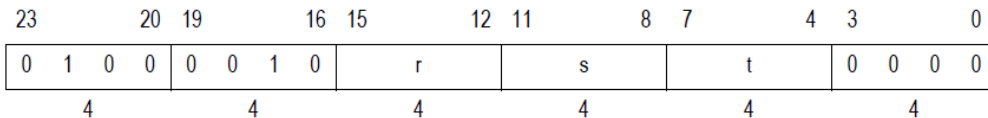
```
AR[r] ← AR[s] xor AR[t]
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.351 XORB—Boolean Exclusive Or

Instruction Word (RRR)



Required Configuration Option

Boolean Option (See [Boolean Option](#) on page 97)

Assembler Syntax

```
XORB br, bs, bt
```

Description

XORB performs the logical exclusive or of Boolean registers `bs` and `bt` and writes the result to Boolean register `br`.

When the sense of one of the source Booleans is inverted (0 → true, 1 → false), use an inverted test of the result. When the sense of both of the source Booleans is inverted, use a non-inverted test of the result.

Operation

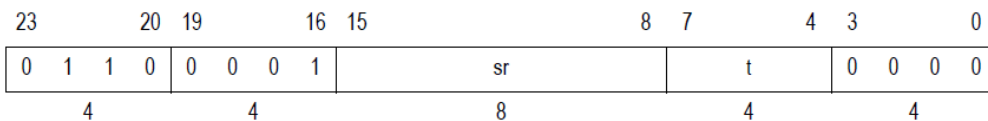
```
BRr ← BRs xor BRt
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))

8.3.352 XSR.*—Exchange Special Register

Instruction Word (RSR)



Required Configuration Option

Core Architecture (See [Core Architecture](#) on page 77) (added in T1040)

Assembler Syntax

```
XSR.* at
```

```
XSR at, *
XSR at, 0..255
```

Description

`XSR.*` simultaneously reads and writes the special registers that are described in [Processor Control Instructions](#) on page 70. See [Special Registers](#) on page 272 for more detailed information on the operation of this instruction for each Special Register.

The contents of address register `at` and the Special Register designated by the immediate in the 8-bit `sr` field of the instruction word are both read. The read address register value is then written to the Special Register, and the read Special Register value is written to `at`. The name of the Special Register is used in place of the "*" in the assembler syntax above and the translation is made to the 8-bit `sr` field by the assembler.

`XSR` is an assembler macro for `XSR.*`, which provides compatibility with the older versions of the instruction containing either the name or the number of the Special Register.

The point at which `XSR.*` to certain registers affects subsequent instructions is not always defined (`SAR` and `ACC` are exceptions). In these cases, the Special Register Tables in [Special Registers](#) on page 272 explain how to ensure the effects are seen by a particular point in the instruction stream (typically involving the use of one of the `ISYNC`, `RSYNC`, `ESYNC`, or `DSYNC` instructions). An `XSR.*` followed by an `RSR.*` to the same register should be separated with an `ESYNC` to guarantee the value written is read back. An `XSR.PS` followed by `RSIL` also requires an `ESYNC`. In general, the restrictions on `XSR.*` include the union of the restrictions of the corresponding `RSR.*` and `WSR.*`.

`XSR.*` with Special Register numbers ≥ 64 is privileged. An `XSR.*` for an unconfigured register generally will raise an illegal instruction exception.

Operation

```
if sr ≥ 64 and CRING ≠ 0 then
    Exception (PrivilegedCause)
else
    t0 ← AR[t]
    t1 ← see RSR frame of the Tables in Special Registers on page 272 see WSR frame of the
    Tables in Special Registers on page 272 on page 300 ← t0
    AR[t] ← t1
endif
```

Exceptions

- EveryInstR Group (see [EveryInstR Group](#))
- GenExcep(IllegalInstructionCause) if Exception Option 2
- Privileged Instruction Group (see [Privileged Instruction Group](#))

9. Instruction Formats and Opcodes

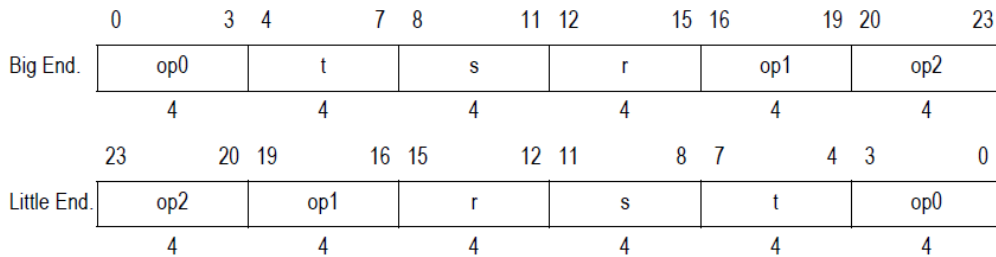
Topics:

- *Formats*
- *Instruction Fields*
- *Opcode Encodings*

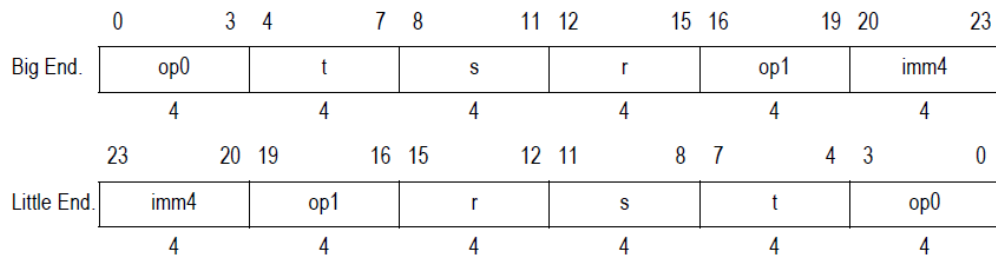
9.1 Formats

The following sections show the named opcode formats for instruction encodings. The field names in these formats are used in the opcode tables in *Opcode Maps* on page 661. The format names are used throughout this document. Each chart shows both big-endian and little-endian encodings with bits numbered appropriately for that endianness. The vertical bars in the formats indicate the points at which the opcode is separated, reversed in order, and reassembled to arrive at the opposite endianness format.

