

Official
Release

AndeStar™

Instruction Set Architecture Manual

Document Number UM025-18

Date Issued 2013-10-14

Copyright© 2007-2013 Andes Technology Corporation.
All rights reserved.



Copyright Notice

Copyright© 2007-2013 Andes Technology Corporation. All rights reserved.

Words and logos marked with ™ are registered trademarks owned by Andes Technology Corporation. Other brands and names mentioned herein may be the trademarks of their respective owners.

This document contains confidential information of Andes Technology Corporation. Use of this copyright notice is precautionary and does not imply publication or disclosure. Neither the whole nor part of the information contained herein may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means without the written permission of Andes Technology Corporation

The product described herein is subject to continuous development and improvement; information herein is given by Andes in good faith but without warranties.

This document is intended only to assist the reader in the use of the product. Andes Technology Corporation shall not be liable for any loss or damage arising from the use of any information in this document, or any incorrect use of the product.

Contact Information

Should you have any problems with the information contained herein, please contact Andes Technology Corporation
by email support@andestech.com
or online website <https://es.andestech.com/eservice/>
for support giving:

- the document title
- the document number
- the page number(s) to which your comments apply
- a concise explanation of the problem

General suggestions for improvements are welcome.

Revision History

Rev.	Revision Date	Revised Chapter-Section	Revised Content
1.8	2013-10-14	Official Release 1. BGEZAL, BLTZAL 2. SMW 3. DSB 4. Section 9.2	(1) Fixed operation description errors for BGEZAL and BLTZAL. The error happens when (Rt == R30). (2) Added one cacheability type constraint for SMW instruction. (3) Added IRET (Reader) in the DSB requirement table. (4) Refine N9/N10's instruction latency table
1.7	2013-9-2	All	(1) Corrected typos and grammar errors (2) Fixed inconsistent formatting (3) Fixed heading formatting (4) Adjusted some writing styles for better readability (5) Removed section 1.4 and added section 1.2 architecture versions and extended instruction sets (6) Completed the incomplete descriptions (7) Used specific/precise reference (8) Refined N9/N10 Latency Table
1.5	2011-5-31	Section 8.8	Added Baseline V3 and V3m instructions.
1.4	2009-9-24	Section 8.3.1, 8.7.5, 8.7.6	Updated descriptions for ABS/DIVR/DIVSR instructions. (Sync to internal v6.6)
1.3	2009-5-5	Section 8.7.9, 8.7.19	Added LBUP/SBUP instructions.
1.2	2008-12-31	Section 4	Added Baseline V2 instructions.

Rev.	Revision Date	Revised Chapter-Section	Revised Content
1.1	2008-6-13	Section 1.4.1	(1) Added N10 latency information. (2) Added Reduced Register configuration description.
1.0 RB	2008-2-25	Section 8.2.1, 8.2.2	Added DIV/DIVS instructions.
1.0	2007-6-5	All	First version.

Table of Contents

COPYRIGHT NOTICE.....	I
CONTACT INFORMATION.....	I
REVISION HISTORY.....	II
LIST OF TABLES	XII
LIST OF FIGURES	XIV
1. INTRODUCTION	1
1.1. 32/16-BIT ISA	2
1.2. ARCHITECTURE VERSIONS AND EXTENDED INSTRUCTION SETS	2
1.3. DATA TYPES	3
1.4. REGISTERS	4
1.4.1. <i>Reduced Register Configuration Option.</i>	6
1.5. INSTRUCTION ENCODING.....	8
1.5.1. <i>32-bit Instruction Format.</i>	8
1.5.2. <i>16-bit Instruction Format.</i>	8
1.6. MISCELLANEOUS.....	9
2. 32-BIT BASELINE V1 INSTRUCTIONS.....	10
2.1. DATA-PROCESSING INSTRUCTIONS.....	11
2.2. LOAD AND STORE INSTRUCTIONS	14
2.3. JUMP AND BRANCH INSTRUCTIONS	18
2.4. PRIVILEGE RESOURCE ACCESS INSTRUCTIONS	19
2.5. MISCELLANEOUS INSTRUCTIONS	21
3. 16-BIT BASELINE V1 INSTRUCTIONS	24
3.1. 16-BIT TO 32-BIT INSTRUCTION MAPPING	24
4. 16/32-BIT BASELINE VERSION 2 INSTRUCTIONS.....	28
4.1. 16-BIT BASELINE V2 INSTRUCTIONS	29
4.2. 32-BIT BASELINE V2 INSTRUCTIONS.....	30
5. 16/32-BIT BASELINE VERSION 3 AND VERSION 3M INSTRUCTIONS.....	32
6. 32-BIT ISA EXTENSIONS	35
6.1. PERFORMANCE EXTENSION V1 INSTRUCTIONS	36
6.2. PERFORMANCE EXTENSION V2 INSTRUCTIONS	37
6.3. 32-BIT STRING EXTENSION	38

7. COPROCESSOR EXTENSION INSTRUCTIONS	39
8. DETAILED INSTRUCTION DESCRIPTION	40
8.1. 32-BIT BASELINE V1 INSTRUCTIONS	41
8.1.1. ADD (<i>Addition</i>).....	41
8.1.2. ADDI (<i>Add Immediate</i>).....	42
8.1.3. AND (<i>Bit-wise Logical And</i>).....	43
8.1.4. ANDI (<i>And Immediate</i>).....	44
8.1.5. BEQ (<i>Branch on Equal</i>).....	45
8.1.6. BEQZ (<i>Branch on Equal Zero</i>)	46
8.1.7. BGEZ (<i>Branch on Greater than or Equal to Zero</i>).....	47
8.1.8. BGEZAL (<i>Branch on Greater than or Equal to Zero and Link</i>).....	48
8.1.9. BGTZ (<i>Branch on Greater than Zero</i>).....	49
8.1.10. BLEZ (<i>Branch on Less than or Equal to Zero</i>).....	50
8.1.11. BLTZ (<i>Branch on Less than Zero</i>).....	51
8.1.12. BLTZAL (<i>Branch on Less than Zero and Link</i>).....	52
8.1.13. BNE (<i>Branch on Not Equal</i>).....	53
8.1.14. BNEZ (<i>Branch on Not Equal Zero</i>).....	54
8.1.15. BREAK (<i>Breakpoint</i>).....	55
8.1.16. CCTL (<i>Cache Control</i>).....	56
8.1.17. CMOVN (<i>Conditional Move on Not Zero</i>).....	67
8.1.18. CMOVZ (<i>Conditional Move on Zero</i>).....	68
8.1.19. DPREF/DPREFI (<i>Data Prefetch</i>).....	69
8.1.20. DSB (<i>Data Serialization Barrier</i>)	73
8.1.21. IRET (<i>Interruption Return</i>).....	76
8.1.22. ISB (<i>Instruction Serialization Barrier</i>).....	79
8.1.23. ISYNC (<i>Instruction Data Coherence Synchronization</i>).....	81
8.1.24. J (<i>Jump</i>).....	84
8.1.25. JAL (<i>Jump and Link</i>)	85
8.1.26. JR (<i>Jump Register</i>)	86
8.1.27. JR.xTOFF (<i>Jump Register and Translation OFF</i>).....	87
8.1.28. JRAL (<i>Jump Register and Link</i>)	89
8.1.29. JRAL.xTON (<i>Jump Register and Link and Translation ON</i>)	90
8.1.30. LB (<i>Load Byte</i>)	92
8.1.31. LBI (<i>Load Byte Immediate</i>)	94
8.1.32. LBS (<i>Load Byte Signed</i>).....	96
8.1.33. LBSI (<i>Load Byte Signed Immediate</i>).....	98
8.1.34. LH (<i>Load Halfword</i>)	100
8.1.35. LHI (<i>Load Halfword Immediate</i>)	102

8.1.36.	<i>LHS (Load Halfword Signed)</i>	104
8.1.37.	<i>LHSI (Load Halfword Signed Immediate)</i>	106
8.1.38.	<i>LLW (Load Locked Word)</i>	108
8.1.39.	<i>LMW (Load Multiple Word)</i>	111
8.1.40.	<i>LW (Load Word)</i>	116
8.1.41.	<i>LWI (Load Word Immediate)</i>	118
8.1.42.	<i>LWUP (Load Word with User Privilege Translation)</i>	120
8.1.43.	<i>MADD32 (Multiply and Add to Data Low)</i>	122
8.1.44.	<i>MADD64 (Multiply and Add Unsigned)</i>	123
8.1.45.	<i>MADDS64 (Multiply and Add Signed)</i>	124
8.1.46.	<i>MFSR (Move From System Register)</i>	125
8.1.47.	<i>MFUSR (Move From User Special Register)</i>	126
8.1.48.	<i>MOVI (Move Immediate)</i>	129
8.1.49.	<i>MSUB32 (Multiply and Subtract to Data Low)</i>	130
8.1.50.	<i>MSUB64 (Multiply and Subtract Unsigned)</i>	131
8.1.51.	<i>MSUBS64 (Multiply and Subtract Signed)</i>	132
8.1.52.	<i>MSYNC (Memory Data Coherence Synchronization)</i>	133
8.1.53.	<i>MTSR (Move To System Register)</i>	136
8.1.54.	<i>MTUSR (Move To User Special Register)</i>	137
8.1.55.	<i>MUL (Multiply Word to Register)</i>	140
8.1.56.	<i>MULT32 (Multiply Word to Data Low)</i>	141
8.1.57.	<i>MULT64 (Multiply Word Unsigned)</i>	142
8.1.58.	<i>MULTS64 (Multiply Word Signed)</i>	143
8.1.59.	<i>NOP (No Operation)</i>	144
8.1.60.	<i>NOR (Bit-wise Logical Nor)</i>	145
8.1.61.	<i>OR (Bit-wise Logical Or)</i>	146
8.1.62.	<i>ORI (Or Immediate)</i>	147
8.1.63.	<i>RET (Return from Register)</i>	148
8.1.64.	<i>RET.xTOFF (Return from Register and Translation OFF)</i>	149
8.1.65.	<i>ROTR (Rotate Right)</i>	151
8.1.66.	<i>ROTRI (Rotate Right Immediate)</i>	152
8.1.67.	<i>SB (Store Byte)</i>	153
8.1.68.	<i>SBI (Store Byte Immediate)</i>	155
8.1.69.	<i>SCW (Store Conditional Word)</i>	157
8.1.70.	<i>SEB (Sign Extend Byte)</i>	161
8.1.71.	<i>SEH (Sign Extend Halfword)</i>	162
8.1.72.	<i>SETEND (Set data endian)</i>	163
8.1.73.	<i>SETGIE (Set global interrupt enable)</i>	164

8.1.74.	<i>SETHI (Set High Immediate)</i>	165
8.1.75.	<i>SH (Store Halfword)</i>	166
8.1.76.	<i>SHI (Store Halfword Immediate)</i>	168
8.1.77.	<i>SLL (Shift Left Logical)</i>	170
8.1.78.	<i>SLLI (Shift Left Logical Immediate)</i>	171
8.1.79.	<i>SLT (Set on Less Than)</i>	172
8.1.80.	<i>SLTI (Set on Less Than Immediate)</i>	173
8.1.81.	<i>SLTS (Set on Less Than Signed)</i>	174
8.1.82.	<i>SLTSI (Set on Less Than Signed Immediate)</i>	175
8.1.83.	<i>SMW (Store Multiple Word)</i>	176
8.1.84.	<i>SRA (Shift Right Arithmetic)</i>	181
8.1.85.	<i>SRAI (Shift Right Arithmetic Immediate)</i>	182
8.1.86.	<i>SRL (Shift Right Logical)</i>	183
8.1.87.	<i>SRLI (Shift Right Logical Immediate)</i>	184
8.1.88.	<i>STANDBY (Wait for External Event)</i>	185
8.1.89.	<i>SUB (Subtraction)</i>	188
8.1.90.	<i>SUBRI (Subtract Reverse Immediate)</i>	189
8.1.91.	<i>SVA (Set on Overflow Add)</i>	190
8.1.92.	<i>SVS (Set on Overflow Subtract)</i>	191
8.1.93.	<i>SW (Store Word)</i>	192
8.1.94.	<i>SWI (Store Word Immediate)</i>	194
8.1.95.	<i>SWUP (Store Word with User Privilege Translation)</i>	196
8.1.96.	<i>SYSCALL (System Call)</i>	198
8.1.97.	<i>TEQZ (Trap if equal 0)</i>	199
8.1.98.	<i>TNEZ (Trap if not equal 0)</i>	200
8.1.99.	<i>TLBOP (TLB Operation)</i>	201
8.1.100.	<i>TRAP (Trap exception)</i>	209
8.1.101.	<i>WSBH (Word Swap Byte within Halfword)</i>	210
8.1.102.	<i>XOR (Bit-wise Logical Exclusive Or)</i>	211
8.1.103.	<i>XORI (Exclusive Or Immediate)</i>	212
8.1.104.	<i>ZEB (Zero Extend Byte)</i>	213
8.1.105.	<i>ZEH (Zero Extend Halfword)</i>	214
8.2.	32-BIT BASELINE V1 OPTIONAL INSTRUCTIONS	215
8.2.1.	<i>DIV (Unsigned Integer Divide)</i>	215
8.2.2.	<i>DIVS (Signed Integer Divide)</i>	217
8.3.	32-BIT PERFORMANCE EXTENSION INSTRUCTIONS	219
8.3.1.	<i>ABS (Absolute)</i>	219
8.3.2.	<i>AVE (Average with Rounding)</i>	220

8.3.3.	<i>BCLR (Bit Clear)</i>	221
8.3.4.	<i>BSET (Bit Set)</i>	222
8.3.5.	<i>BTGL (Bit Toggle)</i>	223
8.3.6.	<i>BTST (Bit Test)</i>	224
8.3.7.	<i>CLIP (Clip Value)</i>	225
8.3.8.	<i>CLIPS (Clip Value Signed)</i>	226
8.3.9.	<i>CLO (Count Leading Ones)</i>	227
8.3.10.	<i>CLZ (Count Leading Zeros)</i>	228
8.3.11.	<i>MAX (Maximum)</i>	229
8.3.12.	<i>MIN (Minimum)</i>	230
8.4.	32-BIT PERFORMANCE EXTENSION VERSION 2 INSTRUCTIONS.....	231
8.4.1.	<i>BSE (Bit Stream Extraction)</i>	231
8.4.2.	<i>BSP (Bit Stream Packing)</i>	237
8.4.3.	<i>PBSAD (Parallel Byte Sum of Absolute Difference)</i>	242
8.4.4.	<i>PBSADA (Parallel Byte Sum of Absolute Difference Accum)</i>	243
8.5.	32-BIT STRING EXTENSION INSTRUCTIONS	244
8.5.1.	<i>FFB (Find First Byte)</i>	244
8.5.2.	<i>FFBI (Find First Byte Immediate)</i>	246
8.5.3.	<i>FFMISM (Find First Mis-match)</i>	248
8.5.4.	<i>FLMISM (Find Last Mis-match)</i>	250
8.6.	16-BIT BASELINE V1 INSTRUCTIONS.....	252
8.6.1.	<i>ADD (Add Register)</i>	253
8.6.2.	<i>ADDI (Add Immediate)</i>	255
8.6.3.	<i>BEQS38 (Branch on Equal Implied R5)</i>	257
8.6.4.	<i>BEQZ38 (Branch on Equal Zero)</i>	258
8.6.5.	<i>BEQZS8 (Branch on Equal Zero Implied R15)</i>	259
8.6.6.	<i>BNES38 (Branch on Not Equal Implied R5)</i>	260
8.6.7.	<i>BNEZ38 (Branch on Not Equal Zero)</i>	261
8.6.8.	<i>BNEZS8 (Branch on Not Equal Zero Implied R15)</i>	262
8.6.9.	<i>BREAK16 (Breakpoint)</i>	263
8.6.10.	<i>J8 (Jump Immediate)</i>	264
8.6.11.	<i>JR5 (Jump Register)</i>	265
8.6.12.	<i>JRAL5 (Jump Register and Link)</i>	266
8.6.13.	<i>LBI333 (Load Byte Immediate Unsigned)</i>	267
8.6.14.	<i>LHI333 (Load Halfword Immediate Unsigned)</i>	268
8.6.15.	<i>LWI333 (Load Word Immediate)</i>	270
8.6.16.	<i>LWI37 (Load Word Immediate with Implied FP)</i>	272
8.6.17.	<i>LWI450 (Load Word Immediate)</i>	274