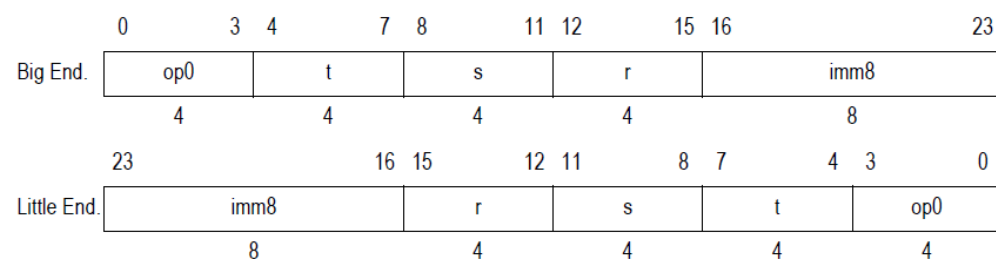
9.1.1 RRR



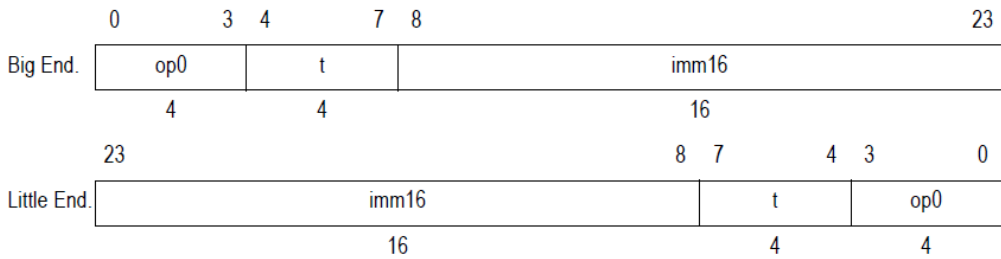
9.1.2 RRI4



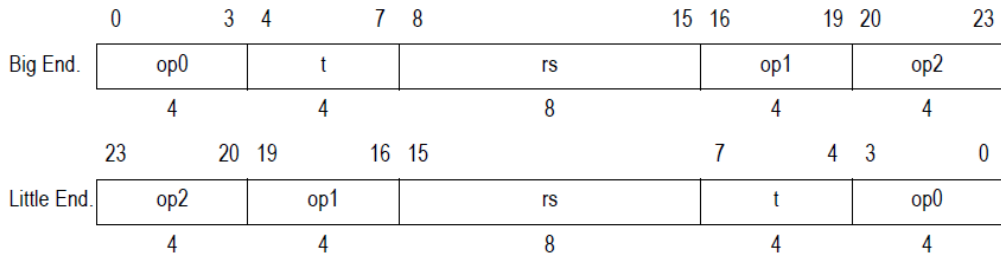
9.1.3 RRI8



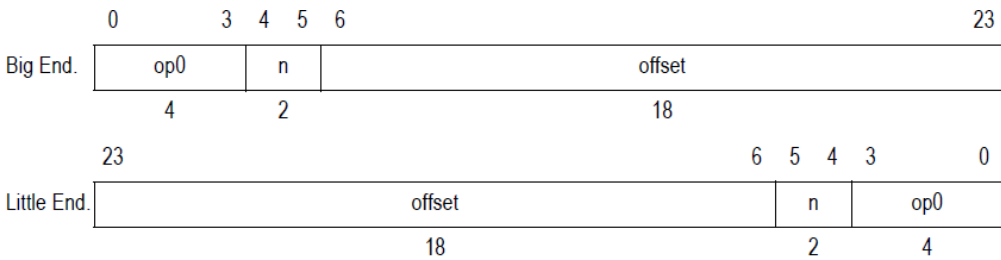
9.1.4 RI16



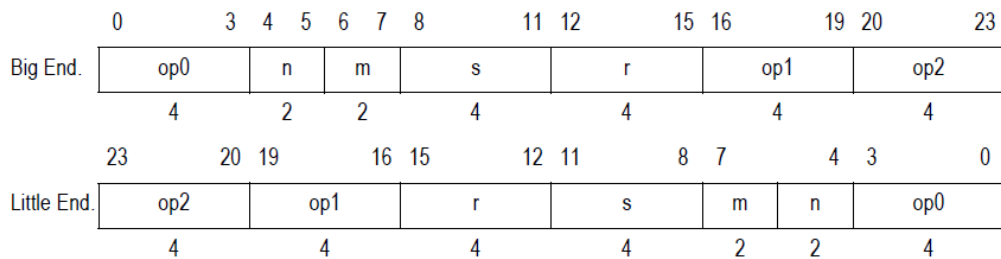
9.1.5 RSR



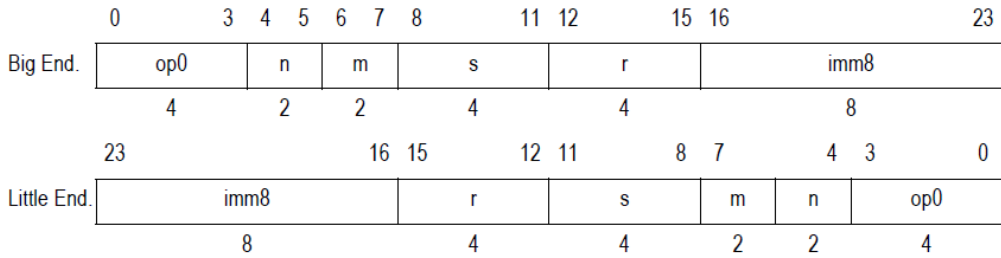
9.1.6 CALL



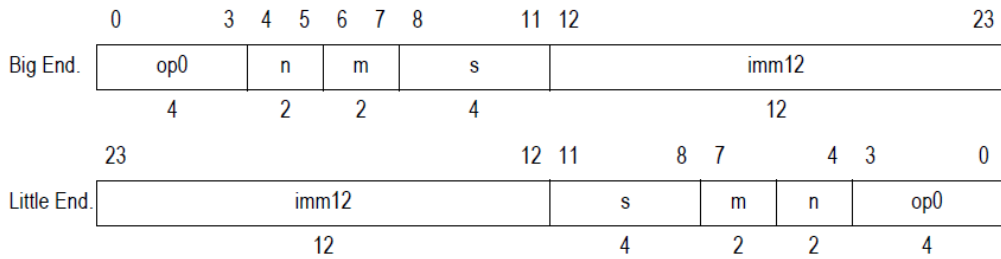
9.1.7 CALLX



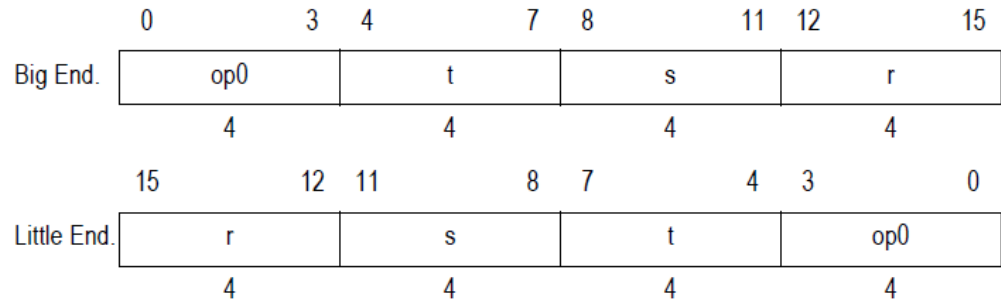
9.1.8 BRI8



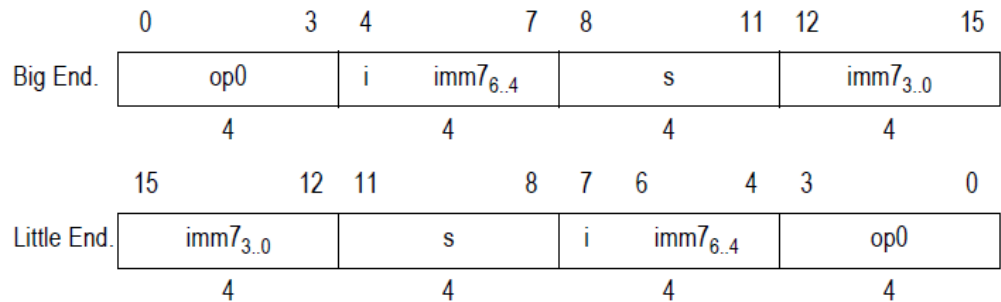
9.1.9 BRI12



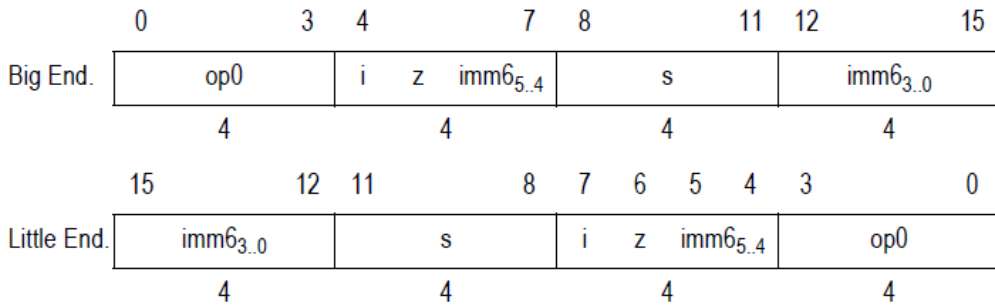
9.1.10 RRRN



9.1.11 RI7



9.1.12 RI6



9.2 Instruction Fields

Table 197: Uses Of Instruction Fields

Field	Definition
op0	Major opcode
op1	4-bit sub-opcode for 24-bit instructions
op2	4-bit sub-opcode for 24-bit instructions
r	AR target (result), BR target (result), 4-bit immediate, 4-bit sub-opcode
s	AR source, BR source AR target
t	AR target, BR target, AR source, BR source, 4-bit immediate, 4-bit sub-opcode
n	Register window increment, 2-bit sub-opcode, n 00 is used as a AR target on CALLn/CALLXn
m	2-bit sub-opcode

Field	Definition
i	1-bit sub-opcode
z	1-bit sub-opcode
imm6	6-bit immediate (PC-relative offset)
imm7	7-bit immediate (for <code>MOVI , N</code>)
imm8	8-bit immediate
imm12	12-bit immediate
imm16	16-bit immediate
offset	18-bit PC-relative offset

9.3 Opcode Encodings

The following tables show the instruction-field bit values assigned to specific opcodes. The following special notation is used:

- The table titles tell the name of the parent opcode and what table the parent is in, the formats for instructions in this table, and in parentheses at the end, what fields still vary for items listed in this table. In the upper left corner of the table is the field decoded in the table. Below it and to the right are templates which the field matches for the corresponding row or column.
- Non-italic opcodes are instructions. These have page numbers where the corresponding instruction is described in more detail.
- *Italics opcodes* are not instructions, but are parents to other opcodes. These have table numbers that show further decode into instructions or other parents to other opcodes.
- Some entries have further conditions after them such as (s=0), which means that the s field must be zero. All other opcodes are illegal; therefore another table seems unnecessary.
- The bit-range of opcodes that use more than one table entry is delimited by vertical bars.
- Subscripts on opcodes indicate the architectural option(s) in which the opcode is implemented. The subscripts and their associated architectural options are:
 - C—Instruction Cache or Data Cache Options
 - D—MAC16 Option
 - F—Floating-Point Coprocessor Option
 - I—32-Bit Integer Multiply/Divide Option

- L—Instruction or Data Cache Index Lock Option
- M—MMU Option
- N—Code Density (Narrow instructions) Option
- P—Coprocessor Option
- U—Miscellaneous Operations Option
- W—Windowed Registers Option
- X—Exception or Interrupt Options
- Y—Multiprocessor Synchronization Option

9.3.1 Opcode Maps

Table 198: Whole Opcode Space

op0	xx00	xx01	xx10	xx11
00xx	QRST — Table 283	L32R — Assembler Syntax	LSAI — Table 283	LSCIP — Table 283
01xx	MAC16D — Table 283	CALLN — Table 283	SI — Table 283	B — Table 283
10xx	L32I.N _N — Assembler Syntax	S32I.N _N — Assembler Syntax	ADD.N _N — Assembler Syntax	ADDI.N _N — Assembler Syntax
11xx	ST2 _N — Table 283	ST3 _N — Table 283	reserved	reserved

Table 199: QRST (from Table 7–283) Formats RRR, CALLX, and RSR (t, s, r, op2 vary)

op1	xx00	xx01	xx10	xx11
00xx	RST0 — Table 284	RST1 — Table 284	RST2 — Table 284	RST3 — Table 284
01xx	EXTUI — Assembler Syntax	CUST0 —	CUST1 —	
10xx	LSCX _p — Table 284	LSC4 — Table 284	FP0 _F — Table 284	FP1 _F — Table 284
11xx	reserved	reserved	DFP1 _F — Table 284	DFP0 _F — Table 284

Table 200: RST0 (from Table 7–284) Formats RRR and CALLX (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	ST0 — Table 285	AND — Assembler Syntax	OR — Assembler Syntax	XOR — Assembler Syntax
01xx	ST1 — Table 285	TLB — Table 285	RT0 — Table 285	reserved
10xx	ADD — Assembler Syntax	ADDX2 — Assembler Syntax	ADDX4 — Assembler Syntax	ADDX8 — Assembler Syntax
11xx	SUB — Assembler Syntax	SUBX2 — Assembler Syntax	SUBX4 — Assembler Syntax	SUBX8 — Assembler Syntax

Table 201: ST0 (from Table 7–285) Formats RRR and CALLX (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	SNM0 — Table 286	MOVSPW — Assembler Syntax	SYNC — Table 286	RFEI _x — Table 286
01xx	BREAK _x — Assembler Syntax	SYSIM _x — Table 286	RSIL _x — Assembler Syntax	WTLS — Table 286
10xx	ANY4 _p — Assembler Syntax	ALL4 _p — Assembler Syntax	ANY8 _p — Assembler Syntax	ALL8 _p — Assembler Syntax
11xx	reserved	reserved	reserved	reserved

Table 202: SNM0 (from Table 7–286) Format CALLX (n, s vary)

m	00	01	10	11
	ILL — Assembler Syntax (s,n=0)	reserved	JR — Table 287	CALLX — Table 287

Table 203: JR (from Table 7–287) Format CALLX (s varies)

n	00	01	10	11
	RET — Assembler Syntax (s=0)	RETW _w — Assembler Syntax (s=0)	JX — Assembler Syntax	reserved

Table 204: CALLX (from Table 7–287) Format CALLX (s varies)

n	00	01	10	11
	CALLX0 — <i>Assembler Syntax</i>	CALLX4 _W — <i>Assembler Syntax</i>	CALLX8 _W — <i>Assembler Syntax</i>	CALLX12 _W — <i>Assembler Syntax</i>

Table 205: SYNC (from Table 7–286) Format RRR (s varies)

t	xx00	xx01	xx10	xx11
00xx	SYNC0 — <i>Table 290</i>	RSYNC — <i>Assembler Syntax</i> (s=0)	ESYNC — <i>Assembler Syntax</i> (s=0)	DSYNC — <i>Assembler Syntax</i> (s=0)
01xx	reserved	reserved	reserved	reserved
10xx	EXCW — <i>Assembler Syntax</i> (s=0)	reserved	reserved	reserved
11xx	MEMW — <i>Assembler Syntax</i> (s=0)	EXTW — <i>Assembler Syntax</i> (s=0)	reserved	NOP — <i>Assembler Syntax</i> (s=0)

Table 206: SYNC0 (from Table 7–290) Format RRR

s	xx00	xx01	xx10	xx11
00xx	ISYNC — <i>Assembler Syntax</i>	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	FSYNC — <i>Assembler Syntax</i>			
11xx				

Table 207: RFEI (from Table 7–286) Format RRR (s varies)

t	xx00	xx01	xx10	xx11
00xx	RFET _x — <i>Table 292</i>	RFI _x — <i>Assembler Syntax</i>	RFM — <i>Table 292</i>	BLKSR — <i>Table 292</i>
01xx	reserved	reserved	reserved	reserved

t	xx00	xx01	xx10	xx11
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 208: SYSIM (from Table 7–286) Format RRR (t varies)

s	xx00	xx01	xx10	xx11
00xx	SYSCALL _x — <i>Assembler Syntax</i> (t=0)	SIMCALL _x — <i>Assembler Syntax</i> (t=0)	HALT _x — <i>Assembler Syntax</i>	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 209: WTLS (from Table 7–286) Format RRR (s varies)

t	xx00	xx01	xx10	xx11
00xx	WAIT _x — <i>Assembler Syntax</i>	reserved	reserved	reserved
01xx	CSRPTST <i>CSRPTST—CSR Parity Error Test</i>			
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	LDDR32.P — <i>Assembler Syntax</i>	SDDR32.P — <i>Assembler Syntax</i>

Table 210: BLKSR (from Table 7–292) Format RRR (no bits vary)

t	xx00	xx01	xx10	xx11
00xx	PFEND.A — <i>Assembler Syntax</i>	PFEND.O — <i>Assembler Syntax</i>	PFNXT.F — <i>Assembler Syntax</i>	PFWAIT.A — <i>Assembler Syntax</i>
01xx	PFWAIT.R — <i>Assembler Syntax</i>	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved

t	xx00	xx01	xx10	xx11
11xx	reserved	reserved	reserved	reserved

Table 211: RFET (from Table 7–292) Format RRR (no bits vary)

s	xx00	xx01	xx10	xx11
00xx	RFE _x — <i>Assembler Syntax</i>	RFUE _x — <i>Assembler Syntax</i>	RFDE _x — <i>Assembler Syntax</i>	reserved
01xx	RFWO _w — <i>Assembler Syntax</i>	RFWU _w — <i>Assembler Syntax</i>	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 212: RFM (from Table 7–292) Format RRR (nothing varies)

s	xx00	xx01	xx10	xx11
00xx	RFME — <i>Assembler Syntax</i>	CLREX — <i>Assembler Syntax</i>	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 213: ST1 (from Table 7–285) Format RRR (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	SSR — <i>Assembler Syntax</i> (t=0)	SSL — <i>Assembler Syntax</i> (t=0)	SSA8L — <i>Assembler Syntax</i> (t=0)	SSA8B — <i>Assembler Syntax</i> (t=0)
01xx	SSAI — <i>Assembler Syntax</i> (t=0)	reserved	RER — <i>Assembler Syntax</i>	WER — <i>Assembler Syntax</i>
10xx	ROTW _w — <i>Assembler Syntax</i> (s=0)	reserved	GETEX — <i>Assembler Syntax</i> (s=0)	reserved

r	xx00	xx01	xx10	xx11
11xx	reserved	reserved	NSAU _U — <i>Assembler Syntax</i>	NSAU _U — <i>Assembler Syntax</i>

Table 214: TLB (from Table 7–285) Format RRR (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	RITLB0 — <i>Assembler Syntax</i>
01xx	IITLB — <i>Assembler Syntax</i> (t=0)	PITLB — <i>Assembler Syntax</i>	WITLB — <i>Assembler Syntax</i>	RITLB1 — <i>Assembler Syntax</i>
10xx	reserved	reserved	reserved	RDTLB0 — <i>Assembler Syntax</i> or RPTLB0 — <i>Assembler Syntax</i>
11xx	IDTLB — <i>Assembler Syntax</i> (t=0)	PDTLB — <i>Assembler Syntax</i> or PPTLB — <i>Assembler Syntax</i>	WDTLB — <i>Assembler Syntax</i> or WPTLB — <i>Assembler Syntax</i>	RDTLB1 — <i>Assembler Syntax</i> or RPTLB1 — <i>Assembler Syntax</i>

Table 215: RT0 (from Table 7–285) Format RRR (t, r vary)

s	xx00	xx01	xx10	xx11
00xx	NEG — <i>Assembler Syntax</i>	ABS — <i>Assembler Syntax</i>	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 216: RST1 (from Table 7–284) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	SLLI — <i>Assembler Syntax</i>		SRAI — <i>Assembler Syntax</i>	
01xx	SRLI — <i>Assembler Syntax</i>	reserved	XSR — <i>Assembler Syntax</i>	ACCER — <i>Table 301</i>

op2	xx00	xx01	xx10	xx11
10xx	SRC — <i>Assembler Syntax</i>	SRL — <i>Assembler Syntax</i> (s=0)	SLL — <i>Assembler Syntax</i> (t=0)	SRA — <i>Assembler Syntax</i> (s=0)
11xx	MUL16U — <i>Assembler Syntax</i>	MUL16S — <i>Assembler Syntax</i>	reserved	IMP — <i>Table 301</i>

Table 217: ACCER (from Table 7–301) Format RRR (t, s vary)

op2	xx00	xx01	xx10	xx11
00xx	RER — <i>Assembler Syntax</i>			
01xx				
10xx	WER — <i>Assembler Syntax</i>			
11xx				

Table 218: IMP (from Table 7–301) Format RRR (t, s vary) ()

r	xx00	xx01	xx10	xx11
00xx	LICT — <i>Assembler Syntax</i>	SICT — <i>Assembler Syntax</i>	LICW — <i>Assembler Syntax</i>	SICW — <i>Assembler Syntax</i>
01xx	L32EX — <i>Assembler Syntax</i>	S32EX — <i>Assembler Syntax</i>	reserved	reserved
10xx	LDCT — <i>Assembler Syntax</i>	SDCT — <i>Assembler Syntax</i>	LDCW — <i>Assembler Syntax</i>	SDCW — <i>Assembler Syntax</i>
11xx	reserved	reserved	RFDX — <i>Table 303</i>	reserved

Table 219: RFDX (from Table 7–303) Format RRR (s varies)

t	xx00	xx01	xx10	xx11
00xx	RFDO — <i>Assembler Syntax</i> (s=0)	RFDD — <i>Assembler Syntax</i> (s=0,1)	reserved	reserved
01xx	reserved	reserved	reserved	reserved

t	xx00	xx01	xx10	xx11
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 220: RST2 (from Table 7–284) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	ANDB _P — <i>Assembler Syntax</i>	ANDBC _P — <i>Assembler Syntax</i>	ORB _P — <i>Assembler Syntax</i>	ORBC _P — <i>Assembler Syntax</i>
01xx	XORB _P — <i>Assembler Syntax</i>	reserved	SALT — <i>Assembler Syntax</i>	SALT — <i>Assembler Syntax</i>
10xx	MULL _I — <i>Assembler Syntax</i>	reserved	MULUH _I — <i>Assembler Syntax</i>	MULSH _I — <i>Assembler Syntax</i>
11xx	QUOU _I — <i>Assembler Syntax</i>	QUOS _I — <i>Assembler Syntax</i>	REMU _I — <i>Assembler Syntax</i>	REMS _I — <i>Assembler Syntax</i>

Table 221: RST3 (from Table 7–284) Formats RRR and RSR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	RSR — <i>Assembler Syntax</i>	WSR — <i>Assembler Syntax</i>	SEXT _U — <i>Assembler Syntax</i>	CLAMPS _U — <i>Assembler Syntax</i>
01xx	MIN _U — <i>Assembler Syntax</i>	MAX _U — <i>Assembler Syntax</i>	MINU _U — <i>Assembler Syntax</i>	MAXU _U — <i>Assembler Syntax</i>
10xx	MOVEQZ — <i>Assembler Syntax</i>	MOVNEZ — <i>Assembler Syntax</i>	MOVLTZ — <i>Assembler Syntax</i>	MOVGEZ — <i>Assembler Syntax</i>
11xx	MOVFP — <i>Assembler Syntax</i>	MOVTP — <i>Assembler Syntax</i>	RUR — <i>Assembler Syntax</i>	WUR — <i>Assembler Syntax</i>

Table 222: LSCX (from Table 7–284) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	LSXF — <i>Assembler Syntax</i>	LSXUF — <i>Assembler Syntax</i>	LDXF — <i>Assembler Syntax</i>	LDXUF — <i>Assembler Syntax</i>

op2	xx00	xx01	xx10	xx11
01xx	SSX _F — <i>Assembler Syntax</i>	SSXU _F — <i>Assembler Syntax</i>	SDX _F — <i>Assembler Syntax</i>	SDXU _F — <i>Assembler Syntax</i>
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 223: LSC4 (from Table 7–284) Format RRI4 (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	L32E — <i>Assembler Syntax</i>	BLKPRF — <i>Table 308</i>	DISPL — <i>Table 308</i>	reserved
01xx	S32E — <i>Assembler Syntax</i>	S32NB — <i>Assembler Syntax</i>	DISPS — <i>Table 308</i>	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 224: BLKPRF (from Table 7–308) Format RRR (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	reserved	DPFR.B — <i>Assembler Syntax</i>	DPFW.B — <i>Assembler Syntax</i>	DPFM.B — <i>Assembler Syntax</i>
01xx	reserved	DPFR.BF — <i>Assembler Syntax</i>	DPFW.B — <i>Assembler Syntax</i>	DPFM.BF — <i>Assembler Syntax</i>
10xx	reserved	DHI.B — <i>Assembler Syntax</i>	DHWB.B — <i>Assembler Syntax</i>	DHWBI.B — <i>Assembler Syntax</i>
11xx	reserved	reserved	reserved	reserved

Table 225: DISPL (from Table 7–308) Format RRR (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	L32DIS.* — <i>Assembler Syntax</i>	L32DIS.* — <i>Assembler Syntax</i>	L32DIS.* — <i>Assembler Syntax</i>	L32DIS.* — <i>Assembler Syntax</i>

r	xx00	xx01	xx10	xx11
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 226: DISPS (from Table 7–308) Format RRR (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	S32DIS.* — <i>Assembler Syntax</i>	S32DIS.* — <i>Assembler Syntax</i>	S32DIS.* — <i>Assembler Syntax</i>	S32DIS.* — <i>Assembler Syntax</i>
01xx	reserved	reserved	S32SI.* — <i>Assembler Syntax</i>	S32SI.* — <i>Assembler Syntax</i>
10xx	S32STK — <i>Assembler Syntax</i>	S32STK — <i>Assembler Syntax</i>	S32STK — <i>Assembler Syntax</i>	S32STK — <i>Assembler Syntax</i>
11xx	S32STK — <i>Assembler Syntax</i>	S32STK — <i>Assembler Syntax</i>	S32STK — <i>Assembler Syntax</i>	S32STK — <i>Assembler Syntax</i>

Table 227: FP0 (from Table 7–284) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	ADD.S _F — <i>Assembler Syntax</i>	SUB.S _F — <i>Assembler Syntax</i>	MUL.S _F — <i>Assembler Syntax</i>	reserved
01xx	MADD.S _F — <i>Assembler Syntax</i>	MSUB.S _F — <i>Assembler Syntax</i>	MADDN.S _F — <i>Assembler Syntax</i>	DIVN.S _F — <i>Assembler Syntax</i>
10xx	ROUND.S _F — <i>Assembler Syntax</i>	TRUNC.S _F — <i>Assembler Syntax</i>	FLOOR.S _F — <i>Assembler Syntax</i>	CEIL.S _F — <i>Assembler Syntax</i>
11xx	FLOAT.S _F — <i>Assembler Syntax</i>	UFLOAT.S _F — <i>Assembler Syntax</i>	UTRUNC.S _F — <i>Assembler Syntax</i>	FP1OP _F — <i>Table 312</i>

Table 228: FP1OP (from Table 7–312) Format RRR (s, r vary)

t	xx00	xx01	xx10	xx11
00xx	MOV.S _F — <i>Assembler Syntax</i>	ABS.S _F — <i>Assembler Syntax</i>	CVTD.S _F — <i>Assembler Syntax</i>	CONST.S _F — <i>Assembler Syntax</i>
01xx	RFR _F — <i>Assembler Syntax</i>	WFR _F — <i>Assembler Syntax</i>	NEG.S _F — <i>Assembler Syntax</i>	DIV0.S _F — <i>Assembler Syntax</i>
10xx	RECIP0.S _F — <i>Assembler Syntax</i>	SQRT0.S _F — <i>Assembler Syntax</i>	RSQRT0.S _F — <i>Assembler Syntax</i>	NEXP01.S _F — <i>Assembler Syntax</i>
11xx	MKSADJ.S _F — <i>Assembler Syntax</i>	MKDADJ.S _F — <i>Assembler Syntax</i>	ADDEXP.S _F — <i>Assembler Syntax</i>	ADDEXPM.S _F — <i>Assembler Syntax</i>