8.6.18.	<i>MOV55 (Move Register)</i>	276
8.6.19.	<i>MOVI55 (Move Immediate)</i>	277
8.6.20.	<i>NOP16 (No Operation)</i>	278
8.6.21.	<i>RET5 (Return from Register)</i>	279
8.6.22.	<i>SBI333 (Store Byte Immediate)</i>	280
8.6.23.	<i>SEB33 (Sign Extend Byte)</i>	281
8.6.24.	<i>SEH33 (Sign Extend Halfword)</i>	282
8.6.25.	<i>SHI333 (Store Halfword Immediate)</i>	283
8.6.26.	<i>SLLI333 (Shift Left Logical Immediate)</i>	285
8.6.27.	<i>SLT45 (Set on Less Than Unsigned)</i>	286
8.6.28.	<i>SLTI45 (Set on Less Than Unsigned Immediate)</i>	287
8.6.29.	<i>SLTS45 (Set on Less Than Signed)</i>	288
8.6.30.	<i>SLTSI45 (Set on Less Than Signed Immediate)</i>	289
8.6.31.	<i>SRAI45 (Shift Right Arithmetic Immediate)</i>	290
8.6.32.	<i>SRLI45 (Shift Right Logical Immediate)</i>	291
8.6.33.	<i>SUB (Subtract Register)</i>	292
8.6.34.	<i>SUBI (Subtract Immediate)</i>	294
8.6.35.	<i>SWI333 (Store Word Immediate)</i>	296
8.6.36.	<i>SWI37 (Store Word Immediate with Implied FP)</i>	298
8.6.37.	<i>SWI450 (Store Word Immediate)</i>	300
8.6.38.	<i>X11B33 (Extract the Least 11 Bits)</i>	302
8.6.39.	<i>XLSB33 (Extract LSB)</i>	303
8.6.40.	<i>ZEB33 (Zero Extend Byte)</i>	304
8.6.41.	<i>ZEH33 (Zero Extend Halfword)</i>	305
8.7.	16-BIT AND 32-BIT BASELINE VERSION 2 INSTRUCTIONS	306
8.7.1.	<i>ADDI10S (Add Immediate with Implied Stack Pointer)</i>	306
8.7.2.	<i>LWI37SP (Load Word Immediate with Implied SP)</i>	307
8.7.3.	<i>SWI37SP (Store Word Immediate with Implied SP)</i>	309
8.7.4.	<i>ADDI.gp (GP-implied Add Immediate)</i>	311
8.7.5.	<i>DIVR (Unsigned Integer Divide to Registers)</i>	312
8.7.6.	<i>DIVSR (Signed Integer Divide to Registers)</i>	314
8.7.7.	<i>LBI.gp (GP-implied Load Byte Immediate)</i>	316
8.7.8.	<i>LBSI.gp (GP-implied Load Byte Signed Immediate)</i>	318
8.7.9.	<i>LBUP (Load Byte with User Privilege Translation)</i>	319
8.7.10.	<i>LHI.gp (GP-implied Load Halfword Immediate)</i>	320
8.7.11.	<i>LHSI.gp (GP-implied Load Signed Halfword Immediate)</i>	321
8.7.12.	<i>LMWA (Load Multiple Word with Alignment Check)</i>	322
8.7.13.	<i>LWI.gp (GP-implied Load Word Immediate)</i>	327

8.7.14.	<i>MADDR32 (Multiply and Add to 32-Bit Register)</i>	328
8.7.15.	<i>MSUBR32 (Multiply and Subtract from 32-Bit Register)</i>	329
8.7.16.	<i>MULR64 (Multiply Word Unsigned to Registers)</i>	330
8.7.17.	<i>MULSR64 (Multiply Word Signed to Registers)</i>	332
8.7.18.	<i>SBI.gp (GP-implied Store Byte Immediate)</i>	334
8.7.19.	<i>SBUP (Store Byte with User Privilege Translation)</i>	335
8.7.20.	<i>SHI.gp (GP-implied Store Halfword Immediate)</i>	336
8.7.21.	<i>SMWA (Store Multiple Word with Alignment Check)</i>	337
8.7.22.	<i>SWI.gp (GP-implied Store Word Immediate)</i>	342
8.8.	16-BIT AND 32-BIT BASELINE VERSION 3 INSTRUCTIONS	343
8.8.1.	<i>ADD_SLLI (Addition with Shift Left Logical Immediate)</i>	345
8.8.2.	<i>ADD_SRLI (Addition with Shift Right Logical Immediate)</i>	346
8.8.3.	<i>ADDR36.SP (Add Immediate to Register with Implied Stack Pointer)</i>	347
8.8.4.	<i>ADD5.PC (Add with Implied PC)</i>	348
8.8.5.	<i>AND_SLLI (Logical And with Shift Left Logical Immediate)</i>	349
8.8.6.	<i>AND_SRLI (Logical And with Shift Right Logical Immediate)</i>	350
8.8.7.	<i>AND33 (Bit-wise Logical And)</i>	351
8.8.8.	<i>BEQC (Branch on Equal Constant)</i>	352
8.8.9.	<i>BNEC (Branch on Not Equal Constant)</i>	353
8.8.10.	<i>BITC (Bit Clear)</i>	354
8.8.11.	<i>BITCI (Bit Clear Immediate)</i>	355
8.8.12.	<i>BMSKI33 (Bit Mask Immediate)</i>	356
8.8.13.	<i>CCTL L1D_WBALL (L1D Cache Writeback All)</i>	357
8.8.14.	<i>FEXTI33 (Field Extraction Immediate)</i>	358
8.8.15.	<i>JRALNEZ (Jump Register and Link on Not Equal Zero)</i>	359
8.8.16.	<i>JRNEZ (Jump Register on Not Equal Zero)</i>	360
8.8.17.	<i>LWI45.FE (Load Word Immediate with Implied Function Entry Point)</i>	361
8.8.18.	<i>MOVD44 (Move Double Registers)</i>	363
8.8.19.	<i>MOVPI45 (Move Positive Immediate)</i>	365
8.8.20.	<i>MUL33 (Multiply Word to Register)</i>	366
8.8.21.	<i>NEG33 (Negative)</i>	367
8.8.22.	<i>NOT33 (Not)</i>	368
8.8.23.	<i>OR_SLLI (Logical Or with Shift Left Logical Immediate)</i>	369
8.8.24.	<i>OR_SRLI (Logical Or with Shift Right Logical Immediate)</i>	370
8.8.25.	<i>OR33 (Bit-wise Logical Or)</i>	371
8.8.26.	<i>POP25 (Pop Multiple Words from Stack and Return)</i>	372
8.8.27.	<i>PUSH25 (Push Multiple Words to Stack and Set FE)</i>	375
8.8.28.	<i>SUB_SLLI (Subtraction with Shift Left Logical Immediate)</i>	378

8.8.29.	<i>SUB_SRLI (Subtraction with Shift Right Logical Immediate)</i>	379
8.8.30.	<i>XOR_SLLI (Logical Exclusive Or with Shift Left Logical Immediate)</i>	380
8.8.31.	<i>XOR_SRLI (Logical Exclusive Or with Shift Right Logical Immediate)</i>	381
8.8.32.	<i>XOR33 (Bit-wise Logical Exclusive Or)</i>	382
9.	INSTRUCTION LATENCY FOR ANDESCORE IMPLEMENTATIONS	383
9.1.	N12 FAMILY IMPLEMENTATION.....	384
9.1.1.	<i>Instruction Latency due to Resource Dependency</i>	384
9.1.2.	<i>Cycle Penalty due to N12 Pipeline Control Mishaps Recovery</i>	388
9.2.	N9/N10 FAMILY IMPLEMENTATION	389
9.2.1.	<i>Dependency-related Instruction Latency</i>	389
9.2.2.	<i>Self-stall-related Instruction Latency</i>	393
9.2.3.	<i>Cycle Penalty due to N9/N10 Pipeline Control Mishap Recover</i>	395
9.2.4.	<i>Cycle Penalty due to Resource Contention</i>	396
9.3.	N8 FAMILY IMPLEMENTATION	397
9.3.1.	<i>Dependency-related Instruction Latency</i>	397
9.3.2.	<i>Execution Latency</i>	400
9.3.3.	<i>Data Hazard Penalty</i>	402
9.3.4.	<i>Miscellaneous Penalty</i>	403
10.	ANDESCORE N12 IMPLEMENTATION ISA FEATURE LIST	404
10.1.	CCTL INSTRUCTION.....	405
10.2.	STANDBY INSTRUCTION	405
11.	ANDESCORE N1213 HARDCORE N1213_43U1HA0 IMPLEMENTATION RESTRICTION	406
11.1.	INSTRUCTION RESTRICTION	407
11.2.	ISYNC INSTRUCTION NOTE	407

List of Tables

TABLE 1. ANDESTAR GENERAL PURPOSE REGISTERS	4
TABLE 2. ANDESTAR USER SPECIAL REGISTERS	6
TABLE 3. ANDESTAR STATUS REGISTERS	6
TABLE 4. REGISTERS FOR REDUCED REGISTER CONFIGURATION	7
TABLE 5. ALU INSTRUCTION WITH IMMEDIATE (BASELINE V1)	11
TABLE 6. ALU INSTRUCTION (BASELINE V1)	11
TABLE 7. SHIFTER INSTRUCTION (BASELINE V1)	12
TABLE 8. MULTIPLY INSTRUCTION (BASELINE V1)	12
TABLE 9. DIVIDE INSTRUCTIONS (BASELINE V1)	13
TABLE 10. LOAD / STORE ADDRESSING MODE	14
TABLE 11. LOAD / STORE INSTRUCTION (BASELINE)	14
TABLE 12. LOAD / STORE INSTRUCTION (BASELINE)	15
TABLE 13. LOAD / STORE INSTRUCTION (BASELINE)	15
TABLE 14. LOAD / STORE INSTRUCTION (BASELINE)	16
TABLE 15. LOAD / STORE MULTIPLE WORD INSTRUCTION (BASELINE)	17
TABLE 16. LOAD / STORE INSTRUCTION FOR ATOMIC UPDATES (BASELINE)	17
TABLE 17. LOAD / STORE INSTRUCTIONS WITH USER-MODE PRIVILEGE	17
TABLE 18. JUMP INSTRUCTION (BASELINE)	18
TABLE 19. BRANCH INSTRUCTION (BASELINE)	18
TABLE 20. BRANCH WITH LINK INSTRUCTION (BASELINE)	19
TABLE 21. READ/WRITE SYSTEM REGISTERS (BASELINE)	19
TABLE 22. JUMP REGISTER WITH SYSTEM REGISTER UPDATE (BASELINE)	19
TABLE 23. MMU INSTRUCTION (BASELINE)	20
TABLE 24. CONDITIONAL MOVE (BASELINE)	21
TABLE 25. SYNCHRONIZATION INSTRUCTION (BASELINE)	21
TABLE 26. PREFETCH INSTRUCTION (BASELINE)	21
TABLE 27. NOP INSTRUCTION (BASELINE)	21
TABLE 28. SERIALIZATION INSTRUCTION (BASELINE)	21
TABLE 29. EXCEPTION GENERATION INSTRUCTION (BASELINE)	22
TABLE 30. SPECIAL RETURN INSTRUCTION (BASELINE)	22
TABLE 31. CACHE CONTROL INSTRUCTION (BASELINE)	22
TABLE 32. MISCELLANEOUS INSTRUCTIONS (BASELINE)	22
TABLE 33. MOVE INSTRUCTION (16-BIT)	24
TABLE 34. ADD/SUB INSTRUCTION WITH IMMEDIATE (16-BIT)	24
TABLE 35. ADD/SUB INSTRUCTION (16-BIT)	24
TABLE 36. SHIFT INSTRUCTION WITH IMMEDIATE (16-BIT)	25
TABLE 37. BIT FIELD MASK INSTRUCTION WITH IMMEDIATE (16-BIT)	25

TABLE 38. LOAD / STORE INSTRUCTION (16-BIT)	25
TABLE 39. LOAD/STORE INSTRUCTION WITH IMPLIED FP (16-BIT)	26
TABLE 40. BRANCH AND JUMP INSTRUCTION (16-BIT)	26
TABLE 41. COMPARE AND BRANCH INSTRUCTION (16-BIT)	27
TABLE 42. MISC. INSTRUCTION (16-BIT)	27
TABLE 43. ALU INSTRUCTIONS	29
TABLE 44. LOAD/STORE INSTRUCTION	29
TABLE 45. ALU INSTRUCTIONS	30
TABLE 46. MULTIPLY AND DIVIDE INSTRUCTIONS	30
TABLE 47. LOAD/STORE INSTRUCTIONS	31
TABLE 48. BASELINE V3 AND V3M INSTRUCTIONS	32
TABLE 49. BASELINE V1/V2 INSTRUCTIONS EXCLUDED FROM V3M PROFILE	33
TABLE 50. PERFORMANCE EXTENSION V1 INSTRUCTIONS	36
TABLE 51. PERFORMANCE EXTENSION V2 INSTRUCTIONS	37
TABLE 52. STRING EXTENSION INSTRUCTIONS	38
TABLE 53. CCTL SUBTYPE ENCODING	57
TABLE 54. CCTL SUBTYPE DEFINITIONS	57
TABLE 55. GROUP 0 MFUSR DEFINITIONS	126
TABLE 56. GROUP 1 MFUSR DEFINITIONS	127
TABLE 57. GROUP 2 MFUSR DEFINITIONS	127
TABLE 58. MSYNC SUBTYPE DEFINITIONS	133
TABLE 59. GROUP 0 MTUSR DEFINITIONS	137
TABLE 60. GROUP 1 MTUSR DEFINITIONS	138
TABLE 61. GROUP 2 MTUSR DEFINITIONS	139
TABLE 62. STANDBY INSTRUCTION SUBTYPE DEFINITIONS	185
TABLE 63. TLBOP SUBTYPE DEFINITIONS	201
TABLE 64. N8 PRODUCER-CONSUMER LATENCY	399
TABLE 65. N8 INSTRUCTION EXECUTION LATENCY	400
TABLE 66. N8 DATA HAZARD PENALTY	402
TABLE 67. N8 MISCELLANEOUS PENALTY	403
TABLE 68. N12 IMPLEMENTATION OF CCTL INSTRUCTION	405

List of Figures

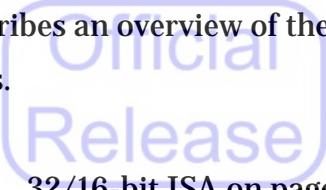
FIGURE 1. INDEX TYPE FORMAT FOR RA OF CCTL INSTRUCTION	60
FIGURE 2. STATE DIAGRAM OF THE LOCK FLAG OF A CACHE LINE CONTROLLED BY CCTL.....	66
FIGURE 3. BASIC BSE OPERATION WITH RB(30) == 0	232
FIGURE 4. BASIC BSE OPERATION WITH RB(30) == 1	233
FIGURE 5. BSE OPERATION EXTRACTING ALL REMAINING BITS WITH RB(30) == 0	233
FIGURE 6. BSE OPERATION WITH THE “UNDERFLOW” CONDITION WITH RB(30) == 0.....	234
FIGURE 7. BASIC BSP OPERATION.....	238
FIGURE 8. BSP OPERATION WITH RB(30) == 1.....	239
FIGURE 9. BSP OPERATION FILLING UP RT.....	239
FIGURE 10. BSP OPERATION WITH THE “OVERFLOW” CONDITION.	240

Typographical Convention Index

Document Element	Font	Font Style	Size	Color
Normal	Georgia	Normal	12	Black
Code	Lucida Console	Normal	11	Indigo
USER VARIABLE	Lucida Console	ALL-CAPS	11	INDIGO
Note/Warning	Georgia	Normal	12	Red
<u>Hyperlink</u>	Georgia	<u>Bold + Underlined</u>	12	Blue

1. Introduction

This manual provides detailed descriptions of the AndeStar™ Instruction Set Architecture (ISA). This chapter describes an overview of the AndeStar instruction set architecture and contains the following sections.

- 
- A faint watermark in the center of the page reads "Official Release" in a light blue, rounded rectangular box.
- 1.1 32/16-bit ISA on page 2
 - 1.2 Architecture versions and extended instruction sets on page 2
 - 1.3 Data Types on page 3
 - 1.4 Registers on page 4
 - 1.5 Instruction Encoding on page 8
 - 1.6 Miscellaneous on page 9

1.1. 32/16-bit ISA

In order to achieve optimal system performance, code density and power efficiency, a set of (32/16-bit) mixed-length instructions has been implemented for the AndeStar ISA.

The AndeStar 32/16-bit mixed-length ISA has the following features:

1. The 32-bit and 16-bit instructions can be freely mixed in a program.
2. Most of 16-bit instructions are a frequently-used subset of 32-bit instructions.
3. No 32/16-bit mode switching performance penalty when executing mixed 32-bit and 16-bit instructions.
4. The 32/16-bit mixed-length ISA is in a big-endian format.
5. Most of the instructions are RISC-style register-based instructions while some instructions are combinations of a few RISC-style register-based operations.
6. 5-bit register index in 32-bit instruction format.
7. 5/4/3-bit register index in 16-bit instruction format.
8. The ISA provides hardware acceleration for a mixed-endian environment.