Table 229: FP1 (from Table 7–284) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	reserved	UN.S _F — <i>Assembler Syntax</i>	OEQ.S _F — <i>Assembler Syntax</i>	UEQ.S _F — <i>Assembler Syntax</i>
01xx	OLT.S _F — <i>Assembler Syntax</i>	ULT.S _F — <i>Assembler Syntax</i>	OLE.S _F — <i>Assembler Syntax</i>	ULE.S _F — <i>Assembler Syntax</i>
10xx	MOVEQZ.S _F — <i>Assembler Syntax</i>	MOVNEZ.S _F — <i>Assembler Syntax</i>	MOVLTZ.S _F — <i>Assembler Syntax</i>	MOVGEZ.S _F — <i>Assembler Syntax</i>
11xx	MOVF.S _F — <i>Assembler Syntax</i>	MOVT.S _F — <i>Assembler Syntax</i>	reserved	reserved

Table 230: DFP0 (from Table 7–284) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	ADD.D _F — <i>Assembler Syntax</i>	SUB.D _F — <i>Assembler Syntax</i>	MUL.D _F — <i>Assembler Syntax</i>	reserved
01xx	MADD.D _F — <i>Assembler Syntax</i>	MSUB.D _F — <i>Assembler Syntax</i>	MADDN.D _F — <i>Assembler Syntax</i>	DIVN.D _F — <i>Assembler Syntax</i>
10xx	ROUND.D _F — <i>Assembler Syntax</i>	TRUNC.D _F — <i>Assembler Syntax</i>	FLOOR.D _F — <i>Assembler Syntax</i>	CEIL.D _F — <i>Assembler Syntax</i>
11xx	FLOAT.D _F — <i>Assembler Syntax</i>	UFLOAT.D _F — <i>Assembler Syntax</i>	UTRUNC.D _F — <i>Assembler Syntax</i>	FP2OP _F — <i>Table 315</i>

Table 231: FP2OP (from Table 7–315) Format RRR (s, r vary)

t	xx00	xx01	xx10	xx11
00xx	MOV.D _F — <i>Assembler Syntax</i>	ABS.D _F — <i>Assembler Syntax</i>	CVTS.D _F — <i>Assembler Syntax</i>	CONST.D _F — <i>Assembler Syntax</i>
01xx	RFRD _F — <i>Assembler Syntax</i>	reserved	NEG.D _F — <i>Assembler Syntax</i>	DIV0.D _F — <i>Assembler Syntax</i>
10xx	RECIP0.D _F — <i>Assembler Syntax</i>	SQRT0.D _F — <i>Assembler Syntax</i>	RSQRT0.D _F — <i>Assembler Syntax</i>	NEXP01.D _F — <i>Assembler Syntax</i>
11xx	MKSADJ.D _F — <i>Assembler Syntax</i>	MKDADJ.D _F — <i>Assembler Syntax</i>	ADDEXP.D _F — <i>Assembler Syntax</i>	ADDEXPM.D _F — <i>Assembler Syntax</i>

Table 232: DFP1 (from Table 7–284) Format RRR (t, s, r vary)

op2	xx00	xx01	xx10	xx11
00xx	reserved	UN.D _F — <i>Assembler Syntax</i>	OEQ.D _F — <i>Assembler Syntax</i>	UEQ.D _F — <i>Assembler Syntax</i>
01xx	OLT.D _F — <i>Assembler Syntax</i>	ULT.D _F — <i>Assembler Syntax</i>	OLE.D _F — <i>Assembler Syntax</i>	ULE.D _F — <i>Assembler Syntax</i>
10xx	WFRD _F — <i>Assembler Syntax</i>	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 233: LSAI (from Table 7–283) Formats RRI8 and RRI4 (t, s, imm8 vary)

r	xx00	xx01	xx10	xx11
00xx	L8UI — <i>Assembler Syntax</i>	L16UI — <i>Assembler Syntax</i>	L32I — <i>Assembler Syntax</i>	reserved
01xx	S8I — <i>Assembler Syntax</i>	S16I — <i>Assembler Syntax</i>	S32I — <i>Assembler Syntax</i>	CACHE _C — <i>Table 318</i>
10xx	reserved	L16SI — <i>Assembler Syntax</i>	MOVI — <i>Assembler Syntax</i>	L32AI _Y — <i>Assembler Syntax</i>
11xx	ADDI — <i>Assembler Syntax</i>	ADDMI — <i>Assembler Syntax</i>	S32C1I _Y — <i>Assembler Syntax</i>	S32RI _Y — <i>Assembler Syntax</i>

Table 234: CACHE (from Table 7–318) Formats RRI8 and RRI4 (s, imm8 vary)

t	xx00	xx01	xx10	xx11
00xx	DPFR _C — <i>Assembler Syntax</i>	DPFW _C — <i>Assembler Syntax</i>	DPFRO _C — <i>Assembler Syntax</i>	DPFWO _C — <i>Assembler Syntax</i>
01xx	DHWB _C — <i>Assembler Syntax</i>	DHWBI _C — <i>Assembler Syntax</i>	DHI _C — <i>Assembler Syntax</i>	DII _C — <i>Assembler Syntax</i>
10xx	DCE _C — <i>Table 319</i>	reserved	reserved	reserved
11xx	IPF _C — <i>Assembler Syntax</i>	ICE _C — <i>Table 319</i>	IHI _C — <i>Assembler Syntax</i>	III _C — <i>Assembler Syntax</i>

Table 235: DCE (from Table 7–319) Format RRI4 (s, imm4 vary)

op1	xx00	xx01	xx10	xx11
00xx	DPFL _L — <i>Assembler Syntax</i>	DCI _C — <i>Assembler Syntax</i>	DHU _L — <i>Assembler Syntax</i>	DIU _L — <i>Assembler Syntax</i>
01xx	DIWB _C — <i>Assembler Syntax</i>	DIWBI _C — <i>Assembler Syntax</i>	DCWB _C — <i>Assembler Syntax</i>	DCWBI _C — <i>Assembler Syntax</i>
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 236: ICE (from Table 7–319) Format RRI4 (s, imm4 vary)

op1	xx00	xx01	xx10	xx11
00xx	IPFL _L — <i>Assembler Syntax</i>	reserved	IHU _L — <i>Assembler Syntax</i>	IIU _L — <i>Assembler Syntax</i>
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 237: LSCI (from Table 7–283) Format RRI8 (t, s, imm8 vary)

r	xx00	xx01	xx10	xx11
00xx	LSI _F — <i>Assembler Syntax</i>	LDI _F — <i>Assembler Syntax</i>	reserved	reserved
01xx	SSI _F — <i>Assembler Syntax</i>	SDI _F — <i>Assembler Syntax</i>	reserved	reserved
10xx	LSIU _F — <i>Assembler Syntax</i>	LDIU _F — <i>Assembler Syntax</i>	reserved	reserved
11xx	SSIU _F — <i>Assembler Syntax</i>	SDIU _F — <i>Assembler Syntax</i>	reserved	reserved

Table 238: MAC16 (from Table 7–283) Format RRR (t, s, r, op1 vary)

op2	xx00	xx01	xx10	xx11
00xx	MACID — <i>Table 323</i>	MACCD — <i>Table 323</i>	MACDD — <i>Table 323</i>	MACAD — <i>Table 323</i>
01xx	MACIA — <i>Table 323</i>	MACCA — <i>Table 323</i>	MACDA — <i>Table 323</i>	MACAA — <i>Table 323</i>
10xx	MACI — <i>Table 323</i>	MACC — <i>Table 323</i>	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 239: MACID (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	MULA.DD.LL.LDINC — <i>Assembler Syntax</i>	MULA.DD.HL.LDINC — <i>Assembler Syntax</i>	MULA.DD.LH.LDINC — <i>Assembler Syntax</i>	MULA.DD.HH.LDINC C — <i>Assembler Syntax</i>
11xx	reserved	reserved	reserved	reserved

Table 240: MACIA (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	MULA.DA.LL.LDINC — <i>Assembler Syntax</i>	MULA.DA.HL.LDINC — <i>Assembler Syntax</i>	MULA.DA.LH.LDINC — <i>Assembler Syntax</i>	MULA.DA.HH.LDINC — <i>Assembler Syntax</i>
11xx	reserved	reserved	reserved	reserved

Table 241: MACDD (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	MUL.DD.LL — <i>Assembler Syntax</i>	MUL.DD.HL — <i>Assembler Syntax</i>	MUL.DD.LH — <i>Assembler Syntax</i>	MUL.DD.HH — <i>Assembler Syntax</i>
10xx	MULA.DD.LL — <i>Assembler Syntax</i>	MULA.DD.HL — <i>Assembler Syntax</i>	MULA.DD.LH — <i>Assembler Syntax</i>	MULA.DD.HH — <i>Assembler Syntax</i>
11xx	MULS.DD.LL — <i>Assembler Syntax</i>	MULS.DD.HL — <i>Assembler Syntax</i>	MULS.DD.LH — <i>Assembler Syntax</i>	MULS.DD.HH — <i>Assembler Syntax</i>

Table 242: MACAD (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	MUL.AD.LL — <i>Assembler Syntax</i>	MUL.AD.HL — <i>Assembler Syntax</i>	MUL.AD.LH — <i>Assembler Syntax</i>	MUL.AD.HH — <i>Assembler Syntax</i>
10xx	MULA.AD.LL — <i>Assembler Syntax</i>	MULA.AD.HL — <i>Assembler Syntax</i>	MULA.AD.LH — <i>Assembler Syntax</i>	MULA.AD.HH — <i>Assembler Syntax</i>
11xx	MULS.AD.LL — <i>Assembler Syntax</i>	MULS.AD.HL — <i>Assembler Syntax</i>	MULS.AD.LH — <i>Assembler Syntax</i>	MULS.AD.HH — <i>Assembler Syntax</i>

Table 243: MACCD (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	MULA.DD.LL.LDDE C — <i>Assembler Syntax</i>	MULA.DD.HL.LDDE C — <i>Assembler Syntax</i>	MULA.DD.LH.LDDE C — <i>Assembler Syntax</i>	MULA.DD.HH.LDDE C — <i>Assembler Syntax</i>
11xx	reserved	reserved	reserved	reserved

Table 244: MACCA (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	MULA.DA.LL.LDDE C — <i>Assembler Syntax</i>	MULA.DA.HL.LDDE C — <i>Assembler Syntax</i>	MULA.DA.LH.LDDE C — <i>Assembler Syntax</i>	MULA.DA.HH.LDDE C — <i>Assembler Syntax</i>
11xx	reserved	reserved	reserved	reserved

Table 245: MACDA (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	reserved	reserved	reserved	reserved
01xx	MUL.DA.LL — <i>Assembler Syntax</i>	MUL.DA.HL — <i>Assembler Syntax</i>	MUL.DA.LH — <i>Assembler Syntax</i>	MUL.DA.HH — <i>Assembler Syntax</i>
10xx	MULA.DA.LL — <i>Assembler Syntax</i>	MULA.DA.HL — <i>Assembler Syntax</i>	MULA.DA.LH — <i>Assembler Syntax</i>	MULA.DA.HH — <i>Assembler Syntax</i>
11xx	MULS.DA.LL — <i>Assembler Syntax</i>	MULS.DA.HL — <i>Assembler Syntax</i>	MULS.DA.LH — <i>Assembler Syntax</i>	MULS.DA.HH — <i>Assembler Syntax</i>

Table 246: MACAA (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	UMUL.AA.LL — <i>Assembler Syntax</i>	UMUL.AA.HL — <i>Assembler Syntax</i>	UMUL.AA.LH — <i>Assembler Syntax</i>	UMUL.AA.HH — <i>Assembler Syntax</i>
01xx	MUL.AA.LL — <i>Assembler Syntax</i>	MUL.AA.HL — <i>Assembler Syntax</i>	MUL.AA.LH — <i>Assembler Syntax</i>	MUL.AA.HH — <i>Assembler Syntax</i>
10xx	MULA.AA.LL — <i>Assembler Syntax</i>	MULA.AA.HL — <i>Assembler Syntax</i>	MULA.AA.LH — <i>Assembler Syntax</i>	MULA.AA.HH — <i>Assembler Syntax</i>
11xx	MULS.AA.LL — <i>Assembler Syntax</i>	MULS.AA.HL — <i>Assembler Syntax</i>	MULS.AA.LH — <i>Assembler Syntax</i>	MULS.AA.HH — <i>Assembler Syntax</i>

Table 247: MACI (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	LDINC — <i>Assembler Syntax</i> (t=0)	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 248: MACC (from Table 7–323) Format RRR (t, s, r vary)

op1	xx00	xx01	xx10	xx11
00xx	LDDEC — <i>Assembler Syntax</i> (t=0)	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 249: CALLN (from Table 7–283) Format CALL (offset varies)

n	00	01	10	11
	CALL0 — <i>Assembler Syntax</i>	CALL4 — <i>Assembler Syntax</i>	CALL8 — <i>Assembler Syntax</i>	CALL12 — <i>Assembler Syntax</i>

Table 250: SI (from Table 7–283) Formats CALL, BRI8 and BRI12(offset varies)

n	00	01	10	11
	<i>J</i> — <i>Assembler Syntax</i>	<i>BZ</i> — <i>Table 335</i>	<i>B10</i> — <i>Table 335</i>	<i>B11</i> — <i>Table 335</i>

Table 251: BZ (from Table 7–335) Format BRI12 (s, imm12 vary)

m	00	01	10	11
	BEQZ — <i>Assembler Syntax</i>	BNEZ — <i>Assembler Syntax</i>	BLTZ — <i>Assembler Syntax</i>	BGEZ — <i>Assembler Syntax</i>

Table 252: B10 (from Table 7–335) Format BRI8 (s, r, imm8 vary)

m	00	01	10	11
	BEQI — <i>Assembler Syntax</i>	BNEI — <i>Assembler Syntax</i>	BLTI — <i>Assembler Syntax</i>	BGEI — <i>Assembler Syntax</i>

Table 253: B11 (from Table 7–335) Formats BRI8 and BRI12 (s, r, imm8 vary)

m	00	01	10	11
	ENTRYW — <i>Assembler Syntax</i>	<i>B1</i> — <i>Table 338</i>	BLTUI — <i>Assembler Syntax</i>	BGEUI — <i>Assembler Syntax</i>

Table 254: B1 (from Table 7–338) Format BRI8 (s, imm8 vary)

r	xx00	xx01	xx10	xx11
00xx	BFP — <i>Assembler Syntax</i>	BTP — <i>Assembler Syntax</i>	reserved	reserved
01xx	reserved	reserved	reserved	reserved

r	xx00	xx01	xx10	xx11
10xx	LOOP — <i>Assembler Syntax</i>	LOOPNEZ — <i>Assembler Syntax</i>	LOOPGTZ — <i>Assembler Syntax</i>	reserved
11xx	reserved	reserved	reserved	reserved

Table 255: B (from Table 7–283) Format RRI8 (t, s, imm8 vary)

r	xx00	xx01	xx10	xx11
00xx	BNONE — <i>Assembler Syntax</i>	BEQ — <i>Assembler Syntax</i>	BLT — <i>Assembler Syntax</i>	BLTU — <i>Assembler Syntax</i>
01xx	BALL — <i>Assembler Syntax</i>	BBC — <i>Assembler Syntax</i>	BBCI — <i>Assembler Syntax</i>	
10xx	BANY — <i>Assembler Syntax</i>	BNE — <i>Assembler Syntax</i>	BGE — <i>Assembler Syntax</i>	BGEU — <i>Assembler Syntax</i>
11xx	BNALL — <i>Assembler Syntax</i>	BBS — <i>Assembler Syntax</i>	BBSI — <i>Assembler Syntax</i>	

Table 256: ST2 (from Table 7–283) Formats RI7 and RI6 (s, r vary)

t	xx00	xx01	xx10	xx11
00xx	MOVI.N _N — <i>Assembler Syntax</i>			
01xx				
10xx	BEQZ.N _N — <i>Assembler Syntax</i>			
11xx	BNEZ.N _N — <i>Assembler Syntax</i>			

Table 257: ST3 (from Table 7–283) Format RRRN (t, s vary)

r	xx00	xx01	xx10	xx11
00xx	MOV.N _N — <i>Assembler Syntax</i>	reserved	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved

r	xx00	xx01	xx10	xx11
11xx	reserved	reserved	reserved	S3 — Table 342

Table 258: S3 (from Table 7–342) Format RRRN (s varies)

t	xx00	xx01	xx10	xx11
00xx	RET.N _N — Assembler Syntax (s=0)	RETW.N _{WN} — Assembler Syntax (s=0)	BREAK.N _N — Assembler Syntax (s=0)	NOP.N _N — Assembler Syntax (s=0)
01xx	reserved	reserved	ILH — Table 342	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

Table 259: ILH (from Table 7–342) Format RRRN (no fields vary)

s	xx00	xx01	xx10	xx11
00xx	ILL.N _N — Assembler Syntax	HALT.N _N — Assembler Syntax	reserved	reserved
01xx	reserved	reserved	reserved	reserved
10xx	reserved	reserved	reserved	reserved
11xx	reserved	reserved	reserved	reserved

9.3.2 CUST0 and CUST1 Opcode Encodings

CUST0 and CUST1 opcode encodings shown in [Table 199: QRST \(from Table 7–283\) Formats RRR, CALLX, and RSR \(t, s, r, op2 vary\)](#) on page 661 are not used by the core architecture, and are reserved for DSP coprocessors and designer-defined opcodes. In the future, customers who use these spaces for their own designer-defined opcodes will be able to add new Cadence-defined core architecture options without changing their opcodes or binary executables. Some versions of DSP coprocessors may use different portions of the CUST0 and CUST1 opcode space, which may overlap with designer-defined opcodes.

9.3.3 Cache-Option Opcode Encodings (Implementation-Specific)

The encodings for the r field sub-opcodes of the IMP family of opcodes, which are implementation-specific Cache-Option opcodes, are shown in [IMP \(from Table 7–301\)](#)

Format RRR (t, s vary) (Cache-Option Opcode Encodings (Implementation-Specific)). The `IMP` family of opcodes is reserved for these implementation-specific instructions. For a description of these instructions, see *Instruction Descriptions* on page 321.

10. Using the Xtensa Architecture

Topics:

- *The Windowed Register and CALL0 ABIs*
- *Floating Point Type Arguments and Return Values*
- *Boolean (Xtbool) Types Arguments and Return Values*
- *State Register Conventions*
- *Stack Frame with Wide Alignment*
- *Stack Initialization*
- *Other Conventions*
- *Assembly Code*

This chapter describes the Cadence software tool support of the Xtensa ISA and the conventions used by software.

10.1 The Windowed Register and CALL0 ABIs

The Xtensa ISA supports two different application binary interfaces (ABIs). The windowed register ABI works with the Windowed Register Option and is the default ABI. It comes in two variants, Fixed Window and Variable Window, mostly affecting specialized code that traverses or manipulates the call stack. The CALL0 ABI can be used with any Xtensa processor. It does not make use of register windows, so it typically has slightly worse performance and code size and better context-switch time than the windowed register ABI.

These two ABIs share much in common and diverge mostly in the areas of stack frame layout and general-purpose AR register usage. The basic data type sizes and alignments are identical, and the argument passing and return value conventions are nearly the same. Furthermore, the usage of TIE registers is controlled by `callee_saved` property in the TIE file and is independent of the choice of ABIs.

10.1.1 Windowed Register Usage and Stack Layout

The Windowed Register ABI supports:

- Variable Window supports call windows of 4, 8 or 12 registers. It is used in XEA2.

[Windowed AR Register Usage](#) shows the general-purpose register usage for the windowed register ABI. Registers a0 and a1 are reserved for the return address and stack pointer, respectively. They must always contain those values, because they are used for stack unwinding in debuggers and exception handling. Incoming arguments are stored in registers a2 through a7. The location of outgoing arguments depends on the window size.

Table 260: Windowed AR Register Usage

Register	Use
a0	Return address
a1 (sp)	Stack pointer
a2 – a7	Incoming arguments
a7	Callee's stack-frame pointer (optional)

The stack frame layout for the windowed register ABI is shown in [Stack Frame for the Windowed Register ABI \(Variable Window\)](#) (Variable Window) and [Stack Frame for the Windowed Register ABI \(Fixed Window\)](#) (Fixed Window). The stack grows down, from high to low addresses. The stack pointer (SP) must be aligned to 16-byte boundaries, unless the stack-frame contains wide-aligned data as described in [Stack Frame with Wide Alignment](#) on page 692. A stack-frame pointer (FP) may (but is not required to) be allocated in register a7. It is generally needed when the routine dynamically allocates space on the stack, such as

by calling `alloca`. If a frame pointer is used, its value is equal to the original stack pointer (immediately after entry to the function), before any `alloca` or other stack space allocation.

In the Variable Window variant, the register-spill overflow area is equal to $N-4$ words, where N can be 4, 8, or 12 as determined by the largest `CALLN` or `CALLXN` in the function. No such overflow area exists in the Fixed Window variant of the ABI. For details, see “Windowed Procedure-Call Protocol”.

Within any Windowed ABI compliant code, the stack pointer `SP` should only be modified by `ENTRY` and `MOVSP` instructions. If some other instruction modifies `SP`, any values in the register-spill area will not be moved. An exception to this rule is when setting the initial stack pointer for a new stack, before the first function call in a thread, where the register-spill area is guaranteed to be empty and where `MOVSP` cannot safely be used.