1.2. Architecture Versions and Extended instruction sets

Three versions of the baseline instruction set have been defined, denoted by the version numbers 1 to 3:

- Baseline Version 1 gives a basic instruction set.
- Baseline Version 2 extends Baseline Version 1 as follows:
 - Global pointer (GP) and stack pointer (SP) implied access
 - General purpose register based division, multiplication, and multiplication and accumulation.
- Baseline Version 3 extends Baseline Version 2 as follows:
 - Combine ALU and SHIFT into one single instruction
 - Branch on Equal/NotEqual Constant
 - Combine typical prolog/epilog instructions into one single instruction
 - For frequently used 32-bit instructions, add their 16-bit instruction mappings

- Performance enhancement for thread local storage

Beside baseline instruction sets, 4 extended instruction sets are defined for different purposes:

- Performance Extension
- Floating-point Extension
- Audio DSP Extension
- String Processing Extension



1.3. Data Types

The AndeStar ISA supports the following data types:

1. Integer
 - Bit (1-bit, b)
 - Byte (8-bit, B)
 - Halfword (16-bit, H)
 - Word (32-bit, W)
2. Floating-point Extension
 - 32-bit Single Precision Floating-point (32-bit, S)
 - 64-bit Double Precision Floating-point (64-bit, D)

1.4. Registers

The AndeStar 32-bit instructions can access thirty-two 32-bit General Purpose Registers (GPR) r0 to r31 and four 32-bit User Special Registers (USR) d0.lo, d0.hi, d1.lo, and d1.hi. The four 32-bit USRs can be combined into two 64-bit accumulator registers and store the multiplication result of two 32-bit numbers.

For the AndeStar 16-bit instructions, a register index can be 5-bit, 4-bit, or 3-bit. The 3-bit and 4-bit register operand field can only access a subset of the thirty-two GPRs. Table 1 demonstrates the 5/4/3-bit register sets, the mapping of index number to register number, and the software usage convention of the AndeStar tool chains.

Table 1. AndeStar General Purpose Registers

Register	32/16-bit (5)	16-bit (4)	16-bit (3)	Comments
r0	a0	h0	o0	
r1	a1	h1	o1	
r2	a2	h2	o2	
r3	a3	h3	o3	
r4	a4	h4	o4	
r5	a5	h5	o5	Implied register for beqs38 and bnes38
r6	s0	h6	o6	Saved by callee
r7	s1	h7	o7	Saved by callee
r8	s2	h8		Saved by callee
r9	s3	h9		Saved by callee
r10	s4	h10		Saved by callee
r11	s5	h11		Saved by callee
r12	s6			Saved by callee
r13	s7			Saved by callee
r14	s8			Saved by callee

Register	32/16-bit (5)	16-bit (4)	16-bit (3)	Comments
r15	ta			Temporary register for assembler Implied register for slt(s i)45, b[eq ne]zs8
r16	t0	h12		Saved by caller
r17	t1	h13		Saved by caller
r18	t2	h14		Saved by caller
r19	t3	h15		Saved by caller
r20	t4			Saved by caller
r21	t5			Saved by caller
r22	t6			Saved by caller
r23	t7			Saved by caller
r24	t8			Saved by caller
r25	t9			Saved by caller
r26	p0			Reserved for Privileged-mode use.
r27	p1			Reserved for Privileged-mode use
r28	s9/fp			Frame pointer / Saved by callee
r29	gp			Global pointer
r30	lp			Link pointer
r31	sp			Stack pointer

Four USRs can only be accessed by 32-bit instructions. Their names and usages are listed in Table 2.

Table 2. AndeStar User Special Registers

Register	32-bit Instr.	16-bit Instr.	Comments
D0	d0.{hi, lo}	N/A	For multiplication and division related instructions (64-bit)
D1	d1.{hi, lo}	N/A	For multiplication and division related instructions (64-bit)

The AndeStar ISA assumes an implied Program Counter (PC) which records the address of the currently executing instruction. The value of this implied PC can be read out using the MFUSR instruction and changed by the control flow changing instructions such as branches and jumps. In addition to the implied PC, the AndeStar ISA also assumes an implied endian mode bit (PSW.BE) to provide hardware acceleration for accessing mixed-endian data in memory. The PSW.BE affects the endian behavior of the load/store instruction and its value, though being not able to be read, can be changed using the SETEND instruction.

Table 3. AndeStar Status Registers

Register	32-bit Instr.	16-bit Instr.	Comments
PC	implied	implied	Affected by control flow changing instructions
PSW.BE	implied	implied	Affected by SETEND instruction.

1.4.1. Reduced Register Configuration Option

For small systems which are more sensitive to cost, AndeStar ISA architecture provides a configuration option to reduce the total number of general purpose registers to 16. In this “Reduced Register” configuration, r11-r14, and r16-r27 are not available; the use of an unimplemented register in an instruction causes a Reserved Instruction exception. Table 4 lists the general purpose registers that can be used by instructions in this configuration.

Table 4. Registers for Reduced Register Configuration

Register	32/16-bit (5)	16-bit (4)	16-bit (3)	Comments
r0	a0	h0	o0	
r1	a1	h1	o1	
r2	a2	h2	o2	
r3	a3	h3	o3	
r4	a4	h4	o4	
r5	a5	h5	o5	Implied register for beqs38 and bnes38
r6	s0	h6	o6	Saved by callee
r7	s1	h7	o7	Saved by callee
r8	s2	h8		Saved by callee
r9	s3	h9		Saved by callee
r10	s4	h10		Saved by callee
r15	ta			Temporary register for assembler Implied register for slt(s i)45, b[eq ne]zs8
r28	fp			Frame pointer / Saved by callee
r29	gp			Global pointer
r30	lp			Link pointer
r31	sp			Stack pointer

1.5. Instruction Encoding

Bit [31] of a 32-bit instruction and Bit [15] of a 16-bit instruction distinguish between 32-bit and 16-bit instructions:

Bit [31] = 0 => 32-bit instructions

Bit [15] = 1 => 16-bit instructions



1.5.1. 32-bit Instruction Format

A typical 32-bit instruction contains the following three fields:

0	30:25 (6)	24:0 (25)
---	-----------	-----------

Bit [31] = 0 => 32-bit ISA

Bit [30:25] => OPC [5:0]

Bit [24:0] => Operand / Immediate / Sub-op

For the detailed encoding information, please refer to individual instructions.

1.5.2. 16-bit Instruction Format

A typical 16-bit instruction contains the following two fields:

1	14:0 (15)
---	-----------

Bit [15] = 1 => 16-bit ISA

Bit [14:0] => Opcode / Operand / Immediate

For the detailed encoding information, please refer to individual instructions.

1.6. Miscellaneous

- Instruction addressing is halfword aligned.
- Integer branch instructions will NOT use conditional codes (but use GPR instead).
- Branch/Jump types
 - Branch on Registers
 - Branch destination
 - PC + Immediate Offset
 - Jump destination
 - PC + Immediate Offset
 - Register
 - Branch/Jump and link
- The immediate values of load/store word/halfword are shifted.
 - lw rt, [ra, imm], addr=ra+(imm << 2)
 - lh rt, [ra, imm], addr=ra+(imm << 1)
- Register remapping for 16-bit instructions
 - 32 registers (32-bit instructions) -> 32/16/8 registers (16-bit instructions)
- The word “implementation” means the design of specified processors.

2. 32-bit Baseline V1 Instructions

This chapter describes the 32-bit baseline V1 instructions in the following sections:

- 
- 2.1 Data-processing Instructions on page 11
 - 2.2 Load and Store Instructions on page 14
 - 2.3 Jump and Branch Instructions on page 18
 - 2.4 Privileged Resource Access Instructions on page 19
 - 2.5 Miscellaneous Instructions on page 21

2.1. Data-processing Instructions

Table 5. ALU Instruction with Immediate (Baseline V1)

Mnemonic	Instruction	Operation
ADDI rt5, ra5, imm15s	Add Immediate	$rt5 = ra5 + SE(imm15s)$
SUBRI rt5, ra5, imm15s	Subtract Reverse Immediate	$rt5 = SE(imm15s) - ra5$
ANDI rt5, ra5, imm15u	And Immediate	$rt5 = ra5 \& ZE(imm15u)$
ORI rt5, ra5, imm15u	Or Immediate	$rt5 = ra5 ZE(imm15u)$
XORI rt5, ra5, imm15u	Exclusive Or Immediate	$rt5 = ra5 ^ ZE(imm15u)$
SLTI rt5, ra5, imm15s	Set on Less Than Immediate	$rt5 = (ra5 \text{ (unsigned)} < SE(imm15s)) ? 1 : 0$
SLTSI rt5, ra5, imm15s	Set on Less Than Signed Immediate	$rt5 = (ra5 \text{ (signed)} < SE(imm15s)) ? 1 : 0$
MOVI rt5, imm20s	Move Immediate	$rt5 = SE(imm20s)$
SETHI rt5, imm20u	Set High Immediate	$rt5 = \{imm20u, 12'b0\}$

Table 6. ALU Instruction (Baseline V1)

Mnemonic	Instruction	Operation
ADD rt5, ra5, rb5	Add	$rt5 = ra5 + rb5$
SUB rt5, ra5, rb5	Subtract	$rt5 = ra5 - rb5$
AND rt5, ra5, rb5	And	$rt5 = ra5 \& rb5$
NOR rt5, ra5, rb5	Nor	$rt5 = \sim(ra5 rb5)$
OR rt5, ra5, rb5	Or	$rt5 = ra5 rb5$
XOR rt5, ra5, rb5	Exclusive Or	$rt5 = ra5 ^ rb5$
SLT rt5, ra5, rb5	Set on Less Than	$rt5 = (ra5 \text{ (unsigned)} < rb5) ? 1 : 0$
SLTS rt5, ra5, rb5	Set on Less Than Signed	$rt5 = (ra5 \text{ (signed)} < rb5) ? 1 : 0$
SVA rt5, ra5, rb5	Set on Overflow Add	$rt5 = ((ra5 + rb5) \text{ overflow})? 1 : 0$
SVS rt5, ra5, rb5	Set on Overflow Subtract	$rt5 = ((ra5 - rb5) \text{ overflow}))? 1 : 0$
SEB rt5, ra5	Sign Extend Byte	$rt5 = SE(ra5[7:0])$

Mnemonic	Instruction	Operation
SEH rt5, ra5	Sign Extend Halfword	$rt5 = SE(ra5[15:0])$
ZEB rt5, ra5 (alias of ANDI rt5, ra5, 0xFF)	Zero Extend Byte	$rt5 = ZE(ra5[7:0])$
ZEH rt5, ra5	Zero Extend Halfword	$rt5 = ZE(ra5[15:0])$
WSBH rt5, ra5	Word Swap Byte within Halfword	$rt5 = \{ra5[23:16], ra5[31:24], ra5[7:0], ra5[15:8]\}$

Table 7. Shifter Instruction (Baseline V1)

Mnemonic	Instruction	Operation
SLLI rt5, ra5, imm5u	Shift Left Logical Immediate	$rt5 = ra5 << imm5u$
SRLI rt5, ra5, imm5u	Shift Right Logical Immediate	$rt5 = ra5 \text{ (logic)} >> imm5u$
SRAI rt5, ra5, imm5u	Shift Right Arithmetic Immediate	$rt5 = ra5 \text{ (arith)} >> imm5u$
ROTRI rt5, ra5, imm5u	Rotate Right Immediate	$rt5 = ra5 >> imm5u$
SLL rt5, ra5, rb5	Shift Left Logical	$rt5 = ra5 << rb5(4,0)$
SRL rt5, ra5, rb5	Shift Right Logical	$rt5 = ra5 \text{ (logic)} >> rb5(4,0)$
SRA rt5, ra5, rb5	Shift Right Arithmetic	$rt5 = ra5 \text{ (arith)} >> rb5(4,0)$
ROTR rt5, ra5, rb5	Rotate Right	$rt5 = ra5 >> rb5(4,0)$

Table 8. Multiply Instruction (Baseline V1)

Mnemonic	Instruction	Operation
MUL rt5, ra5, rb5	Multiply Word to Register	$rt5 = ra5 * rb5$
MULTS64 d1, ra5, rb5	Multiply Word Signed	$d1 = ra5 \text{ (signed)} * rb5$
MULT64 d1, ra5, rb5	Multiply Word	$d1 = ra5 \text{ (unsigned)} * rb5$
MADDS64 d1, ra5, rb5	Multiply and Add Signed	$d1 = d1 + ra5 \text{ (signed)} * rb5$
MADD64 d1, ra5, rb5	Multiply and Add	$d1 = d1 + ra5 \text{ (unsigned)} * rb5$
MSUBS64 d1, ra5, rb5	Multiply and Subtract	$d1 = d1 - ra5 \text{ (signed)} * rb5$

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Mnemonic	Instruction	Operation
	Signed	
MSUB64 d1, ra5, rb5	Multiply and Subtract	$d1 = d1 - ra5 \text{ (unsigned)} * rb5$
MULT32 d1, ra5, rb5	Multiply Word	$d1.LO = ra5 * rb5$
MADD32 d1, ra5, rb5	Multiply and Add	$d1.LO = d1.LO + ra5 * rb5$
MSUB32 d1, ra5, rb5	Multiply and Subtract	$d1.LO = d1.LO - ra5 * rb5$
MFUSR rt5, USR	Move From User Special Register	$rt5 = USReg[USR]$
MTUSR rt5, USR	Move To User Special Register	$USReg[USR] = rt5$

Table 9. Divide Instructions (Baseline V1)

Mnemonic	Instruction	Operation
DIV Dt, ra5, rb5	Unsigned Integer Divide	$Dt.L = ra5 \text{ (unsigned)} / rb5;$ $Dt.H = ra5 \text{ (unsigned)} \bmod rb5;$
DIVS Dt, ra5, rb5	Signed Integer Divide	$Dt.L = ra5 \text{ (signed)} / rb5;$ $Dt.H = ra5 \text{ (signed)} \bmod rb5;$

2.2. Load and Store Instructions

Table 10. Load / Store Addressing Mode

Mode	Operand Type	Index Left Shift (0-3 bits)	Before Increment with Base Update
1	Base Register + Immediate	No	No
2	Base Register + Immediate	No	Yes
3	Base Register + Register	Yes	No
4	Base Register + Register	Yes	Yes

Table 11. Load / Store Instruction (Baseline)

Mnemonic	Instruction	Operation
LWI rt5, [ra5 + (imm15s << 2)]	Load Word Immediate	address = ra5 + SE(imm15s << 2) rt5 = Word-memory(address)
LHI rt5, [ra5 + (imm15s << 1)]	Load Halfword Immediate	address = ra5 + SE(imm15s << 1) rt5 = ZE(Halfword-memory(address))
LHSI rt5, [ra5 + (imm15s << 1)]	Load Halfword Signed Immediate	address = ra5 + SE(imm15s << 1) rt5 = SE(Halfword-memory(address))
LBI rt5, [ra5 + imm15s]	Load Byte Immediate	address = ra5 + SE(imm15s) rt5 = ZE(Byte-memory(address))
LBSI rt5, [ra5 + imm15s]	Load Byte Signed Immediate	address = ra5 + SE(imm15s) rt5 = SE(Byte-memory(address))
SWI rt5, [ra5 + (imm15s << 2)]	Store Word Immediate	address = ra5 + SE(imm15s << 2) Word-memory(address) = rt5
SHI rt5, [ra5 + (imm15s << 1)]	Store Halfword Immediate	address = ra5 + SE(imm15s << 1) Halfword-memory(address) = rt5[15:0]
SBI rt5, [ra5 + imm15s]	Store Byte Immediate	address = ra5 + SE(imm15s) Byte-memory(address) = rt5[7:0]

Table 12. Load / Store Instruction (Baseline)

Mnemonic	Instruction	Operation
LWI.bi rt5, [ra5], (imm15s << 2)	Load Word Immediate with Post Increment	rt5 = Word-memory(ra5) ra5 = ra5 + SE(imm15s << 2)
LHI.bi rt5, [ra5], (imm15s << 1)	Load Halfword Immediate with Post Increment	rt5 = ZE(Halfword-memory(ra5)) ra5 = ra5 + SE(imm15s << 1)
LHSI.bi rt5, [ra5], (imm15s << 1)	Load Halfword Signed Immediate with Post Increment	rt5 = SE(Halfword-memory(ra5)) ra5 = ra5 + SE(imm15s << 1)
LBI.bi rt5, [ra5], imm15s	Load Byte Immediate with Post Increment	rt5 = ZE(Byte-memory(ra5)) ra5 = ra5 + SE(imm15s)
LBSI.bi rt5, [ra5], imm15s	Load Byte Signed Immediate with Post Increment	rt5 = SE(Byte-memory(ra5)) ra5 = ra5 + SE(imm15s)
SWI.bi rt5, [ra5], (imm15s << 2)	Store Word Immediate with Post Increment	Word-memory(ra5) = rt5 ra5 = ra5 + SE(imm15s << 2)
SHI.bi rt5, [ra5], (imm15s << 1)	Store Halfword Immediate with Post Increment	Halfword-memory(ra5) = rt5[15:0] ra5 = ra5 + SE(imm15s << 1)
SBI.bi rt5, [ra5], imm15s	Store Byte Immediate with Post Increment	Byte-memory(ra5) = rt5[7:0] ra5 = ra5 + SE(imm15s)

Table 13. Load / Store Instruction (Baseline)

Mnemonic	Instruction	Operation
LW rt5, [ra5 + (rb5 << sv)]	Load Word	address = ra5 + (rb5 << sv) rt5 = Word-memory(address)
LH rt5, [ra5 + (rb5 << sv)]	Load Halfword	address = ra5 + (rb5 << sv) rt5 = ZE(Halfword-memory(address))
LHS rt5, [ra5 + (rb5 << sv)]	Load Halfword Signed	address = ra5 + (rb5 << sv) rt5 = SE(Halfword-memory(address))
LB rt5, [ra5 + (rb5 << sv)]	Load Byte	address = ra5 + (rb5 << sv) rt5 = ZE(Byte-memory(address))

Mnemonic	Instruction	Operation
LBS rt5, [ra5 + (rb5 << sv)]	Load Byte Signed	address = ra5 + (rb5 << sv) rt5 = SE(Byte-memory(address))
SW rt5, [ra5 + (rb5 << sv)]	Store Word	address = ra5 + (rb5 << sv) Word-memory(address) = rt5
SH rt5, [ra5 + (rb5 << sv)]	Store Halfword	address = ra5 + (rb5 << sv) Halfword-memory(address) = rt5[15:0]
SB rt5, [ra5 + (rb5 << sv)]	Store Byte	address = ra5 + (rb5 << sv) Byte-memory(address) = rt5[7:0]

Table 14. Load / Store Instruction (Baseline)

Mnemonic	Instruction	Operation
LW.bi rt5, [ra5], rb5<<sv	Load Word with Post Increment	rt5 = Word-memory(ra5) ra5 = ra5 + (rb5 << sv)
LH.bi rt5, [ra5], rb5<<sv	Load Halfword with Post Increment	rt5 = ZE(Halfword-memory(ra5)) ra5 = ra5 + (rb5 << sv)
LHS.bi rt5, [ra5], rb5<<sv	Load Halfword Signed with Post Increment	rt5 = SE(Halfword-memory(ra5)) ra5 = ra5 + (rb5 << sv)
LB.bi rt5, [ra5], rb5<<sv	Load Byte with Post Increment	rt5 = ZE(Byte-memory(ra5)) ra5 = ra5 + (rb5 << sv)
LBS.bi rt5, [ra5], rb5<<sv	Load Byte Signed with Post Increment	rt5 = SE(Byte-memory(ra5)) ra5 = ra5 + (rb5 << sv)
SW.bi rt5, [ra5], rb5<<sv	Store Word with Post Increment	Word-memory(ra5) = rt5 ra5 = ra5 + (rb5 << sv)
SH.bi rt5, [ra5], rb5<<sv	Store Halfword with Post Increment	Halfword-memory(ra5) = rt5[15:0] ra5 = ra5 + (rb5 << sv)
SB.bi rt5, [ra5], rb5<<sv	Store Byte with Post Increment	Byte-memory(ra5) = rt5[7:0] ra5 = ra5 + (rb5 << sv)

Table 15. Load / Store Multiple Word Instruction (Baseline)

Mnemonic	Instruction	Operation
LMW.{b a}{i d}{m?} Rb5, [Ra5], Re5, Enable4	Load Multiple Word (before/after; in/decrement; update/no-update base)	See page 111 for details.
SMW.{b a}{i d}{m?} Rb5, [Ra5], Re5, Enable4	Store Multiple Word (before/after; in/decrement; update/no-update base)	See page 176 for details.

Table 16. Load / Store Instruction for Atomic Updates (Baseline)

Mnemonic	Instruction	Operation
LLW rt5, [ra5 + (rb5 << sv)]	Load Locked Word	See page 108 for details.
SCW rt5, [ra5 + (rb5 << sv)]	Store Condition Word	See page 157 for details.

Table 17. Load / Store Instructions with User-mode Privilege

Mnemonic	Instruction	Operation
LWUP rt5, [ra5 + (rb5 << sv)]	Load Word with User-mode Privilege Translation	Equivalent to the LW instruction but with the user mode privilege address translation. See page 120 for details.
SWUP rt5, [ra5 + (rb5 << sv)]	Store Word with User-mode Privilege Translation	Equivalent to SW instruction but with the user mode privilege address translation. See page 196 for details.

2.3. Jump and Branch Instructions

Table 18. Jump Instruction (Baseline)

Mnemonic	Instruction	Operation
J imm24s	Jump	$PC = PC + SE(\text{imm24s} \ll 1)$
JAL imm24s	Jump and Link	$LP = \text{next sequential PC } (PC + 4);$ $PC = PC + SE(\text{imm24s} \ll 1)$
JR rb5	Jump Register	$PC = rb5$
RET rb5	Return from Register	$PC = rb5$
JRAL rb5	Jump Register and Link	$jaddr = rb5;$
JRAL rt5, rb5		$LP = PC + 4; \text{ or } rt5 = PC + 4;$ $PC = jaddr;$

Table 19. Branch Instruction (Baseline)

Mnemonic	Instruction	Operation
BEQ rt5, ra5, imm14s	Branch on Equal (2 Register)	$PC = (rt5 == ra5)? (PC + SE(\text{imm14s} \ll 1)) : (PC + 4)$
BNE rt5, ra5, imm14s	Branch on Not Equal (2 Register)	$PC = (rt5 != ra5)? (PC + SE(\text{imm14s} \ll 1)) : (PC + 4)$
BEQZ rt5, imm16s	Branch on Equal Zero	$PC = (rt5 == 0)? (PC + SE(\text{imm16s} \ll 1)) : (PC + 4)$
BNEZ rt5, imm16s	Branch on Not Equal Zero	$PC = (rt5 != 0)? (PC + SE(\text{imm16s} \ll 1)) : (PC + 4)$
BGEZ rt5, imm16s	Branch on Greater than or Equal to Zero	$PC = (rt5 (\text{signed}) \geq 0)? (PC + SE(\text{imm16s} \ll 1)) : (PC + 4)$
BLTZ rt5, imm16s	Branch on Less than Zero	$PC = (rt5 (\text{signed}) < 0)? (PC + sign-ext(\text{imm16s} \ll 1)) : (PC + 4)$
BGTZ rt5, imm16s	Branch on Greater than Zero	$PC = (rt5 (\text{signed}) > 0)? (PC + SE(\text{imm16s} \ll 1)) : (PC + 4)$
BLEZ rt5, imm16s	Branch on Less than or Equal to Zero	$PC = (rt5 (\text{signed}) \leq 0)? (PC + SE(\text{imm16s} \ll 1)) : (PC + 4)$

Table 20. Branch with Link Instruction (Baseline)

Mnemonic	Instruction	Operation
BGEZAL rt5, imm16s	Branch on Greater than or Equal to Zero and Link	LP = next sequential PC (PC + 4); PC = (rt5 (signed) >= 0)? (PC + SE(imm16s << 1)), (PC + 4);
BLTZAL rt5, imm16s	Branch on Less than Zero and Link	LP = next sequential PC (PC + 4); PC = (rt5 (signed) < 0)? (PC + SE(imm16s << 1)), (PC + 4);

2.4. Privilege Resource Access Instructions

Table 21. Read/Write System Registers (Baseline)

Mnemonic	Instruction	Operation
MFSR rt5, SRIDX	Move from System Register	rt5 = SR[SRIDX]
MTSR rt5, SRIDX	Move to System Register	SR[SRIDX] = rt5

Table 22. Jump Register with System Register Update (Baseline)

Mnemonic	Instruction	Operation
JR.ITOFF rb5	Jump Register and Instruction Translation OFF	PC = rb5; PSW.IT = 0;
JR.TOFF rb5	Jump Register and Translation OFF	PC = rb5; PSW.IT = 0, PSW.DT = 0;
JRAL.ITON rb5 JRAL.ITON rt5, rb5	Jump Register and Link and Instruction Translation ON	jaddr = rb5; LP = PC+4 or rt5 = PC+4; PC = jaddr; PSW.IT = 1;

Mnemonic	Instruction	Operation
JRAL.TON rb5	Jump Register and Link and Translation ON	jaddr = rb5; LP = PC+4 or rt5 = PC+4;
JRAL.TON rt5, rb5		PC = jaddr; PSW.IT = 1, PSW.DT = 1;

Table 23. MMU Instruction (Baseline)

Mnemonic	Instruction	Operation
TLBOP Ra, TargetRead (TRD)	Read targeted TLB entry	See page 202 for details.
TLBOP Ra, TargetWrite (TWR)	Write targeted TLB entry	See page 202 for details.
TLBOP Ra, RWrite (RWR)	Write PTE into a TLB entry	See page 203 for details.
TLBOP Ra, RWriteLock (RWLK)	Write PTE into a TLB entry and lock	See page 204 for details.
TLBOP Ra, Unlock (UNLK)	Unlock a TLB entry	See page 204 for details.
TLBOP Rt, Ra, Probe (PB)	Probe TLB entry	See page 205 for details.
TLBOP Ra, Invalidate (INV)	Invalidate TLB entries	Invalidate the TLB entry containing VA stored in Rx.
TLBOP FlushAll (FLUA)	Flush all TLB entries except locked entries	See page 206 for details.
LD_VLPT	Load VLPT page table (optional instruction)	Load the VLPT page table which always goes through data TLB translation. On a TLB miss, generate a double TLB miss exception.

2.5. Miscellaneous Instructions

Table 24. Conditional Move (Baseline)

Mnemonic	Instruction	Operation
CMOVZ rt5, ra5, rb5	Conditional Move on Zero	rt5 = ra5 if (rb5 == 0)
CMOVN rt5, ra5, rb5	Conditional Move on Not Zero	rt5 = ra5 if (rb5 != 0)

Table 25. Synchronization Instruction (Baseline)

Mnemonic	Instruction	Operation
MSYNC	Memory Synchronize	Synchronize Shared Memory
ISYNC	Instruction Synchronize	Synchronize Caches to Make Instruction Stream Writes Effective

Table 26. Prefetch Instruction (Baseline)

Mnemonic	Instruction	Operation
DPREFI [ra5 + imm15s]	Data Prefetch Immediate	See page 69 for details.
DPREF [ra5 + (rb5 << si)]	Data Prefetch	

Table 27. NOP Instruction (Baseline)

Mnemonic	Instruction	Operation
NOP (alias of SRLI R0, R0, 0)	No Operation	No Operation

Table 28. Serialization Instruction (Baseline)

Mnemonic	Instruction	Operation
DSB	Data Serialization Barrier	See page 73 for details.
ISB	Instruction Serialization Barrier	See page 79 for details.

Table 29. Exception Generation Instruction (Baseline)

Mnemonic	Instruction	Operation
BREAK	Breakpoint	See page 55 for details.
SYSCALL	System Call	See page 198 for details.
TRAP	Trap Always	See page 209 for details.
TEQZ	Trap on Equal Zero	See page 199 for details.
TNEZ	Trap on Not Equal Zero	See page 200 for details.

Table 30. Special Return Instruction (Baseline)

Mnemonic	Instruction	Operation
IRET	Interruption Return	Return from Interruption (exception or interrupt). Please see page 76.
RET.ITOFF Rb5	Return and turn off instruction address translation	PC = Rb5, PSW.IT = 0 (page 149)
RET.TOFF Rb5	Return and turn off address translation (instruction/data)	PC = Rb5, PSW.IT = 0, PSW.DT = 0 (page 149)

Table 31. Cache Control Instruction (Baseline)

Mnemonic	Instruction	Operation
CCTL	Cache Control	Read, write, and control cache states. Please see page 56.

Table 32. Miscellaneous Instructions (Baseline)

Mnemonic	Instruction	Operation
SETEND.B	Atomic set or clear of PSW.BE bit	PSW.BE = 1; // SETEND.B
SETEND.L	PSW.BE bit	PSW.BE = 0; // SETEND.L
SETGIE.E	Atomic set or clear of PSW.GIE bit	PSW.GIE = 1; // SETGIE.E
SETGIE.D	PSW.GIE bit	PSW.GIE = 0; // SETGIE.D

Mnemonic	Instruction	Operation
STANDBY	Wait for External Event	Enter standby state and wait for external events. Please see page 185.



3. 16-bit Baseline V1 Instructions

The AndeStar 16-bit baseline V1 instruction set is a subset of the 32-bit baseline V1 instruction set. That is, every 16-bit instruction can be properly mapped onto a corresponding 32-bit instruction and the mappings are listed in Section 3.1.

3.1. 16-bit to 32-bit Instruction Mapping

Table 33. Move Instruction (16-bit)

Mnemonic	Instruction	32-bit Operation	
MOVI55 rt5, imm5s	Move Immediate	MOVI	rt5, SE(imm5s)
MOV55 rt5, ra5	Move	ADDI/ORI	rt5, ra5, 0

Table 34. Add/Sub Instruction with Immediate (16-bit)

Mnemonic	Instruction	32-bit Operation	
ADDI45 rt4, imm5u	Add Word Immediate	ADDI	rt5, rt5, ZE(imm5u)
ADDI333 rt3, ra3, imm3u	Add Word Immediate	ADDI	rt5, ra5, ZE(imm3u)
SUBI45 rt4, imm5u	Subtract Word Immediate	ADDI	rt5, rt5, NEG(imm5u)
SUBI333 rt3, ra3, imm3u	Subtract Word Immediate	ADDI	rt5, ra5, NEG(imm3u)

Table 35. Add/Sub Instruction (16-bit)

Mnemonic	Instruction	32-bit Operation	
ADD45 rt4, rb5	Add Word	ADD	rt5, rt5, rb5
ADD333 rt3, ra3, rb3	Add Word	ADD	rt5, ra5, rb5
SUB45 rt4, rb5	Subtract Word	SUB	rt5, rt5, rb5
SUB333 rt3, ra3, rb3	Subtract Word	SUB	rt5, ra5, rb5

Table 36. Shift Instruction with Immediate (16-bit)

Mnemonic	Instruction	32-bit Operation
SRAI45 rt4, imm5u	Shift Right Arithmetic Immediate	SRAI rt5, rt5, imm5u
SRLI45 rt4, imm5u	Shift Right Logical Immediate	SRLI rt5, rt5, imm5u
SLLI333 rt3, ra3, imm3u	Shift Left Logical Immediate	SLLI rt5, ra5, ZE(imm3u)

Table 37. Bit Field Mask Instruction with Immediate (16-bit)

Mnemonic	Instruction	32-bit Operation
BFMI333 rt3, ra3, imm3u	Bit Field Mask Immediate	See the following individual mappings.
ZEB33 rt3, ra3 (BFMI333 rt3, ra3, 0)	Zero Extend Byte	ZEB rt5, ra5
ZEH33 rt3, ra3 (BFMI333 rt3, ra3, 1)	Zero Extend Halfword	ZEH rt5, ra5
SEB33 rt3, ra3 (BFMI333 rt3, ra3, 2)	Sign Extend Byte	SEB rt5, ra5
SEH33 rt3, ra3 (BFMI333 rt3, ra3, 3)	Sign Extend Halfword	SEH rt5, ra5
Extension Set (Non-Baseline)		
XLSB33 rt3, ra3 (BFMI333 rt3, ra3, 4)	Extract LSB	ANDI rt5, ra5, 1
X11B33 rt3, ra3 (BFMI333 rt3, ra3, 5)	Extract the least 11 Bits	ANDI rt5, ra5, 0x7ff

Table 38. Load / Store Instruction (16-bit)

Mnemonic	Instruction	32-bit Operation
LWI450 rt4, [ra5]	Load Word Immediate	LWI rt5, [ra5+0]
LWI333 rt3, [ra_3+ imm3u]	Load Word Immediate	LWI rt5, [ra5+ZE(imm3u)]

Mnemonic	Instruction	32-bit Operation
LWI333.bi rt3, [ra_3], imm3u	Load Word Immediate with Post-increment	LWI.bi rt5, [ra5], ZE(imm3u)
LHI333 rt3, [ra_3+ imm3u]	Load Halfword Immediate	LHI rt5, [ra5+ZE(imm3u)]
LBI333 rt3, [ra_3+ imm3u]	Load Byte Immediate	LBI rt5, [ra5+ZE(imm3u)]
SWI450 rt4, [ra5]	Store Word Immediate	SWI rt5, [ra5+0]
SWI333 rt3, [ra_3+ imm3u]	Store Word Immediate	SWI rt5, [ra5+ZE(imm3u)]
SWI333.bi rt3, [ra_3], imm3u	Store Word Immediate with Post-increment	SWI.bi rt5, [ra5], ZE(imm3u)
SHI333 rt3, [ra_3+ imm3u]	Store Halfword Immediate	SHI rt5, [ra5+ZE(imm3u)]
SBI333 rt3, [ra_3+ imm3u]	Store Byte Immediate	SBI rt5, [ra5+ZE(imm3u)]

Table 39. Load/Store Instruction with Implied FP (16-bit)

Mnemonic	Instruction	32-bit Operation
LWI37 rt3, [fp+imm7u]	Load Word with Implied FP	LWI rt5, [fp+ZE(imm7u)]
SWI37 rt3, [fp+imm7u]	Store Word with Implied FP	SWI rt5, [fp+ZE(imm7u)]

Table 40. Branch and Jump Instruction (16-bit)

Mnemonic	Instruction	32-bit Operation
BEQS38 rt3, imm8s	Branch on Equal (implied r5)	BEQ rt5, r5, SE(imm8s) (next sequential PC = PC + 2)
BNES38 rt3, imm8s	Branch on Not Equal (implied r5)	BNE rt5, r5, SE(imm8s) (next sequential PC = PC + 2)

Mnemonic	Instruction	32-bit Operation	
BEQZ38 rt3, imm8s	Branch on Equal Zero	BEQZ	rt5, SE(imm8s) (next sequential PC = PC + 2)
BNEZ38 rt3, imm8s	Branch on Not Equal Zero	BNEZ	rt5, SE(imm8s) (next sequential PC = PC + 2)
J8 imm8s	Jump Immediate	J	SE(imm8s)
JR5 rb5	Jump Register	JR	rb5
RET5 rb5	Return from Register	RET	rb5
JRAL5 rb5	Jump Register and Link	JRAL	rb5

Table 41. Compare and Branch Instruction (16-bit)

Mnemonic	Instruction	32-bit Operation	
SLTI45 ra4, imm5u	Set on Less Than Unsigned Immediate	SLTI	r15, ra5, ZE(imm5u)
SLTSI45 ra4, imm5u	Set on Less Than Signed Immediate	SLTSI	r15, ra5, ZE(imm5u)
SLT45 ra4, rb5	Set on Less Than Unsigned	SLT	r15, ra5, rb5
SLTS45 ra4, rb5	Set on Less Than Signed	SLTS	r15, ra5, rb5
BEQZS8 imm8s	Branch on Equal Zero (implied r15)	BEQZ	r15, SE(imm8s) (next sequential PC = PC + 2)
BNEZS8 imm8s	Branch on Not Equal Zero (implied r15)	BNEZ	r15, SE(imm8s) (next sequential PC = PC + 2)

Table 42. Misc. Instruction (16-bit)

Mnemonic	Instruction	32-bit Operation	
BREAK16	Breakpoint	BREAK	
NOP16 (alias of SRLI45 R0,0)	No Operation	NOP	

4. 16/32-bit Baseline Version 2 Instructions

Several 16/32-bit instructions are added to the Baseline instruction set. This set of Baseline instructions is defined as Baseline version 2 instruction set to distinguish with the original Baseline V1 instruction set.