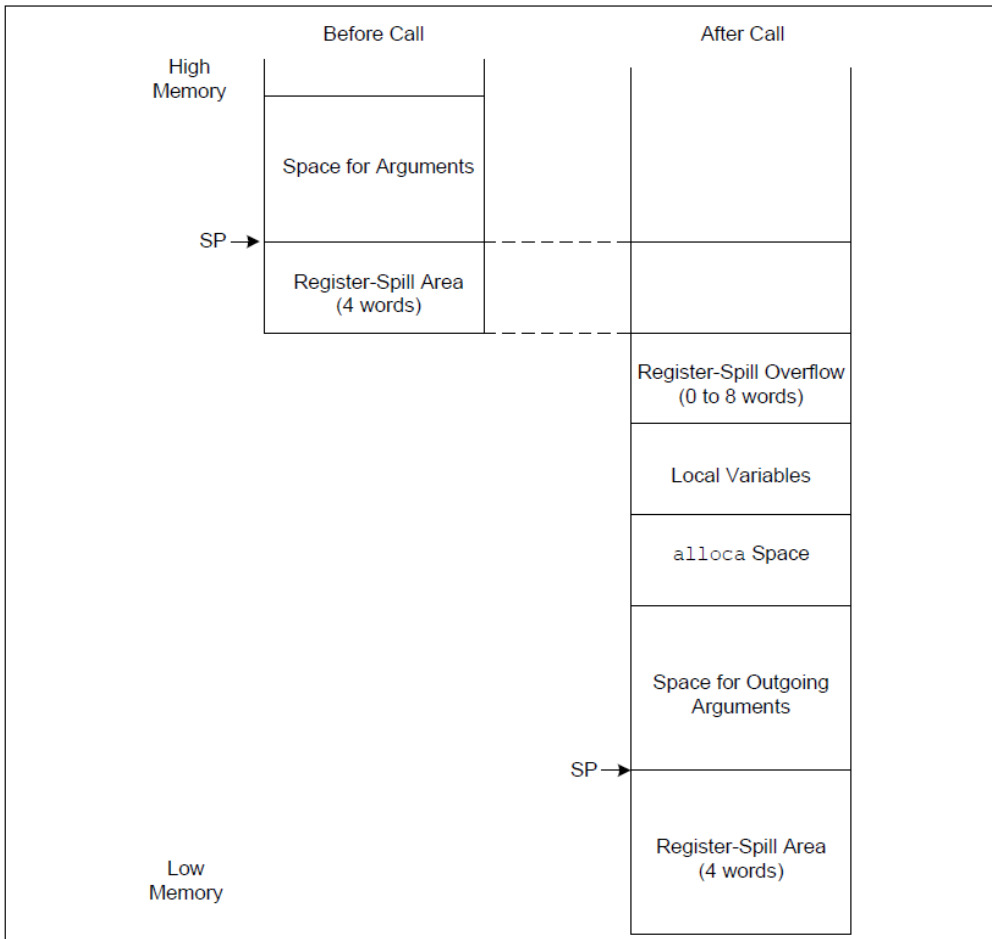


Figure 54: Stack Frame for the Windowed Register ABI (Variable Window)

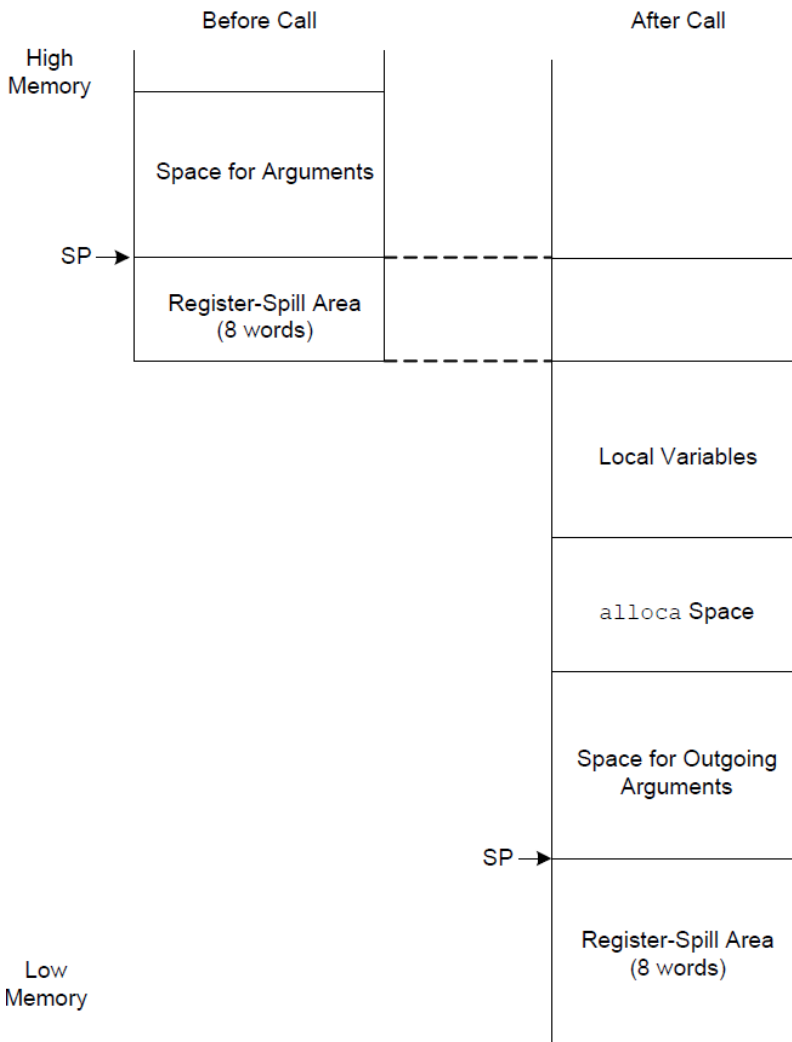


Figure 55: Stack Frame for the Windowed Register ABI (Fixed Window)

10.1.2 CALL0 AR Register Usage and Stack Layout

CALL0 AR Register Usage shows the general-purpose register usage for the CALL0 ABI. The stack pointer in register a1 and registers a12–a15 are callee-saved, but the rest of the registers are caller-saved. Register a0 holds the return address upon entry to a function, but unlike the windowed register ABI, it is not reserved for this purpose and may hold other values after the return address has been saved. Function arguments are passed in registers a2 through a7.

Table 261: CALL0 AR Register Usage

Register	Use
a0	Return Address
a1 (sp)	Stack Pointer (callee-saved)
a2 – a7	Function Arguments
a12 – a15	Callee-saved
a15	Stack-Frame Pointer (optional)

The stack frame layout for the CALL0 ABI is the same as for the windowed register ABI, except without the reserved register-spill areas. (Registers will need to be saved to the stack, but there is no convention for where in the frame to place that storage.) Like the windowed register ABI, the stack grows down and the stack pointer must be aligned to 16-byte boundaries. The optional stack-frame pointer is also used in the same way, but it is placed in register a15 with the CALL0 ABI.

10.1.3 Data Types and Alignment

Data Types and Alignment shows the data-type sizes and their alignment. The maximum alignment for TIE ctypes is 64 bytes.

Table 262: Data Types and Alignment

Data Type	Size and Alignment
char ¹	1 byte
short	2 bytes
int	4 bytes
long	4 bytes
long long	8 bytes
float	4 bytes
complex float	4 or 8 bytes ³
double	8 bytes

Data Type	Size and Alignment
complex double	8 or 16 bytes ⁴
long double	8 bytes
pointer	4 bytes
xtbool ²	1 byte
xtbool2 ²	1 byte
xtbool4 ²	1 byte
xtbool8 ²	1 byte
xtbool16 ²	2 bytes
TIE ctypes	user-defined

1. The `char` type is unsigned by default for Xtensa processors.
2. The `xtbool` types are only available if the Boolean registers are included in the processor configuration. See “Boolean Option” for information about the Boolean registers.
3. On configurations that support native complex, complex float is aligned to 8 bytes. On all other configurations it is aligned to 4 bytes. Note that an 8 byte alignment is in conflict with a strict reading of the ISO C standard.
4. On configurations that support native complex, complex double is aligned to 16 bytes. On all other configurations, it is aligned to 8 bytes. Note that a 16 byte alignment is in conflict with a strict reading of the ISO C standard.

10.1.4 Argument Passing in AR Registers

Arguments are passed in both registers and memory. In general, the first six words of arguments go in the AR register file, and any remaining arguments go on the stack. For a `CALLN` instruction (where N is 0 for the `CALL0` ABI, N is 8 for the Fixed Window variant of the windowed register ABI, and N is 4, 8, or 12 for the Variable Window variant of that ABI) the caller places the first arguments in registers `AR[N+2]` through `AR[N+7]`. (Note that this implies that `CALL12` can only be used when there are two words of arguments or less; only `AR[N+2]` and `AR[N+3]` can be used when $N=12$.) The callee receives these arguments in `AR[2]` through `AR[7]`.

If there are more than six words of arguments, the additional arguments are stored on the stack beginning at the caller’s stack pointer and at increasingly positive offsets from the stack

pointer. That is, the caller stores the seventh argument word (after the first six words in registers) at $[sp + 0]$, the eighth word at $[sp + 4]$, and so on. The callee can access these arguments in memory beginning at $[sp + \text{FRAMESIZE}]$, where *FRAMESIZE* is the size of the callee's stack frame.

All arguments consist of an integral number of 4-byte words. Thus, the minimum argument size is one word. Integer values smaller than a word (that is, `char` and `short`) are stored in the least significant portion of the argument word, with the upper bits set to zero for unsigned values or sign-extended for signed values.

When a value larger than 4 bytes is passed in registers, the ordering of the words is the same as the byte ordering. With little endian ordering, the least significant word goes in the first register. With big endian ordering, the most significant word comes first.

Each argument must be passed entirely in registers or entirely on the stack; an argument cannot be split with some words in registers and the remainder on the stack. If an argument does not fit entirely in the remaining unused registers, it is passed on the stack and those registers remain unused.

Arguments must be properly aligned. If the type of the argument requires 4-byte or less alignment, this requirement has no effect; all arguments have at least 4-byte alignment anyway. If an argument requires 8-byte alignment and is passed in registers, the first word must be in an even-numbered register. This sometimes requires leaving an odd-numbered register unused. Similarly, if an argument requires 16-byte alignment and is passed in registers, the first word must be in the first argument register ($AR[N+2]$); otherwise, it is passed on the stack. If an argument is passed in memory, the memory location must have the alignment required by the argument type.

Structures and other aggregate types are passed by value. The preceding rules apply to structures in the same way as scalars. If a structure is small enough to be passed in registers, the words of the structure are placed in registers according to their order in memory. A variable-sized structure is always passed on the stack and any remaining argument registers go unused. If the size of a structure is not an integral number of words, padding is inserted at one end of the structure. For structures smaller than a word, the padding is always in the most-significant part of the word. A structure larger than a word is padded in the last bytes of the last argument word, so that the structure is contiguous when the registers are stored to consecutive words of memory.

Values of TIE ctypes can also be passed as arguments. The TIE ctype register usage for parameter passing is described in [TIE Ctype Arguments](#).

10.1.5 Return Values in AR Registers

Values of four words or less are returned in registers. The callee places the return value in registers beginning with $AR[2]$ and continuing up to (and including) $AR[5]$, depending on the size of the value. For a `CALLN` instruction (where N is 0 for the `CALL0` ABI, N is 8 for the Fixed Window variant of the windowed register ABI, and N is 4, 8, or 12 for the Variable Window variant of that ABI) the caller receives these values in registers

AR[N+2] through AR[N+5]. (Note that, as with arguments, this limits the use of `CALL12` instructions. A `CALL12` instruction can only be used when the return value is two words or less; only AR[N+2] and AR[N+3] can be used when $N=12$.)

Return values smaller than a word are stored in the least-significant part of AR[2], with the upper bits set to zero for unsigned values or sign-extended for signed values.

Values larger than four words are returned by invisible reference. The caller passes a pointer as an invisible first argument and the callee stores the return value in the memory referenced by the pointer. The memory allocated by the caller must have the appropriate size and alignment for the return value.

Values of TIE ctypes are allowed as return values. See [Return Values of TIE Ctypes](#) for details.

10.1.6 Variable Arguments

Variable argument lists are handled in the same way as other arguments. There is no change to the calling convention for functions with variable argument lists. However, TIE ctype arguments cannot be used in the unnamed portion of a variable argument list.