These instructions are summarized in the following sections.

- 4.1 16-bit Baseline V2 Instruction on page 29
- 4.2 32-bit Baseline V2 Instruction on page 30

4.1. 16-bit Baseline V2 Instructions

Table 43. ALU Instructions

Mnemonic	Instruction	32-bit Operation
ADDI10.sp imm10s	Add Immediate with implied stack pointer	ADDI sp, sp, SE(imm10s)

Table 44. Load/Store Instruction

Mnemonic	Instruction	32-bit Operation
LWI37.sp rt3, [+ (imm7u << 2)]	Load word Immediate with implied SP	LWI 3T5(Rt3), [SP + ZE(imm7u << 2)]
SWI37.sp rt3, [+ (imm7u << 2)]	Store word Immediate with implied SP	SWI 3T5(Rt3), [SP + ZE(imm7u << 2)]

4.2. 32-bit Baseline V2 Instructions

Table 45. ALU Instructions

Mnemonic	Instruction	32-bit Operation
ADDI.gp rt5, imm19s	GP-implied Add Immediate	$rt5 = gp + SE(imm19s)$

Table 46. Multiply and Divide Instructions

Mnemonic	Instruction	32-bit Operation
MULR64 rt5, ra5, rb5	Multiply unsigned word to registers	$res = ra5 \text{ (unsigned)} * rb5;$ if (PSW.BE == 1) { $(rt5_even, rt5_odd) = res;$ } else { $(rt5_odd, rt5_even) = res;$ }
MULSR64 rt5, ra5, rb5	Multiply signed word to registers	$res = ra5 \text{ (signed)} * rb5;$ if (PSW.BE == 1) { $(rt5_even, rt5_odd) = res;$ } else { $(rt5_odd, rt5_even) = res;$ }
MADDR32 rt5, ra5, rb5	Multiply and add to 32-bit register	$res = ra5 * rb5 ;$ $rt5 = rt5 + res(31,0);$
MSUBR32 rt5, ra5, rb5	Multiply and subtract from 32-bit register	$res = ra5 * rb5 ;$ $rt5 = rt5 - res(31,0);$
DIVR rt5, rs5, ra5, rb5	Unsigned integer divide to registers	$rt5 = \text{Floor}(ra5 \text{ (unsigned)} / rb5);$ $rs5 = ra5 \text{ (unsigned)} \bmod rb5;$
DIVSR rt5, rs5, ra5, rb5	Signed integer divide to registers	$rt5 = \text{Floor}(ra5 \text{ (signed)} / rb5);$ $rs5 = ra5 \text{ (signed)} \bmod rb5;$

Table 47. Load/Store Instructions

Mnemonic	Instruction	32-bit Operation
LBI.gp rt5, [+ imm19s]	GP-implied Load unsigned Byte Immediate	address = gp + SE(imm19s) rt5 = ZE(Byte-memory(address))
LBSI.gp rt5, [+ imm19s]	GP-implied Load signed Byte Immediate	address = gp + SE(imm19s) rt5 = SE(Byte-memory(address))
LHI.gp rt5, [+ (imm18s << 1)]	GP-implied Load unsigned Halfword Immediate	address = gp + SE(imm18s << 1) rt5 = ZE(Halfword-memory(address))
LHSI.gp rt5, [+ (imm18s << 1)]	GP-implied Load signed Halfword Immediate	address = gp + SE(imm18s << 1) rt5 = SE(Halfword-memory(address))
LWI.gp rt5, [+ (imm17s << 2)]	GP-implied Load Word Immediate	address = gp + SE(imm17s << 2) rt5 = Word-memory(address)
SBI.gp rt5, [+ imm19s]	GP-implied Store Byte Immediate	address = gp + SE(imm19s) Byte-memory(address) = rt5[7:0]
SHI.gp rt5, [+ (imm18s << 1)]	GP-implied Store Halfword Immediate	address = gp + SE(imm18s << 1) Halfword-memory(address) = rt5[15:0]
SWI.gp rt5, [+ (imm17s << 2)]	GP-implied Store Word Immediate	address = gp + SE(imm17s << 2) Word-memory(address) = rt5
LMWA.{b a}{i d}{m?} rb5, [ra5], re5, Enable4	Load multiple word with alignment check	Please see page 322 for details.
SMWA.{b a}{i d}{m?} rb5, [ra5], re5, Enable4	Store multiple word with alignment check	Please see page 337 for details.
LBUP Rt, [Ra + (Rb << sv)]	Load Byte with User Privilege	Equivalent to the LB instruction but with the user mode privilege address translation. See page 319 for details.
SBUP Rt, [Ra + (Rb << sv)]	Store Byte with User Privilege	Equivalent to the SB instruction but with the user mode privilege address translation. See page 335 for details.

5. 16/32-bit Baseline Version 3 and Version 3m Instructions

The baseline V3 instructions and V3m instructions are both devised to improve code density.

The baseline V3 instructions further improves TLS implementation efficiency. The baseline V3m instructions include some of the baseline V3 instructions while excluding some of the baseline V1/V2 instructions. See Table 48 for a summary of V3 and V3m instructions.

Table 48. Baseline V3 and V3m Instructions

Mnemonic	Instruction	V3m instruction	16 or 32 bit
ADD_SLLI	Addition with Shift Left Logical Immediate		32-bit
ADD_SRRI	Addition with Shift Right Logical Immediate		32-bit
ADDR36.SP	Add Immediate to Register with Implied Stack Register	Yes	16-bit
ADD5.PC	Add with Implied PC		16-bit
AND_SLLI	Logical And with Shift Left Logical Immediate		32-bit
AND_SRRI	Logical And with Shift Right Logical Immediate		32-bit
AND33	Bit-wise Logical And	Yes	16-bit
BITC	Bit Clear		32-bit
BITCI	Bit Clear Immediate		32-bit
BEQC	Branch on Equal Constant	Yes	32-bit
BNEC	Branch on Not Equal Constant	Yes	32-bit
BMASKI33	Bit Mask Immediate	Yes	16-bit
CCTL L1D_WBALL	L1D Cache Writeback All		32-bit
FEXTI33	Field Extraction Immediate	Yes	16-bit
JRALNEZ	Jump Register and Link on Not Equal Zero		32-bit
JRNEZ	Jump Register on Not Equal Zero		32-bit
LWI45.FE	Load Word Immediate with Implied Function Entry Point (R8)	Yes	16-bit
MOVD44	Move Double Registers	Yes	16-bit

Mnemonic	Instruction	V3m instruction	16 or 32 bit
MOVPI45	Move Positive Immediate	Yes	16-bit
MUL33	Multiply Word to Register	Yes	16-bit
NEG33	Negative	Yes	16-bit
NOT33	Logical Not	Yes	16-bit
OR_SLLI	Logical Or with Shift Left Logical Immediate		32-bit
OR_SRRI	Logical Or with Shift Right Logical Immediate		32-bit
OR33	Bit-wise Logical Or	Yes	16-bit
POP25	Pop Multiple Words from Stack and Return	Yes	16-bit
PUSH25	Push Multiple Words to Stack and Set Function Entry Point (R8)	Yes	16-bit
SUB_SLLI	Subtraction with Shift Left Logical Immediate		32-bit
SUB_SRRI	Subtraction with Shift Right Logical Immediate		32-bit
XOR_SLLI	Logical Exclusive Or with Shift Left Logical Immediate		32-bit
XOR_SRRI	Logical Exclusive Or with Shift Right Logical Immediate		32-bit
XOR33	Bit-wise Logical Exclusive Or	Yes	16-bit

Table 49 lists the baseline V1/V2 instructions excluded from the V3m profile.

Table 49. Baseline V1 / V2 Instructions Excluded from V3m Profile

Inst Type	Insns excluded from V3m	Baseline
lmw/smw	lmwa, smwa	V2
	unaligned data access for lmw and smw	V1
move	mtusr	V1
	mfusr	V1
D related	All D register related instructions	V1
Mul/Div	mulr64	V2
	mulsr64	V2
system	cctl	V1

Inst Type	Insns excluded from V3m	Baseline
	tlbop	V1
	dpref	V1
	dprefi	V1
	llw	V1
	scw	V1
	lwup	V1
	swup	V1
	teqz	V1
	tnez	V1
	trap	V1
	lbup	V2
	sbup	V2
others	j*.xTO*	V1
	ret.x*	V1

Official
Release

6. 32-bit ISA Extensions

This chapter describes instructions of various 32-bit ISA extensions in the following sections:

- 
- 6.1 Performance Extension V1 Instructions on page 36
 - 6.2 Performance Extension V2 Instructions on page 37
 - 6.3 32-bit String Extension Instructions on page 38

6.1. Performance Extension V1 Instructions

The performance extension version 1 instructions can optimize high level language (C / C++) performance. The instructions in this extension are summarized in Table 50 and described in Section 8.3.

Table 50. Performance Extension V1 Instructions

Mnemonic	Instruction	Operation
ABS rt5, ra5	Absolute with Register	$rt5 = ra5 $
AVE rt5, ra5, rb5	Average two signed integers with rounding	$rt5 = (ra5 + rb5 + 1) \text{ (arith)} >> 1$ (page 220)
MAX rt5, ra5, rb5	Return the Larger	$rt5 = \text{signed-max} (ra5, rb5)$
MIN rt5, ra5, rb5	Return the Smaller	$rt5 = \text{signed-min} (ra5, rb5)$
BSET rt5, ra5, imm_5	Bit Set	$rt5 = ra5 (1 << \text{imm_5})$
BCLR rt5, ra5, imm_5	Bit Clear	$rt5 = (ra5 \&\& \sim(1 << \text{imm_5}))$
BTGL rt5, ra5, imm_5	Bit Toggle	$rt5 = ra5 ^ (1 << \text{imm_5})$
BTST rt5, ra5, imm_5	Bit Test	$rt5 = (ra5 \&\& (1 << \text{imm_5})) ? 1 : 0$
CLIPS rt5, ra5, imm_5	Clip Value Signed	$rt5 = (ra5 > 2\text{imm5}-1) ? 2\text{imm5}-1 : ((ra5 < -2\text{imm5}) ? -2\text{imm5} : ra5)$
CLIP rt5, ra5, imm_5	Clip Value	$rt5 = (ra5 > 2\text{imm5}-1) ? 2\text{imm5}-1 : ((ra5 < 0) ? 0 : ra5)$
CLZ rt5, ra5	Counting Leading Zeros in Word	$rt5 = \text{COUNT_ZERO_FROM_MSB}(ra5)$
CLO rt5, ra5	Counting Leading Ones in Word	$rt5 = \text{COUNT_ONE_FROM_MSB}(ra5)$

6.2. Performance Extension V2 Instructions

The performance extension version 2 instructions can optimize multimedia encoding/decoding processing related applications. The instructions in this extension are summarized in Table 51 and described in Section 8.4.

Table 51. Performance Extension V2 Instructions

Mnemonic	Instruction	Operation
BSE rt5, ra5, rb5	Bitstream Extraction	$rt5 = \text{Bitstream_Extract}(ra5, rb5)$ Please see page 231 for details.
BSP rt5, ra5, rb5	Bitstream Packing	$rt5 = \text{Bitstream_Packing}(ra5, rb5)$ Please see page 237 for details.
PBSAD rt5, ra5, rb5	Parallel Byte Sum of Absolute Difference	$a = \text{ABS}(ra5(7,0) - rb5(7,0));$ $b = \text{ABS}(ra5(15,8) - rb5(15,8));$ $c = \text{ABS}(ra5(23,16) - rb5(23,16));$ $d = \text{ABS}(ra5(31,24) - rb5(31,24));$ $rt5 = a + b + c + d;$
PBSADA rt5, ra5, rb5	Parallel Byte Sum of Absolute Difference Accumulate	$a = \text{ABS}(ra5(7,0) - rb5(7,0));$ $b = \text{ABS}(ra5(15,8) - rb5(15,8));$ $c = \text{ABS}(ra5(23,16) - rb5(23,16));$ $d = \text{ABS}(ra5(31,24) - rb5(31,24));$ $rt5 = rt5 + a + b + c + d;$

6.3. 32-bit String Extension

The string extension instructions can optimize text and string processing related algorithms.

These instructions are summarized in Table 52 and described in Section 8.5.

Table 52. String Extension Instructions

Mnemonic	Instruction	Operation
FFB rt5, ra5, rb5	Find First Byte	$rt5 = \text{Find_First_Byte}(ra5, rb5)$ Please see page 244 for details.
FFBI rt5, ra5, imm8	Find First Byte Immediate	$rt5 = \text{Find_First_Byte}(ra5, imm8)$ Please see page 246 for details.
FFMISM rt5, ra5, rb5	Find First Mis-Match	$rt5 = \text{Find_First_Mismatch}(ra5, rb5)$ Please see page 248 for details.
FLMISM rt5, ra5, rb5	Find Last Mis-Match	$rt5 = \text{Find_Last_Mismatch}(ra5, rb5)$ Please see page 250 for details.

7. Coprocessor Extension Instructions

For detailed coprocessor extension instruction information, please see *AndeStar™ Instruction Set Architecture Coprocessor Extension Manual*.



8. Detailed Instruction Description

This chapter lists all instructions in alphabetical order in each section with the following format:



Mnemonic (Brief Description)

Type: Type of this instruction in the Andes ISA

Format: Word encoding and format of this instruction

Syntax: Assembler syntax of this instruction

Purpose: Purpose of this instruction

Description: Detailed descriptions of this instruction

Operations: A more formal way to define the operations of this instruction.

Exceptions: The exceptions caused by this instruction

Privileged level: The processor operating mode(s) for this instruction to execute

Note: Additional information about this instruction

8.1. 32-bit Baseline V1 Instructions

8.1.1. ADD (Addition)

Type: 32-bit Baseline



Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		ADD 00000		

Syntax: ADD Rt, Ra, Rb

Purpose: Add the contents of two registers.

Description: This instruction adds the content of Ra and that of Rb, then writes the result to Rt.

Operations:

$$Rt = Ra + Rb;$$

Exceptions: None

Privilege level: All

Note:

8.1.2. ADDI (Add Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	0
0	ADDI 101000	Rt	Ra					imm15s

Syntax: ADDI Rt, Ra, imm15s

Purpose: Add the content of a register and a signed constant.

Description: This instruction adds the content of Ra and the sign-extended imm15s, then writes the result to Rt.

Operations:

$$Rt = Ra + SE(imm15s);$$

Exceptions: None

Privilege level: All

Note:

8.1.3. AND (Bit-wise Logical And)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		AND 00010		

Syntax: AND Rt, Ra, Rb

Purpose: Perform a bit-wise logical AND operation on the content of two registers.

Description: This instruction uses a bit-wise logical AND operation to combine the content of Ra with that of Rb, then writes the result to Rt.

Operations:

Rt = Ra & Rb;

Exceptions: None

Privilege level: All

Note:

8.1.4. ANDI (And Immediate)

Type: 32-bit Baseline

Format:

	31	30	25	24	20	19	15	14	0
	0	ANDI 101010	Rt	Ra					imm15u

Syntax: ANDI Rt, Ra, imm15u

Purpose: Perform a bit-wise logical AND operation on the content of a register with an unsigned constant.

Description: This instruction uses a bit-wise logical AND operation to combine the content of Ra and the zero-extended imm15u, then writes the result to Rt.

Operations:

Rt = Ra & ZE(imm15u);

Exceptions: None

Privilege level: All

Note:

8.1.5. BEQ (Branch on Equal)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	13	0
0	BR1 100110	Rt			Ra		BEQ 0		imm14s

Syntax: BEQ Rt, Ra, imm14s

Purpose: Perform conditional PC-relative branching based on the result of comparing the contents of two registers.