10.2 Floating Point Type Arguments and Return Values

The traditional floating point ABI uses AR registers for passing and returning floating point values, regardless of whether the configuration supports hardware floating point. On configurations without hardware floating point, floating point values are always in AR registers. On configurations with floating point hardware, computation is performed on dedicated register files and floating point values are moved to and from the AR registers at call and return sites. A variable of type float occupies a single AR register. A variable of type double or long double occupies a pair of AR registers where the first register must be an even numbered register. With the traditional ABI, configurations with and without hardware floating point are compatible with each other, meaning that a calling function can be compiled for one type of configuration and the callee with the other.

The Hardware Floating Point ABI is optionally supported on configurations with hardware floating point. Float, double, and long double are passed in dedicated registers using the rules for TIE ctypes. Each variable is passed in a single register, starting with register 0. Register 0 can also be used for returning a variable. Structures or arrays of floating point variables are passed in AR register or memory just like integer variables. Variable arguments (varargs, such as printf) are passed in AR registers or memory.

The Hardware Floating Point ABI is not compatible with the traditional ABI. Both caller and callee must be compiled with the same ABI. Other than compatibility, there is no advantage in selecting the traditional ABI.

On configurations with single-precision hardware floating point support, but no double-precision support, variables of type float can use the Hardware Floating Point ABI, whereas double and long double variables will still be passed in AR register files.

The base hardware floating point configuration option uses 16 FR register file entries. All the registers are caller-saved. Some DSP processors, such as HiFi 3, HiFi 4 and Fusion use their custom DSP register files for floating point variables. Either floating point or custom fixed-point variables can be passed in the same register file. Each coprocessor decides whether any registers are callee-saved. The various DSP processors and the base floating point processors are not compatible with each other.

Complex float and complex double are treated as two individual float or double variables. With the Hardware Floating Point ABI, they are passed in floating point registers; otherwise they are passed in AR registers. Some DSP processors may have native support for complex and will hold both the real and imaginary components in the same register. See the individual DSP User Guide for details.

On configurations that support native complex, complex float variables are aligned to eight bytes and complex double variables are aligned to 16 bytes. Strictly speaking, this is not compatible with the ISO C99 and C11 language standards which state that complex float should have the same alignment as an array of two floats and complex double should have the same alignment as an array of two doubles. However, the extra alignment significantly improves performance on configurations with native complex support.

10.3 Boolean (*Xtbool*) Types Arguments and Return Values

Values of boolean types are allocated in BR registers with properly aligned indices as shown in *BR Register Usage*. Some BR registers such as b0 can be used for scalar xtbool type as well as packed boolean types such as xtbool2, xtbool4, etc. The width of the boolean values using b0 is implicitly implied by the instruction. For example, an ALL4 instruction with a xtbool4 result in b0 will actually produce results in individual BR registers b0, b1, b2, and b3.

Table 263: BR Register Usage

Boolean Types	Number of Bits	Allowed Register
xtbool	1	b0, b1, ..., b15
xtbool2	2	b0 ,b2, ..., b14
xtboo42	4	b0, b4, b8, b12
xtbool8	8	b0, b8
xtbool16	16	b0

The register used for passing a packed boolean types arguments must be the next one that supports the size of the argument type. For example, only b0 and b8 can be used to pass an `xtbool8` type argument. If the argument list used up to b3 for passing arguments and the next boolean argument is of `xtbool8` type, b4 to b7 will be skipped. Any BR registers skipped due to this requirement will stay unused for parameter passing of the current call. Otherwise, passing the boolean arguments follows the rule described in [TIE Ctype Arguments](#).

Boolean function values are returned in b0 and the size is determined by the type as described in [BR Register Usage](#).

10.4 State Register Conventions

In addition to the general-purpose AR and TIE register file, Xtensa processors may contain a variety of special states and TIE states (which may be mapped to user registers). Except for LITBASE, all non-privileged special registers are caller-saved. The compiler will use these locally so a caller must assume that their values can change when invoking any function. No other states are managed by the compiler. The compiler will not set them implicitly nor save them either in the caller or the callee. The programmer can decide how to use them and can set them in one function and use the set value in another. Note that saving by the compiler is separate from saving by the operating system. Operating systems are expected to save all thread-specific states on pre-emptive context-switches; see the Coprocessor and Custom State section of the HAL chapter in the *Xtensa System Software Reference Manual* for more details.

As a consequence of the `LOOP` special registers (`LBEG`, `LEND`, and `LCOUNT`) being caller-saved, the `LOOP` instructions should not be used for loops containing function calls. Doing so would require saving and restoring the `LOOP` registers around the call, which would overwhelm the advantage of the `LOOP` instructions.

10.5 Stack Frame with Wide Alignment

As described in [Windowed Register Usage and Stack Layout](#) on page 684, by default, the stack frame is 16-byte aligned. However, the maximal alignment allowed for a TIE ctype is 64-bytes. If a function has any wide-aligned (>16-byte aligned) data type for their arguments or the return values, the caller has to ensure that the SP is aligned to the largest alignment right before the call.

To facilitate allocating space for data with alignment greater than 16 bytes on stack, the stack frame may be dynamically aligned by the compiler to an alignment greater than 16 bytes by adjusting the stack pointer and frame pointer upon entry to a called function. The wide-aligned data on the stack may include local data, out-going arguments, and space for register spills. This alignment is transparent to the caller function, which may not have any wide-aligned data and requires only a 16-byte aligned stack frame. In the case when the dynamic alignment is performed, and incoming arguments are passed in memory, the called function

generally requires a frame pointer to access these incoming arguments as they are located at $[SP + FRAME_SIZE + PADDING]$. This alignment *PADDING* is assumed to be placed between the register spill areas in the case of the Windowed ABI (see [Dynamic Alignment for 64 byte-aligned Stack Frame with Windowed ABI](#)).

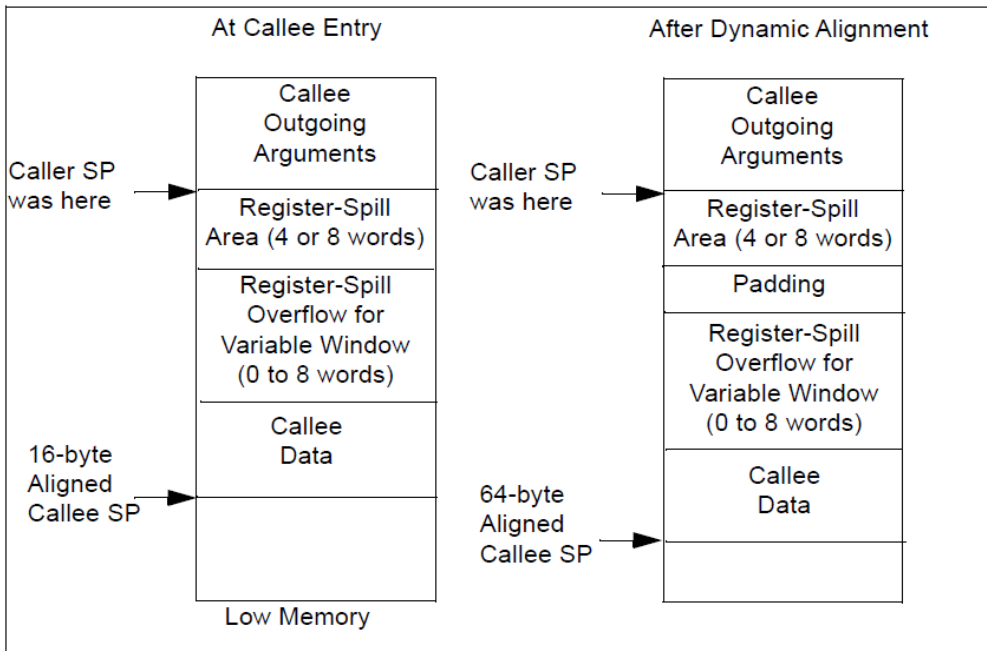


Figure 56: Dynamic Alignment for 64 byte-aligned Stack Frame with Windowed ABI

The padding size is computed at run-time and could be 0 if the original frame base is already wide-aligned. The compiler will generate code to access incoming arguments considering the *PADDING*. The stack layout is almost the same as what is described in [Windowed Register Usage and Stack Layout](#) on page 684 and [CALL0 AR Register Usage and Stack Layout](#) on page 686 after the dynamic alignment, except for the *assumed PADDING* space between Register-Spill Area and Register-Spill Overflow in [Dynamic Alignment for 64 byte-aligned Stack Frame with Windowed ABI](#). On returning from the callee function, the caller's SP is restored to the value before the call, as in the case when no dynamic alignment is performed.

Quite often, this dynamic alignment can be avoided at compile time if the caller's frame is already wide-aligned. If the callee knows that the incoming caller's stack frame has an alignment that is at least that of the callee's required wide-alignment, it can inherit the alignment by ensuring that the *FRAME_SIZE* is a multiple of the callee's wide-alignment, and skips the dynamic alignment at the function entry. Therefore, the Xtensa ABI requires that if any called functions have any wide-aligned data type for their arguments or the return values, the caller has to ensure that the SP is aligned to the largest alignment right before the call. An assembly function calling similar callee functions needs to make sure that the SP is properly aligned. Similarly, an assembly can assume that the SP is wide-aligned if it has

wide-aligned parameter or return value types and the caller function is compiled with proper declaration of the assembly function.

10.6 Stack Initialization

Creating and initializing a stack for a new thread requires:

- reserving some memory,
- setting up the initial stack frame,
- setting the stack pointer to the initial frame, and
- setting the initial return address (in register `a0`) to zero.

If the initial procedure executed by the thread does not store any data in the initial stack frame, and if all the call instructions in the initial procedure use the `CALL0` ABI or the Fixed Window ABI variant or a minimal window size of four for the Variable Window ABI variant, then the initial stack frame can be empty and requires no setup. The default C runtime initialization code meets these conditions, so that the stack can be initialized simply by setting the stack pointer to the high end of the reserved memory.

If the thread begins with some other code that may execute a `CALL8` or `CALL12` instruction in the Variable Window ABI variant or that requires storage on the stack, the initial frame must be constructed before jumping to the initial procedure. The size of the initial frame is equal to the sum of the local storage requirements and the extra save area. The stack pointer should be initialized to the high end of the reserved memory less the size of the initial frame.

Furthermore, assuming the thread begins executing with only the current register window loaded, the base save area at `sp - n` (where $n=16$ for Variable Window and $n=32$ for Fixed Window) must be initialized as if it had been written by a window overflow. Specifically, the stack pointer value stored at `(sp - (n-4))` must be set to the high end of the reserved stack area plus n bytes. This allows subsequent window overflows to locate the extra save area in the initial stack frame.

The return address register (`a0`) for the first procedure on the stack must be explicitly set to zero. This is used to mark the top of the stack for use by stack unwinding code.

The following code is an example of how the stack may be initialized to allow `CALL8` (but not `CALL12`) in the initial thread using the Variable Window ABI variant:

```
movi    a0, 0
movi    sp, stackbase + stacksize - 16
addi    a4, sp, 32      // point 16 past extra save area
s32e    a4, sp, -12     // access to extra save area
call8   firstfunction
```

The following code is an example of how the stack may be initialized to allow `CALL12` and “loc” bytes of locals and parameters in the initial thread (loc is a multiple of 16):

```
movi    a0, 0
movi    sp, stackbase + stacksize - loc - 32
addi    a4, sp, loc + 48 // point 16 past extra save area
s32e    a4, sp, -12     // access to extra save area
call12  firstfunction
```

10.7 Other Conventions

This section describes the usage conventions other than the Xtensa application binary interface (ABI).

10.7.1 Break Instruction Operands

The `break` (24-bit) instruction has two immediate 4-bit operands, and the `break.n` (narrow, 16-bit) instruction has one immediate 4-bit operand. These operands (informally called “break codes” in this section) can be used to convey relevant information to the debug exception handler. Their exact meaning is a matter of convention. However, some of the tools and software (debuggers, OS ports, and so forth) used with Xtensa cores necessarily make use of the break instructions, so some conventions had to be established. The conventions that have been adopted are described in this section.

Half of the break codes are reserved for use by software provided by Cadence and its partners, leaving the remaining half for “user-defined” purposes. Note that making use of user-defined break codes usually requires special OS or monitor support, or at least having control of the debug exception handler (or of the external OCD software when OCD mode is enabled). Break code allocations are described in [Breakpoint Instruction Operand Conventions](#).

Break codes have been allocated for a number of *planted breakpoints* (breakpoints that replace some arbitrary pre-existing instruction, usually under control of a debugger or related software, and usually temporarily) and *coded breakpoints* (breakpoints explicitly coded in the assembly source).

Planted breakpoints have a narrow (16-bit) and a wide (24-bit) version. Because 24-bit instructions exist in all Xtensa processors, instructions 24-bits or wider may be replaced with a 24-bit `BREAK` instruction. With the density option, the narrow version (`BREAK.N`) must generally be used when replacing an existing narrow instruction. Otherwise a wide break instruction would overwrite two sequential instructions, the second of which could be the (now corrupted) target of a branch. Note that without the density option, only the wide form of the break instruction can be used because the narrow version does not exist.

A number of coded breakpoints have been defined to provide a means of making various exceptions (that is, illegal instructions, load/store errors, and so forth) visible to the debugger,

which does not otherwise see these types of exceptions through the debug exception vector. These breakpoints necessarily require support from the OS (or RTOS). They are typically invoked by the OS for those exceptions and interrupts that neither the OS nor the application handles, thus providing an opportunity for a debugger (if one is active) to catch the condition. If the OS has its own mechanism for handling unregistered exceptions and interrupts, the relevant coded breakpoint is normally invoked before this mechanism (there often is no well-defined “after”). Thus, it is very important that the debug exception handler treat the coded breakpoint as a no-op if no debugger is active, to let the OS follow its default course of action. By convention, any `break 1,x` instruction must be skipped and ignored if no debugger is active. If the debug exception handler (or OCD software if OCD mode is enabled) detects the presence of a debugger, it will transfer control to the debugger. Otherwise, it must immediately resume execution at the instruction following the break (which requires incrementing EPC[DEBUGLEVEL] by two for `break.n` or by three for `break`), in effect making the break a no-op.

Another essential requirement for `break 1,0` through `break 1,5` is that the OS invoke these coded breakpoints in exactly the same context (core state) as when the exception was entered (except, necessarily, for PC and EXCSAVE*n*). This allows the debugger to know the exact state of the core at the time the exception (or interrupt) occurred, without requiring any OS dependency. For example, when detecting an unhandled level-1 user exception, the OS has typically saved (in EXCSAVE1 and possibly memory) and modified only a few address registers; these registers must all be restored prior to executing the `break 1,1` instruction. The debug exception handler can then examine all registers as they were when the user exception occurred, including examining EXCCAUSE to determine which exception occurred, and so forth. Similarly, following a `break 1,2` it can resolve which interrupt occurred using EPS[DEBUGLEVEL].INTLEVEL.

Coded breakpoints can always use the wide (24-bit) form of the break instruction, so they were not allocated from the limited number of narrow break instructions.

Table 264: Breakpoint Instruction Operand Conventions

Breakpoint Instruction	Type	Description
<code>break 0,0</code>	planted	<p>Breakpoints set by host debugger for debugging programs. These break instruction appear in code as a result of one of the following actions:</p> <ul style="list-style-type: none"> • The debugger can request the monitor to write the breakpoint instruction into the code. • The debugger can explicitly write this instruction into the code.

Breakpoint Instruction	Type	Description
<code>break 0,1</code>	planted	Breakpoints set by the monitor or OCD software for its own purposes. For example, xmon uses this breakpoint to detect and intercept UART interrupts. Ideally the presence of these breaks in the code is hidden from the debugger.
<code>break 0,2 to 0,15</code>	(undefined)	Reserved (Cadence)
<code>break 1,0</code>	coded	Signals an unhandled level 1 kernel exception
<code>break 1,1</code>	coded	Signals an unhandled level 1 user exception
<code>break 1,2</code>	coded	Signals an unhandled high-priority interrupt
<code>break 1,3</code>	coded	Signals an unhandled window overflow or underflow exception (unlikely to be invoked)
<code>break 1,4</code>	coded	Signals an unhandled double exception
<code>break 1,5</code>	coded	Signals an unhandled memory error exception
<code>break 1,6 to 1,13</code>	coded	Reserved (Cadence)
<code>break 1,14</code>	coded	Issue a request through the debugger. Any use of this break instruction is debugger-specific. For example, certain versions of GDB use this to implement target initiated host I/O.
<code>break 1,15</code>	coded	Transfer control to debugger if present. This is typically inserted manually in the code for debugging purposes, or to signal critical events that should cause entry into the debugger if one is active, but be ignored otherwise.
<code>break 2, x to 7, x</code>	(undefined)	Reserved (Cadence)

Breakpoint Instruction	Type	Description
<code>break 8, x to 15, x</code>	(undefined)	User-defined
<code>break.n 0</code>	planted	Same as <code>break 0, 0</code> , but can also replace narrow (16-bit) instructions.
<code>break.n 1</code>	planted	Same as <code>break 0, 1</code> , but can also replace narrow (16-bit) instructions.
<code>break.n 2 to 7</code>	(undefined)	Reserved (Cadence)
<code>break.n 8 to 15</code>	(undefined)	User-defined

10.7.2 System Calls

The details of system calls are inherently dependent on the operating system, but there are a few conventions that apply to all systems. The `SYSCALL` instruction has no immediate operands, so the system call parameters are passed in registers. Each operating system is free to define its own register usage for system call parameters, with the exception that in the Variable Window ABI, the system call request code must always be in register `a2`.

The system call request code 0 must be defined for all systems that use the Variable Window variant of the windowed register ABI. (If the Xtensa processor configuration uses the `CALL0` ABI, or the Fixed Window ABI variant, system call 0 need not be implemented.) The purpose of system call 0 is to flush the register windows to the stack. It is often useful to have a portable and reasonably efficient means of flushing register windows, such as when walking up the stack to find an exception handler. This system call provides an easy way to flush the register windows on all systems under the Variable Window ABI. The Fixed Window ABI uses the `SPILLW` instruction instead.

In general, each operating system can define its own conventions for which general-purpose registers may be modified by a system call, including which registers will hold any return values or error codes. For system call 0 in particular, no return value is expected and each operating system must guarantee that no general-purpose registers other than `a2` will be modified. The value in `a2` upon return from system call 0 depends on the operating system.

10.8 Assembly Code

This section describes various things of interest to the assembly language writer, including some examples.

10.8.1 Assembler Replacements and the Underscore Form

Machine code generated by the assembler may include opcode replacements for certain assembler opcodes. For example:

- The assembler can turn `ADD` into `ADD.N`, or `ADDI` into `ADDI.N`, and so forth when the density option is enabled.
- The assembler substitutes a different instruction when an operand is out of range. For example, it turns `MOVI` into `L32R` when the immediate is outside the range -2048 to 2047.
- By default, the assembler handles branches that won't reach. For example, writing:

```
beq a1, a2, label
```

might actually generate:

```
bne a1, a2, .L1
j    label
.L1:
```

if `label` is too far to reach with a simple `beq` instruction.

These transformations can be disabled by prefixing the instruction name with an underscore (for example, `_ADD`) and with pseudo-ops. The assembler directives `.begin` and `.end` with `no-transform` can also be used to enable and disable these transformations. See the *GNU Assembler User's Guide* for more detail.

10.8.2 Instruction Idioms

Instruction Idioms specifies the preferred instruction idioms for common operations. These idioms are specified using only core instructions; in some cases substituting density instructions would be appropriate.

Table 265: Instruction Idioms

Operation	Preferred Idiom
$AR[x] \leftarrow AR[y]$	<code>or ax, ay, ay</code> (generated by the <code>MOV</code> assembler macro) (or if present, use 16-bit option <code>MOV.N</code>)
$AR[x] \leftarrow \text{not } AR[y]$	<code>movi at, -1</code>

Operation	Preferred Idiom
	<code>xor ax, ay, at</code>
<code>AR[x] ← AR[y] and not AR[z]</code>	<code>and at, ay, az</code> <code>xor ax, ay, at</code>
<code>AR[x] ← imm32</code>	<code>l32r ax, literalpooloffset</code>
<code>AR[x] ← AR[y] << AR[z]</code>	<code>ssl az</code> <code>sll ax, ay</code>
<code>AR[x] ← AR[y] >>_u AR[z]</code>	<code>ssr az</code> <code>srl ax, ay</code>
<code>AR[x] ← AR[y] >>_s AR[z]</code>	<code>ssr az</code> <code>sra ax, ay</code>
<code>AR[x] ← rot(AR[y], AR[z])</code>	<code>ssa az</code> <code>src ax, ay, ay</code>
<code>AR[x] ← byteswap(AR[y])</code>	<code>ssai 8</code> <code>srli ax, ay, 16</code> <code>src ax, ax, ay</code> <code>src ax, ax, ax</code> <code>src ax, ay, ax</code>
<code>if AR[x] ≤ AR[y] goto L</code>	<code>bge ay, ax, L</code>
<code>if AR[x] > AR[y] goto L</code>	<code>blt ay, ax, L</code>
<code>if AR[x] ≤ imm goto L</code>	<code>blti ax, imm+1, L</code>
<code>if AR[x] > imm goto L</code>	<code>bgei ax, imm+1, L</code>
<code>AR[x] ← AR[y] ≠ AR[z]</code>	<code>movi at, 1</code> <code>xor ax, ay, az</code> <code>movnez ax, at, ax</code>
<code>AR[x] ← AR[y] = AR[z]</code>	<code>movi ax, 1</code>

Operation	Preferred Idiom
	<pre>bne ay, az, L movi ax, 0 L:</pre>
AR[x] ← AR[y] ≠ 0	<pre>movi at, 1 movi ax, 0 movnez ax, at, ay</pre>
AR[x] ← AR[y] = 0	<pre>movi at, 1 movi ax, 0 moveqz ax, at, ay</pre>
64-bit add (x ← y + z)	<pre>add ax0, ay0, az0 add ax1, ay1, az1 bgeu ax0, az0, L1 addi ax1, ax1, 1 L1:</pre>
64-bit subtract (x ← y - z)	<pre>sub ax0, ay0, az0 sub ax1, ay1, az1 bgeu ay0, az0, L addi ax1, ax1, -1 L:</pre>
64-bit compare and branch if x < y goto L	<pre>blt ax1, ay1, L bne ax1, ay1, L1 bltu ax0, ay0, L L1:</pre>
64-bit multiply (x ← y × z)	<pre>mull ax0, ay0, az0 muluh ax1, ay0, az0 mull t, ay0, az1 add ax1, ax1, t mull t, ay1, az0</pre>

Operation	Preferred Idiom
	<code>add ax1, ax1, t</code>
<code>BR[x] ← BR[y]</code>	<code>orb bx, by, by</code>
<code>BR[x] ← 0</code>	<code>xorb bx, b0, b0</code>
<code>BR[x] ← 1</code>	<code>orbc bx, b0, b0</code>