Description: If the content of Rt is equal to that of Ra, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm14s << 1) value. The branch range is $\pm 16K$ bytes.

Operations:

```

if (Rt == Ra) {
    PC = PC + SE(imm14s << 1);
}

```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.6. BEQZ (Branch on Equal Zero)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	16	15	0
0	BR2 100111	Rt			BEQZ 0010			imm16s

Syntax: BEQZ Rt, imm16s

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt is equal to zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm16s << 1) value. The branch range is $\pm 64K$ bytes.

Operations:

```
if (Rt == 0) {
    PC = PC + SE(imm16s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.7. BGEZ (Branch on Greater than or Equal to Zero)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	16	15	0
0	BR2 100111	Rt		BGEZ 0100		imm16s		

Syntax: BGEZ Rt, imm16s

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt, treated as a signed integer, is greater than or equal to zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm16s << 1) value. The branch range is $\pm 64K$ bytes.

Operations:

```
if (Rt >= 0) {
  PC = PC + SE(imm16s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.8. BGEZAL (Branch on Greater than or Equal to Zero and Link)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	16	15	0
0	BR2 100111	Rt		BGEZAL 1100		imm16s		

Syntax: BGEZAL Rt, imm16s

Purpose: Perform conditional PC-relative function call branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt, treated as a signed integer, is greater than or equal to zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm16s << 1) value. The branch range is $\pm 64K$ bytes. The instruction writes the program address of the next sequential instruction (PC+4) to R30 (Link Pointer register) unconditionally for function call return.

Operations:

```
bcond = Rt
R30 = PC + 4;
if (bcond >= 0) {
    PC = PC + SE(imm16s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.9. BGTZ (Branch on Greater than Zero)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	16	15	0
0	BR2 100111	Rt		BGTZ 0110		imm16s		

Syntax: BGTZ Rt, imm16s

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt, treated as a signed integer, is greater than zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm16s << 1) value. The branch range is $\pm 64K$ bytes.

Operations:

```
if (Rt > 0) {
  PC = PC + SE(imm16s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.10. BLEZ (Branch on Less than or Equal to Zero)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	16	15	0
0	BR2 100111	Rt		BLEZ 0111		imm16s		

Syntax: BLEZ Rt, imm16s

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt, treated as a signed integer, is less than or equal to zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm16s << 1) value. The branch range is $\pm 64K$ bytes.

Operations:

```
if (Rt <= 0) {
  PC = PC + SE(imm16s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.11. BLTZ (Branch on Less than Zero)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	16	15	0
0	BR2 100111	Rt		BLTZ 0101		imm16s		

Syntax: BLTZ Rt, imm16s

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt, treated as a signed integer, is less than zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm16s << 1) value. The branch range is $\pm 64K$ bytes.

Operations:

```
if (Rt < 0) {
  PC = PC + SE(imm16s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.12. BLTZAL (Branch on Less than Zero and Link)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	16	15	0
0	BR2 100111	Rt		BLTZAL 1101		imm16s		

Syntax: BLTZAL Rt, imm16s

Purpose: Perform conditional PC-relative function call branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt, treated as a signed integer, is less than zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm16s << 1) value. The branch range is $\pm 64K$ bytes. The instruction writes the program address of the next sequential instruction (PC+4) to R30 (Link Pointer register) unconditionally for function call return.

Operations:

```
bcond = Rt
R30 = PC + 4;
if (bcond < 0) {
    PC = PC + SE(imm16s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.13. BNE (Branch on Not Equal)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	13	0
0	BR1 100110	Rt		Ra		BNE 1		imm14s	

Syntax: BNE Rt, Ra, imm14s

Purpose: Perform conditional PC-relative branching based on the result of comparing the contents of two registers.

Description: If the content of Rt is not equal to that of Ra, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm14s << 1) value. The branch range is $\pm 16K$ bytes.

Operations:

```
if (Rt != Ra) {
    PC = PC + SE(imm14s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.14. BNEZ (Branch on Not Equal Zero)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	16	15	0
0	BR2 100111	Rt		BNEZ 0011		imm16s		

Syntax: BNEZ Rt, imm16s

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt is not equal to zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm16s << 1) value. The branch range is $\pm 64K$ bytes.

Operations:

```
if (Rt != 0) {
    PC = PC + SE(imm16s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.1.15. BREAK (Breakpoint)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	5	4	0
0	MISC 110010	00000			SWID		BREAK 01010	

Syntax: BREAK SWID

Purpose: Generate a Breakpoint exception.

Description: This instruction unconditionally generates a Breakpoint exception and transfers control to the Breakpoint exception handler. Software uses the 15-bit SWID of the instruction as a parameter to distinguish different breakpoint features and usages.

Operations:

Generate_Exception(Breakpoint);

Exceptions: Breakpoint

Privilege level: All

Note:

8.1.16. CCTL (Cache Control)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	11	10	9	5	4	0
0	MISC 110010	Rt/Rb	Ra				0000		level	SubType	CCTL 00001		

Syntax: CCTL Ra, SubType

- | | |
|-------------------------|--|
| CCTL Ra, SubType, level | (VA writeback or invalidate operation) |
| CCTL Rt, Ra, SubType | (Cache read operation) |
| CCTL Rb, Ra, SubType | (Cache write operation) |
| CCTL L1D_INVALALL | (Cache invalidate all operation) |

Purpose: Perform various operations on processor caches. It maintains cache coherence for shared memory.

Description: This instruction performs cache control operations based on the SubType field. The definitions and encodings of the SubType field are listed in Table 53 and Table 54. Some CCTL operations can be used with user privilege to assist in coherence and synchronization management. Some CCTL cache read operations have an additional destination register Rt and some CCTL write operations have an additional source register Rb.

Although defined, certain SubType encodings of the cache control instruction are optional to implement. The execution of an unimplemented optional cache control operation causes a Reserved Instruction exception. So does the execution of an undefined SubType encoding of this instruction.

For the “level” field, please see “multi-level cache management operation” at page 62.

Table 53. CCTL SubType Encoding

SubType		bit 4-3			
		0	1	2	3
bit 2-0		00	01	10	11
		L1D_IX	L1D_VA	L1I_IX	L1I_VA
0	000	L1D_IX_INVAL	L1D_VA_INVAL	L1I_IX_INVAL	L1I_VA_INVAL
1	001	L1D_IX_WB	L1D_VA_WB	-	-
2	010	L1D_IX_WBINVAL	L1D_VA_WBINVAL	-	-
3	011	L1D_IX_RTAG	L1D_VA_FILLCK	L1I_IX_RTAG	L1I_VA_FILLCK
4	100	L1D_IX_RWD	L1D_VA_ULCK	L1I_IX_RWD	L1I_VA_ULCK
5	101	L1D_IX_WTAG	-	L1I_IX_WTAG	-
6	110	L1D_IX_WWD	-	L1I_IX_WWD	-
7	111	L1D_INVALALL	-	-	-

Table 54. CCTL SubType Definitions

Mnemonics	Operation (Category)	Ra Type	Rt?/Rb?	User Privilege	Compliance
L1D_IX_INVAL	Invalidate L1D cache (A)	Index	-/-	-	Required
L1D_VA_INVAL	Invalidate L1D cache (A)	VA	-/-	Yes	Required
L1D_IX_WB	Write Back L1D cache (B)	Index	-/-	-	Required
L1D_VA_WB	Write Back L1D cache (B)	VA	-/-	Yes	Required
L1D_IX_WBINVAL	Write Back & Invalidate L1D cache	Index	-/-	-	Optional

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

Mnemonics	Operation (Category)	Ra Type	Rt?/Rb?	User Privilege	Compliance
	(C)				
L1D_VA_WBINVAL	Write Back & Invalidate L1D cache (C)	VA	-/-	Yes	Optional
L1D_VA_FILLCK	Fill and Lock L1D cache (D)	VA	-/-	-	Optional
L1D_VA_ULCK	unlock L1D cache (E)	VA	-/-	-	Optional
L1D_IX_RTAG	Read tag L1D cache (F)	Index	Yes/-	-	Optional
L1D_IX_RWD	Read word data L1D cache (G)	Index/w	Yes/-		Optional
L1D_IX_WTAG	Write tag L1D cache* (H)	Index	-/Yes	-	Optional
L1D_IX_WWD	Write word data L1D cache* (I)	Index/w	-/Yes	-	Optional
L1D_INVALALL	Invalidate All L1D cache (J)	N/A	-/-	-	Optional

Mnemonics	Operation (Category)	Ra Type	Rt?/Rb?	User Privilege	Compliance
L1I_VA_FILLCK	Fill and Lock L1I cache (D)	VA	-/-	-	Optional
L1I_VA_ULCK	unlock L1I cache (E)	VA	-/-	-	Optional
L1I_IX_INVAL	Invalidate L1I cache (A)	Index	-/-	-	Required
L1I_VA_INVAL	Invalidate L1I cache (A)	VA	-/-	Yes	Required
L1I_IX_RTAG	Read tag L1I cache (F)	Index	Yes/-	-	Optional
L1I_IX_RWD	Read word data L1I cache (G)	Index/w	Yes/-		Optional
L1I_IX_WTAG	Write tag L1I cache (H)	Index	-/Yes	-	Optional
L1I_IX_WWD	Write word data L1I cache (I)	Index/w	-/Yes	-	Optional

***Note:** An implementation may omit the “Write word data L1D cache” operation since it can achieve the same effect using store instructions after the CCTL “Write tag L1D cache” operation and a “Data Serialization Barrier” instruction.

The content of the register Ra shows the correct cache location for the cache operation. The table below lists its usages:

Ra Type	Usage
Index	Use the content of Ra directly as a (Index, Way) pair to access the cache location without going through any translation mechanism. The real format for the (Index, Way) pair in the content of Ra is implementation-dependent. However, it should have a general form shown in Figure 1. Software can discover the real format by consulting the ICM CFG (cr1) and DCM_CFG (cr2) Configuration Registers in the Andes processor core.
Index/w	Use the content of Ra directly as a (Index, Way, Word) triple to access the cache location without going through any translation. The “Word” means a 4-byte word in the cache line pointed to by the (Index, Way) pair. The remaining description is similar to the “Index” type above.
VA	Use the content of Ra as a virtual address to access the cache. The address goes through the same address translation mechanism in the processor pipeline as the address of a load/store instruction for D cache or an instruction fetch for I cache. Perform the specified operation if the address hits in the corresponding cache. If the cache is missed, no specific operation is performed.

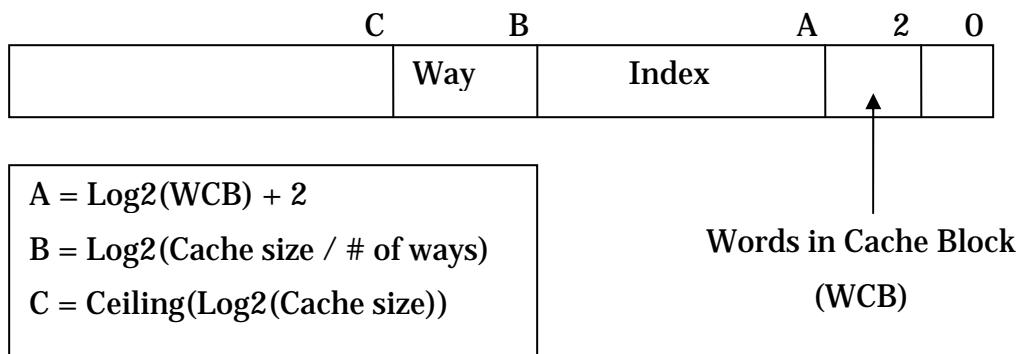


Figure 1. Index Type Format for Ra of CCTL Instruction

The operations of the cache control instruction are grouped and described in the following

categories:

(A). Invalidate cache block

This operation unlocks the cache line and sets its state to invalid. It is implementation-dependent on how this instruction affects other states of the cache line such as “way selection.”

(B). Write back cache block

If the cache line state is valid and dirty in a write-back cache, this operation writes the cache line back to memory. It does not affect the lock state of the cache line.

(C). Write back & invalidate cache block

If the cache line state is valid and dirty in a write-back cache, this operation writes the cache line back to memory, and then performs a cache block invalidating operation described in (A).

(D). Fill and lock cache block

If the desired cache line is not present in the cache, this operation fills the cache line into the cache and then sets the lock state of the cache line; if the desired cache line is present in the cache, this instruction only sets the lock state. A locked cache line will not be replaced when a cache miss/fill event happens and can only be unlocked using a cache invalidate or unlock operation.

Since this cache line lock operation is defined under the implementation assumption of a multi-way set-associative cache, the support of this operation is implementation-dependent. If this instruction is supported for a multi-way set-associative cache, it can only lock up to “way-1” cache lines in a cache set; if software wants to get a predictable behavior, be sure not to lock more than “way-1” cache lines in a cache set. If software locks all “way” cache lines in a cache set and ICALCK/DCALCK of Cache Control system register is 0, it is IMPLEMENTATION-DEPENDENT on when an exception will be generated.

(E). Unlock cache block

This operation clears the lock state of the cache if the desired cache line is present in the cache.

(F). Read tag from cache

This operation reads the contents of a cache line tag into a general register Rt. The tag format is implementation-dependent. A reference format is illustrated as follows.

31	23	22	21	2	1	0
Ignored	dirty		PA(31,12)		valid	lock

The content of Ra specifies the index and way of the target cache line.

(G). Read data word from cache

This operation reads a 4-bytes word from a cache line into a general register Rt. The endian format of this operation should depend on the value of PSW.BE. The content of Ra specifies the index, way, and word of the target cache line.

(H). Write tag to cache

This operation writes the cache line tag from a general register Rb. The tag format is implementation-dependent. A reference format is illustrated as follows.

31	23	22	21	2	1	0
Ignored	dirty		PA(31,12)		valid	lock

The content of Ra specifies the index and way of the target cache line.

(I). Write word data to cache

This operation writes a 4-byte word in a general register Rb into a target cache line. The endian format of this operation should depend on the value of PSW.BE. The content of Ra specifies the index, way, and word of the target cache line.

(J). Invalidate all cache block

This operation unlocks all of the cache lines and sets their states to invalid. It is implementation-dependent on how this instruction affects other states of the cache lines such as “way selection.”

- **Multi-level cache management operation**

For a system with multiple levels of caches, it would be more convenient for software to

manage a cache block if it can control whether a CCTL write-back or invalidate operation is applied to just the first level cache or to other higher levels of caches as well. This benefits software to use just one CCTL instruction to affect all levels of caches, saving the trouble of using separate instructions to manage each level of cache individually.

To enable this, all of CCTL instructions with VA writeback or VA invalidate related operations have two flavors to indicate if the operation is applied to all levels of caches or only one level. The two flavors are as follows:

All level operation: “CCTL Ra, SubType, alevel”

One level operation: “CCTL Ra, SubType, 1level”

The following SubTypes have these two flavors.

L1D_VA_INVAL
L1D_VA_WB
L1D_VA_WBINVAL
L1I_VA_INVAL

For an AndesCore™ with MSC_CFG.L2C == 1, the above 4 CCTL “all level” flavors of instructions will pass the CCTL operation to the higher levels of caches. For an AndesCore with MSC_CFG.L2C == 0, the above 4 CCTL “all level” flavors of instructions may behave either like the corresponding “1 level” flavor of instructions or generate a Reserved Instruction exception. The choice is implementation-dependent.

For all other CCTL SubTypes, the “all level” encoded instructions behave as “1 level” encoded instructions when MSC_CFG.L2C == 1. When MSC_CFG.L2C == 0, the “all level” encoded instructions may behave either as “1 level” encoded instructions or generate a Reserved Instruction exception. This choice is implementation-dependent.

For software to make sure the completeness of the “all level” operations, use a “MSYNC all” instruction after the “all level” operations. It guarantees that any instructions after the “MSYNC all” will see the effects completed by the “all level” operations.

Programming Constraints:

1. CCTL Ra, L1D_VA_WB, alevel:

This instruction guarantees to write back a dirty cache line of L1 cache to memory.

2. CCTL Ra, L1D_VA_INVAL, alevel:

This instruction invalidates cache lines of different sizes in different levels of a cache hierarchy. To use this instruction, first make sure writing back all data in a cache line unit aligned to the largest cache line size in the cache hierarchy. Otherwise, be sure that the portion of the cache unit covering the largest cache line size, when not written back, contains no dirty data.

Operations:

```

If (SubType is not supported)
  Exception(Reserved Instruction);
If (RaType(SubType) == VA) {
  VA = (Ra);
  PA = AddressTranslation(VA);
  VA_cache_control (SubType, PA, Level );
} else {
  {Idx, way, word} = (Ra)
  If (Op(SubType) == CacheTagRead) {
    Rt = Idx_cache_tag_read(SubType, Idx, way);
  } else if (Op(SubType) == CacheDataRead) {
    Rt = Idx_cache_data_read(SubType, Idx, way, word);
  } else if (Op(SubType) == CacheWrite) {
    Idx_cache_write(SubType, Rb, Idx, way, word);
  } else {
    Idx_cache_control (SubType, Idx, way);
  }
}
  
```

Exceptions:

TLB fill exception, Non-Leaf PTE not present exception, Leaf PTE not present exception, Read protection violation exception, TLB VLPT miss exception, Imprecise bus error exception, Reserved instruction exception, Privileged Instruction.

When executed under the *user operating mode*, the following four CCTL subtypes will generate exceptions if the page access permission is not setup correctly.

CCTL Subtypes	Required Permission under <i>user mode</i>	Generated exception
L1D_VA_INVAL	Read	Data write protection violation
L1D_VA_WB	Write	Data write protection violation
L1D_VA_WBINVAL	Write	Data write protection violation
L1I_VA_INVAL	Read	Data read protection violation

When executed under the *superuser operating mode*, all the CCTL SubTypes will not generate exceptions due to the page read/write/execute permission.

As for the checking and generation of page modified exception, access bit exception and non-executable page exception, the following table lists the expected behaviors:

CCTL SubTypes	Page modified exception under all mode	Access bit exception under all mode	Non-executable page exception under all mode
L1D_*	Ignore	Ignore	N/A
L1I_*	Ignore	Ignore	Ignore

Privilege level: Depends on operation types.

Note:

- (1). If the specified cache is implemented, a CCTL instruction should operate on the specified cache regardless of the cache enable/disable control bits in the Cache Control register.
- (2). All non-instruction-fetch-related exceptions generated by a CCTL instruction should have the INST field of the ITYPE register set to 0.
- (3). For normal run time operations (thus excluding CCTL write tag operation), the state transition of the lock flag of a cache line affected by a CCTL instruction is illustrated in the following diagram:

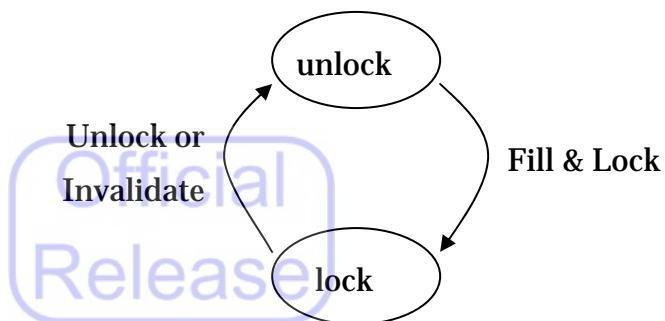


Figure 2. State Diagram of the Lock Flag of a Cache Line Controlled by CCTL

8.1.17. CMOVN (Conditional Move on Not Zero)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		CMOVN 11011		

Syntax: CMOVN Rt, Ra, Rb

Purpose: Move the content of a register based on the result of comparing the content of a register with zero.

Description: If the content of Rb is not equal to zero, this instruction moves the content of Ra to Rt.

Operations:

```
if (Rb != 0) {
    Rt = Ra;
}
```

Exceptions: None

Privilege level: All

Note:

8.1.18. CMOVZ (Conditional Move on Zero)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		CMOVZ 11010		

Syntax: CMOVZ Rt, Ra, Rb

Purpose: Move the content of a register based on the result of comparing the content of a register with zero.

Description: If the content of Rb is equal to zero, this instruction moves the content of Ra to Rt.

Operations:

```
if (Rb == 0) {
    Rt = Ra;
}
```

Exceptions: None

Privilege level: All

Note:

8.1.19. DPREF/DPREFI (Data Prefetch)

Type: 32-bit Baseline

Format:

DPREF													
31	30	25	24	23	20	19	15	14	10	9	8	7	0
0	MEM 011100	0	SubType		Ra		Rb		shift2		DPREF 00010011		

DPREFI

31	30	25	24	23	20	19	15	14	0
0	DPREFI 010011	d	SubType		Ra				imm15s

Syntax: DPREF SubType, [Ra + (Rb << shift2)]

DPREFI.d SubType, [Ra + (imm15s.000)]

DPREFI.w SubType, [Ra + (imm15s.00)]

Purpose: Hint to move data from memory into data caches before the actual load or store operations. It reduces memory access latency.

Description:

An implementation can use the effective byte address calculated from this instruction (DPREF/DPREFI) to improve performance by moving the memory line containing the address from memory into data caches in advance. However, an implementation may also decide to do nothing at any stage by treating this instruction as a NOP. As a hint, this instruction does not generate any observable results or any exceptions which alter the behavior of a program (e.g. an address exception which leads OS to abort the program.) Note that the “cache line write” hint subtype may be an exception from the above statement. Please see the detailed subtype descriptions followed.

To be effective in increasing performance, the data prefetch instruction should be implemented

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

non-blockingly to overlap with other instructions. The actual number of cache lines and memory hierarchies affected are implementation-dependent.

The effective byte address for DPREF is $Ra + (Rb << \text{shift2})$ where shift2 is a 2-bit shift amount for Rb.

The effective byte address for DPREFI is defined in two flavors: one for word and the other for double word. The double word flavor has a byte address of $Ra + \text{SignExtend}(\text{imm15s}.000)$ where imm15s represents an 8-byte-double-word offset. This coarser-grain immediate offset does not sacrifice the data prefetching performance since most of the cache block moves between memory and data cache is greater than 8 bytes. The word flavor has a byte address of $Ra + \text{SignExtend}(\text{imm15s}.00)$ where imm15s represents a 4-byte-word offset. This finer-grain immediate offset may give hardware a chance to optimize the data prefetch instruction by sub-blocking the data cache and filling the critical word of a prefetched cache line first. The “d” bit (bit 24) in the instruction encoding distinguishes these two flavors as follows:

“d” bit	Meaning
0	DPREFI.w
1	DPREFI.d

The SubType field of this instruction tells hardware the intended use of the prefetched data so that the hardware implementation may use different prefetch schemes to optimize the performance. The definitions of the SubType field are listed in the following table.

Sub Type	Mnemonics	Hint	Description
0	SRD	Single Read	The data will be read for only once. (i.e. no temporal locality)
1	MRD	Multiple Read	The data will be read multiple times. (i.e. has temporal locality)
2	SWR	Single Write	The data will be written for only once. (i.e. no temporal locality)

Sub Type	Mnemonics	Hint	Description
3	MWR	Multiple Write	The data will be written multiple times. (i.e. has temporal locality)
4	PTE	PTE Preload	Preload page translation information into TLB.
5	CLWR	Cache Line Write	The whole cache line will be written by store instructions. If the cache line is in the data cache and permitted for writing, the processor will do nothing; if the cache line is in the data cache but not permitted for writing, the processor may request the write permission for the cache line; if the cache line is not in the data cache, it may be allocated in the data cache with all zero data and write permission. This instruction may alter memory states of the cache line if the program fails to write the whole cache line with new data later. The final memory states will be UNPREDICTABLE in this case depending on cache hit or miss when this instruction is issued. This memory state altering characteristic is implementation-dependent.
6-15	-	Imp-dep	Implementation dependent

The data prefetch instruction does not prefetch memory locations with uncached attribute.

Operations:

```

VA = Ra+(Rb << shi ft2); // DPREF
VA = Ra+(i mm15s. 000);    // DPREFI . d
VA = Ra+(i mm15s. 00);     // DPREFI . w
PA_found = MMU_Search(VA);
If (PA_found) {
  (PA, attribute) = MMU_Translate(VA);
  Data_Prefetch(PA, attribute, hint);
} else if (hint == "TLB preload")
  TLB_Preload(VA);
  
```

Exceptions:

None, (Implementation-dependent: imprecise Cache error or imprecise Bus error)

Privilege level: All**Note:**

- (1) The effective address of this data prefetch instruction does not need to be aligned to any data type boundary (e.g. word-aligned) since it indicates a cache line rather than a specific data type.
- (2) This data prefetch instruction does not generate any exception. If a hardware page table walker is implemented, this instruction will not generate any hardware page table walk memory access as well.
- (3) This data prefetch instruction may be used ahead of the real memory access instruction and be generated unguardedly inside a loop, so it is very likely to generate an address which falls outside of the legal data range. Then, generating hardware page table walk memory access on a data prefetch instruction with such data address will cause unnecessary execution penalty which deters the usability of this instruction.

8.1.20. DSB (Data Serialization Barrier)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	5	4	0
0	MISC 110010	00000		0000000000000000		DSB 01000		

Syntax: DSB

Purpose: Serialize a read-after-write data dependency for certain architecture/hardware state updates which affect data processing related operations. It guarantees a modified architecture or hardware state can be seen by following dependent data operations.

Description:

This instruction blocks the execution of any subsequent instructions until all previously modified architecture/hardware states can be observed by subsequent dependent data operations in a pipelined processor. It is not needed for general register states since general register states are serialized by hardware.

For user programs, an endian mode change operation (SETEND) requires a DSB instruction to make sure the endian change is seen by following load/store instructions. Other than this case, all uses of DSB are in privileged programs for managing system states.

Operations:

`Ser i al i ze_Data_States()`

Exceptions: None

Privilege level: All

Note:

- (1) The following table lists some of the state writers and readers that need DSB between them for the readers to get the expected new value/behaviors.

State	Writer	Reader	Value
System register	MTSR	MFSR	New SR value
PSW.BE	SETEND	load/store	New endian state
PSW.DT	MTSR	load/store	New translation behavior
PSW.GIE/ SIM/ H5IM-HOIM	MTSR	following instructions	Interrupt behavior
PSW.GIE	SETGIE	following instructions	New global interrupt enable state
PSW.INTL	MTSR	following instructions	Interruption stack behavior
D\$ line valid	CCTL L1D (sub=inval, WB&inval, WR tag)	load/store	D\$ hit/miss
		CCTL L1D (sub=RD tag)	Tag valid
DTLB	TLBOP RWR/ RWLK/ TWR	load/store	Data PA
	TLBOP FLUA/INV	load/store	TLB miss
	TLBOP TWR	TLBOP TRD	TLB entry data
CACHE_CTL (DC_ENA)	MTSR	load/store	D\$ enable/disable behavior
DLMB (EN)	MTSR	load/store	Data local memory access behavior
L1_PPTB	MTSR	load/store	HPTWK behavior
TLB_ACC_XXX	TLBOP TRD	MFSR	TLB read out

State	Writer	Reader	Value
DMA channel selection (DMA_CHNSEL)	MTSR DMA_CHNSEL	MTSR/MFSR DMA channel registers	New DMA channel number
PSW.INTL IPC/P_IPC IPSW/P_IPSW P_EVA P_P0/P_P1 P_ITYPE OIPC	MTSR	IRET	New SR value used by an IRET instruction

- (2) PSW.GIE controls if an asynchronous interrupt can be inserted between two instructions. That is, when an instruction observes an updated PSW.GIE value, the value affects if the instruction and its next instruction can be interrupted or not.

Based on this definition, the DSB instruction after a PSW.GIE update operation does not guarantee itself to observe the updated PSW.GIE value and may observe the old PSW.GIE value. Thus, if the PSW.GIE is updated to 0 to disable interrupt, an interrupt can still be inserted between DSB and the immediate following instruction (inst1). The DSB instruction in fact only guarantees that its following instructions (inst1, inst2, etc.) can observe the updated PSW.GIE value to be 0, thereby preventing any interrupt inserted between the immediate following instruction (inst1) and its next instruction (inst2), and onward.

```
setgi e. d
dsb
i nst1
i nst2
```

In addition to PSW.GIE, the same definition also works for the following interrupt masking fields:

INT_MASK.SIM
INT_MASK.H5IM-HOIM

8.1.21. IRET (Interruption Return)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	5	4	0
0	MISC 110010	00000		0000000000000000		IRET 00100		

Syntax: IRET

Purpose: Return from interruption to the instruction and states when the processor was being interrupted.

Description: This instruction conditionally updates (pops) the system register and program counter stack such that the processor behavior after the instruction returns to a state when the processor was being interrupted. To be more specific, the following states are updated based on interruption level (INTL) conditions:

For System Privilege Architecture V2:

- For INTL=0, 1

 $PC \leftarrow IPC$
 $PSW \leftarrow IPSW$


- For INTL=2

 $PC \leftarrow IPC$
 $IPC \leftarrow P_IPC$
 $PSW \leftarrow IPSW$
 $IPSW \leftarrow P_IPSW$
 $ITYPE \leftarrow P_ITYPE$
 $EVA \leftarrow P_EVA$
 $P0 \leftarrow P_P0$
 $P1 \leftarrow P_P1$

- For INTL=3

 $PC \leftarrow OIPC$
 $PSW.INTL \leftarrow 2$

For System Privilege Architecture V3:

- For INTL=0, 1, 2

 $PC \leftarrow IPC \leftarrow P_IPC$
 $PSW \leftarrow IPSW \leftarrow P_IPSW$
 $ITYPE \leftarrow P_ITYPE$
 $EVA \leftarrow P_EVA$
 $P0 \leftarrow P_P0$
 $P1 \leftarrow P_P1$

- For INTL=3

 $PC \leftarrow OIPC$
 $PSW.INTL \leftarrow 2$

Since these processor state updates affect processor fetching and data processing behaviors, instruction and data serialization operations are also included in the IRET instruction in addition to the register stack update. Therefore, any instruction or instruction fetching after the IRET can see the newly updated processor states caused by the IRET instruction.

This instruction must read the latest state of the following system registers before performing its operations. So any update of these system registers does not need to add any serialization instruction (ISB or DSB) between the update and this instruction for this IRET to see the latest update. This is an important implementation requirement.

PSW.INTL	IPC	IPSW	P_IPC	P_IPSW
P_ITYPE	P_EVA	P_P0	P_P1	OIPC

Operations:

```

If (((PSW.INTL==0) or (PSW.INTL==1)) and (SPA==V2)) {
    PC = IPC;
    PSW = IPSW;
} else if ((PSW.INTL==0) or (PSW.INTL==1) or (PSW.INTL==2)) {
    PC = IPC;
    PSW = IPSW;
    IPC = P_IPC;
    IPSW = P_IPSW;
    ITYPE = P_ITYPE;
    EVA = P_EVA;
    P0 = P_P0;
    P1 = P_P1;
} else {           // maximum interrupton level
    PC = OIPC;
    PSW.INTL = 2;
}
Serialize_Data_States()
Serialize_Instruction_States()

```

Exceptions: Privileged Instruction

Privilege level: Superuser and above

Note:

8.1.22. ISB (Instruction Serialization Barrier)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	5	4	0
0	MISC 110010	00000		0000000000000000		ISB 01001		

Syntax: ISB

Purpose: Serialize a read-after-write data dependency for certain architecture/hardware state updates which affect instruction fetching related operations. It also serializes all architecture/hardware state updates that are serialized by a DSB instruction. It guarantees a modified architecture or hardware state can be seen by following dependent instruction fetching operations and data processing operations.

Description:

This instruction blocks the execution of any subsequent instructions until all previously modified architecture/hardware states can be observed by subsequent dependent (1) instruction fetching operations and (2) data processing operations that are serialized by a DSB instruction in a pipelined processor. It is not needed when general register states are serialized by hardware.

The interruption return instruction (IRET) includes implicit ISB operation (including data serialization). Thus, you can use either the IRET or ISB instruction to achieve the expected instruction and data serialization behavior.

Operations:

`Ser i al i ze_Data_States()`
`Ser i al i ze_Instr ucti on_States()`

Exceptions: None

Privilege level: All

Note:

The following table lists some of the state writers and readers that need ISB between them for the readers to get the expected new value/behaviors.

State	Writer	Reader	Value
PSW.IT	MTSR	Instruction fetch	Instruction translation behavior
ITLB	TLBOP RWR/ RWLK/ TWR	Instruction fetch	Instruction PA
	TLBOP FLUA/ INV	Instruction fetch	TLB miss
IS line valid/data	ISYNC	Instruction fetch	New instruction data
IS line valid	CCTL L1I (sub=inval)	Instruction fetch	I\$ miss
IS line valid/tag	CCTL L1I (sub=WR tag)	CCTL L1I (sub=Rd tag)	I\$ hit
IS line data	CCTL L1I (sub=WR word)	CCTL L1I (sub=RD word)	Instruction data
Memory	MSYNC	Instruction fetch	New instruction data
CACHE_CTL (IC_ENA)	MTSR	Instruction fetch	I\$ enable/disable behavior
ILMB (EN)	MTSR	Instruction fetch	Instruction local memory access behavior
L1_PPTB	MTSR	Instruction fetch	HPTWK behavior
IVB	MTSR	Instruction fetch	Interruption behavior

8.1.23. ISYNC (Instruction Data Coherence Synchronization)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	5	4	0
0	MISC 110010	Ra			0000000000000000		ISYNC 01101	

Syntax: ISYNC Ra

Purpose: Make sure the instruction fetch event performed after an instruction serialization barrier instruction (ISB or IRET) can observe the instruction data updated in the cache line address specified by Ra.

Description:

This instruction guarantees that an instruction fetch event after an instruction serialization barrier instruction can properly observe instruction data updated before this instruction. It alleviates the Andes architecture implementation by implementing coherence logic between instruction and data caches to handle self-modifying code.

After generating new instruction data, software is required to use this instruction and the correct instruction sequence to synchronize the updated data between I and D caches. This is for correctly executing the newly modified instruction fetched after an instruction serialization barrier instruction. If software doesn't use this instruction and the correct instruction sequence in the above situation, the Andes implementation may fetch and execute the old instruction data afterward .

After a previous store instruction updating instruction data, this instruction initiates a data cache write-back operation and an instruction cache invalidating operation. Particularly, it guarantees that the data write back operation has an address space access order (relative to this processor) BEFORE any subsequent instruction fetch operation (including speculative fetches) to the same cache-line address. The subsequent instruction serialization barrier instruction ensures that all instructions following it are re-fetched into the processor execution unit.

The content of Ra specifies the virtual address of the data cache line which contains the instruction data. It goes through the same address mapping mechanism as those associated with load and store instructions. Thus, address translation and protection exceptions may occur for this instruction.

Operation:


VA = (Ra)

PA = AddressTranslation(VA)

Synchronizing Caches(PA)

Exceptions: TLB fill exception, Non-Leaf PTE not present exception, Leaf PTE not present exception, Read protection violation exception, TLB VLPT miss exception, Imprecise bus error exception, Machine check exception

Privilege level: All

Notes:

- (1) If this instruction affects a locked cache line in the instruction cache, the affected cache line will be unlocked.
- (2) The processor behavior is UNPREDICTABLE if the VA of this instruction points to any cache line that contains instructions between this instruction and the next instruction serialization barrier instruction.
- (3) The effective address of this instruction does not need to be aligned to any data type boundary (e.g. word-aligned) since it indicates a cache line rather than a specific data type. Thus, no Data Alignment Check exception is generated for this instruction.
- (4) Except for AndesCore N12 hardcore, the correct instruction sequence for writing or updating code data that will be executed afterwards is as follows:

UPD_LOOP:

```

// preparing new code data in Ra
.....
// preparing new code address in Rb, Rc
.....
// writing new code data
  
```

```

store Ra, [Rb, Rc]
// Looping control
.....
bne UPD_LOOP

I SYNC_LOOP:
// preparing new code address in Rd
isync Rd
// Looping control
bne I SYNC_LOOP
isb
// execution of new code data can be started from here
.....

```

For AndeStar N12 hardcore, the correct instruction sequence is as follows:

```

UPD_LOOP:
// preparing new code data in Ra
.....
// preparing new code address in Rb, Rc
.....
// writing new code data
store Ra, [Rb, Rc]
// Looping control
.....
bne Rb, Re, UPD_LOOP

WB_LOOP:
// preparing new code address in Rd
isync Rd (or cctl Rd, L1D_VA_WB)
// Looping control
bne Rd, Re, WB_LOOP
msync
isb

ICACHE_INV_LOOP:
// preparing new code address in Rf
cctl Rf, L1I_VA_INVAL
// Looping control
bne Rf, Re, ICACHE_INV_LOOP
isb
// execution of new code data can be started from here
.....

```

8.1.24. J (Jump)

Type: 32-bit Baseline

Format:

31	30	25	24	23	0
0	JI 100100	J 0		imm24s	

Syntax: J imm24s

Purpose: Unconditionally branch to a region relative to current instruction.

Description: This instruction branches unconditionally to a PC-relative region defined by sign-extended (imm24s <<1) where the final branch address is half-word aligned. The branch range is $\pm 16M$ bytes.

Operations:

```
PC = PC + SE(imm24s << 1);
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult Andes Programming Guide to get the correct meaning of the displayed syntax.

8.1.25. JAL (Jump and Link)

Type: 32-bit Baseline

Format:

31	30	25	24	23	0
0	JI 100100	JAL 1			imm24s

Syntax: JAL imm24s

Purpose: Make an unconditional function call relative to current instruction.

Description: This instruction branches unconditionally to a PC-relative region defined by sign-extended (imm24s <<1) where the final branch address is half-word aligned. The branch range is $\pm 16M$ bytes. The program address of the next sequential instruction (PC+4) is written to R30 (Link Pointer register) for function call return.

Operations:

```
R30 = PC + 4;  
PC = PC + SE(imm24s << 1);
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult Andes Programming Guide to get the correct meaning of the displayed syntax.

8.1.26. JR (Jump Register)

Type: 32-bit Baseline

Format:

31	30	25	24	15	14	10	9	8	7	6	5	4	0
0	JREG 100101	0000000000		Rb		DT/IT 00		00		JR hint 0		JR 00000	

Syntax: JR Rb

Purpose: Unconditionally branch to an instruction address stored in a register.

Description: This instruction branches unconditionally to an instruction address stored in Rb. The JR hint field distinguishes this instruction from the RET instruction which has the same architecture behavior but differs in software usages.

Operations:

PC = Rb;

Exceptions: None

Privilege level: All

Note:

8.1.27. JR.xTOFF (Jump Register and Translation OFF)

Type: 32-bit Baseline (with MMU configuration)

Format:

JR.ITOFF													
31	30	25	24	15	14	10	9	8	7	6	5	4	0
0	JREG 100101	0000000000		Rb		DT/IT 01		00		JR hint 0		JR 00000	

JR.TOFF

31	30	25	24	15	14	10	9	8	7	6	5	4	0
0	JREG 100101	0000000000		Rb		DT/IT 11		00		JR hint 0		JR 00000	

Syntax: JR.[T | IT]OFF Rb

Purpose: Unconditionally branch to an instruction address stored in a register and turn off address translation for the target instruction.

Description: This instruction branches unconditionally to an instruction address stored in Rb. It also clears the IT (and DT if included) field of the Processor Status Word (PSW) system register to turn off the instruction (and data if included) address translation process in the memory management unit. This instruction guarantees that fetching of the target instruction sees PSW.IT as 0 (and PSW.DT as 0 if included) and thus does not go through the address translation process. The JR hint field distinguishes this instruction from the RET.xTOFF instruction which has the same architecture behavior but differs in software usages.

Operations:

```
PC = Rb;  
PSW.IT = 0;  
if (INST(9) == 1) {  
    PSW.DT = 0;  
}
```

Official
Release

Exceptions: Privileged Instruction, Reserved Instruction (for non-MMU configuration)

Privilege level: Superuser and above

Note: This instruction is used in an interruption handler or privileged code in a translated address space to return to a non-translated address space. Please see the JRAL.xTON instruction for the reverse process of coming from a non-translated address space to a translated address space.

8.1.28. JRAL (Jump Register and Link)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	8	7	5	4	0
0	JREG 100101	Rt		00000		Rb			DT/IT 00	000		JRAL 00001		

Syntax: JRAL Rb (implied Rt == R30, Link Pointer register)

JRAL Rt, Rb

Purpose: Make an unconditional function call to an instruction address stored in a register.

Description: This instruction branches unconditionally to an instruction address stored in Rb. It writes the program address of the next sequential instruction (PC+4) to Rt for function call return.

Operations:

```
j addr = Rb;
Rt = PC + 4;
PC = j addr;
```

Exceptions: None

Privilege level: All

Note:

8.1.29. JRAL.xTON (Jump Register and Link and Translation ON)

Type: 32-bit Baseline (with MMU configuration)

Format:

JRAL.ITON														
31	30	25	24	20	19	15	14	10	9	8	7	5	4	0
0	JREG 100101		Rt		00000		Rb		DT/IT 01		000		JRAL 00001	

JRAL.TON

31	30	25	24	20	19	15	14	10	9	8	7	5	4	0
0	JREG 000101		Rt		00000		Rb		DT/IT 11		000		JRAL 00001	

Syntax: JRAL.[T | IT]ON Rb (implied Rt == R30, Link Pointer register)

 JRAL.[T | IT]ON Rt, Rb

Purpose: Make an unconditional function call to an instruction address stored in a register and turn on address translation for the target instruction.

Description: This instruction branches unconditionally to an instruction address stored in Rb. It also sets the IT (and DT if included) fields of the Processor Status Word (PSW) system register to turn on the instruction (and data if included) address translation process in the memory management unit. It writes the program address of the next sequential instruction (PC+4) to Rt for function call return. This instruction guarantees that fetching of the target instruction sees PSW.IT as 1 (and PSW.DT as 1 if included) and goes through the address translation process.

Operations:

```
j addr = Rb  
Rt = PC + 4;  
PC = j addr;  
PSW.IT = 1;  
if (INST(9) == 1) {  
    PSW.DT = 1;  
}
```



Exceptions: Privileged Instruction, Reserved Instruction (for non-MMU configuration)

Privilege level: Superuser and above

Note: This instruction is used in an interruption handler or privileged code in a non-translated address space to jump to a function in a translated address space. Please see the JR.xTOFF/RET.xTOFF instruction for the reverse process of returning from a translated address space to a non-translated address space.

8.1.30. LB (Load Byte)

Type: 32-bit Baseline

Format:

LB												0
31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LB 00000000		

LB.bi

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LB.bi 00000100		

Syntax: LB Rt, [Ra + (Rb << sv)]

LB.bi Rt, [Ra], (Rb << sv)

Purpose: Load a zero-extended 8-bit byte from memory into a general register.

Description: This instruction loads a zero-extended byte from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + (Rb << sv) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + (Rb << sv) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format.

Operations:

```

Addr = Ra + (Rb << sv);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Bdata(7,0) = Load_Memory(PAddr, Byte, Attributes);
    Rt = Zero_Extend(Bdata(7,0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

8.1.31. LBI (Load Byte Immediate)

Type: 32-bit Baseline

Format:

LBI									
31	30	25	24	20	19	15	14	0	
0	LBI 000000		Rt		Ra			imm15s	

LBI.bi

31	30	25	24	20	19	15	14	0	
0	LBI.bi 000100		Rt		Ra			imm15s	

Syntax: LBI Rt, [Ra + imm15s]
 LBI.bi Rt, [Ra], imm15s

Purpose: Load a zero-extended 8-bit byte from memory into a general register.

Description: This instruction loads a zero-extended byte from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + SE(imm15s) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + SE(imm15s) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format. Note that imm15s is treated as a signed integer.

Operations:

```

Addr = Ra + Sign_Extend(imm15s);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Bdata(7,0) = Load_Memory(PAddr, BYTE, Attributes);
    Rt = Zero_Extend(Bdata(7,0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

“LBI R0, [R0+0]” baseline version 2 special behavior –

This instruction becomes a Reserved instruction when the INT_MASK.ALZ (INT_MASK[29]) is set to one. INT_MASK is also named as ir14. This special behavior can be used to debug a system. When it is used, compiler and assembler should avoid generating this instruction.

8.1.32. LBS (Load Byte Signed)

Type: 32-bit Baseline

Format:

LBS												
31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LBS 00010000		

LBS.bi

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LBS.bi 00010100		

Syntax: LBS Rt, [Ra + (Rb << sv)]

 LBS.bi Rt, [Ra], (Rb << sv)

Purpose: Load a sign-extended 8-bit byte from memory into a general register.

Description: This instruction loads a sign-extended byte from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + (Rb << sv) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + (Rb << sv) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format.

Operations:

```

Addr = Ra + (Rb << sv);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Bdata(7,0) = Load_Memory(PAddr, Byte, Attributes);
    Rt = Sign_Extend(Bdata(7,0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

8.1.33. LBSI (Load Byte Signed Immediate)

Type: 32-bit Baseline

Format:

LBI									
31	30	25	24	20	19	15	14	0	
0	LBSI 010000		Rt		Ra			imm15s	

LBI.bi

31	30	25	24	20	19	15	14	0	
0	LBSI.bi 010100		Rt		Ra			imm15s	

Syntax: LBSI Rt, [Ra + imm15s]

 LBSI.bi Rt, [Ra], imm15s

Purpose: Load a sign-extended 8-bit byte from memory into a general register.

Description: This instruction loads a sign-extended byte from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + SE(imm15s) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + SE(imm15s) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format. Note that imm15 is treated as a signed integer.

Operations:

```

Addr = Ra + Sign_Extend(imm15s);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Bdata(7,0) = Load_Memory(PAddr, BYTE, Attributes);
    Rt = Sign_Extend(Bdata(7,0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

8.1.34. LH (Load Halfword)

Type: 32-bit Baseline

Format:

LH												
31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LH 00000001		

LH.bi

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LH.bi 00000101		

Syntax: LH Rt, [Ra + (Rb << sv)]

 LH.bi Rt, [Ra], (Rb << sv)

Purpose: Load a zero-extended 16-bit halfword from memory into a general register.

Description:

This instruction loads a zero-extended halfword from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + (Rb << sv) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + (Rb << sv) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format.

The memory address has to be halfword-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + (Rb << sv);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
if (!Halfword_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Hdata(15, 0) = Load_Memory(PAddr, HALFWORD, Attributes);
    Rt = Zero_Extend(Hdata(15, 0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.35. LHI (Load Halfword Immediate)

Type: 32-bit Baseline

Format:

LHI									
31	30	25	24	20	19	15	14	0	
0	LHI 000001		Rt		Ra			imm15s	

LHI.bi

31	30	25	24	20	19	15	14	0	
0	LHI.bi 000101		Rt		Ra			imm15s	

Syntax: LHI Rt, [Ra + (imm15s << 1)]

 LHI.bi Rt, [Ra], (imm15s << 1)

(imm15s is a halfword offset. In assembly programming, always write a byte offset.)

Purpose: Load a zero-extended 16-bit halfword from memory into a general register.

Description:

This instruction loads a zero-extended halfword from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + SE(imm15s << 1) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + SE(imm15s << 1) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format. Note that imm15s is treated as a signed integer.

The memory address has to be half-word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + Sign_Extend(imm15s << 1);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
if (!Halfword_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Hdata(15, 0) = Load_Memory(PAddr, HALFWORD, Attributes);
    Rt = Zero_Extend(Hdata(15, 0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.36. LHS (Load Halfword Signed)

Type: 32-bit Baseline

Format:

LHS												
31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LHS 00010001		

LHS.bi

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LHS.bi 00010101		

Syntax: LHS Rt, [Ra + (Rb << sv)]

 LHS.bi Rt, [Ra], (Rb << sv)

Purpose: Load a sign-extended 16-bit halfword from memory into a general register.

Description:

This instruction loads a sign-extended halfword from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + (Rb << sv) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + (Rb << sv) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format.

The memory address has to be halfword-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + (Rb << sv);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
if (!Halfword_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Hdata(15, 0) = Load_Memory(PAddr, HALFWORD, Attributes);
    Rt = Sign_Extend(Hdata(15, 0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.37. LHSI (Load Halfword Signed Immediate)

Type: 32-bit Baseline

Format:

LHSI									
31	30	25	24	20	19	15	14	0	
0	LHSI 010001		Rt		Ra			imm15s	

LHSI.bi

31	30	25	24	20	19	15	14	0	
0	LHSI.bi 010101		Rt		Ra			imm15s	

Syntax: LHSI Rt, [Ra + (imm15s << 1)]

 LHSI.bi Rt, [Ra], (imm15s << 1)

(imm15s is a halfword offset. In assembly programming, always write a byte offset.)

Purpose: Load a sign-extended 16-bit halfword from memory into a general register.

Description:

This instruction loads a sign-extended halfword from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + SE(imm15s << 1) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + SE(imm15s << 1) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format. Note that imm15s is treated as a signed integer.

The memory address has to be half-word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + Sign_Extend(imm15s << 1);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
if (!Halfword_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Hdata(15, 0) = Load_Memory(PAddr, HALFWORD, Attributes);
    Rt = Sign_Extend(Hdata(15, 0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.38. LLW (Load Locked Word)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt			Ra		Rb		sv		LLW 00011000	

Syntax: LLW Rt, [Ra + (Rb << sv)]

Purpose: Used as a primitive to perform atomic read-modify-write operations.

Description:

LLW and SCW instructions are basic interlocking primitives to perform an atomic read (load-locked), modify, and write (store-conditional) sequence.

```

LLW Rx
... Modifying Rx
SCW Rx
BEQZ Rx

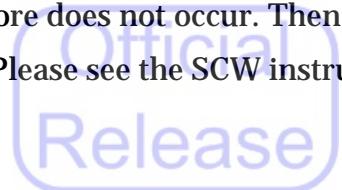
```

A LLW instruction begins the sequence and a SCW instruction completes the sequence. If this sequence can be performed without any intervening interruption or an interfering write from another processor or I/O module, the SCW instruction succeeds. Otherwise the SCW instruction fails and the program has to retry the sequence. There can only be one such active read-modify-write sequence per processor at any time. If a new LLW instruction is issued before a SCW instruction completes an active sequence, the new LLW instruction will start a new sequence which replaces the previous one.

The LLW instruction loads an aligned 32-bit word from a word-aligned memory address calculated by adding Ra and (Rb << sv). The word from memory is loaded into register Rt. When a LLW instruction is executed without generating any exceptions, the processor remembers the

loaded physical word address (Locked Physical Address) and sets a per-processor lock flag.

If the lock flag is still set when a SCW instruction is executed and the stored physical address is the same as the aligned address of the remembered LLW physical address, the store happens; otherwise, the store does not occur. Then, the success or failure status is stored back into the source register (Please see the SCW instruction description in Section 8.1.69 for detailed definition.)



The per-processor lock flag is cleared if the following events happen:

- Any execution of an IRET instruction.
- A coherent store is completed by another processor or coherent I/O module to the “Lock Region” containing the Locked Physical Address. The definition of the “Lock Region” is an aligned power-of-2 byte memory region. Though being implementation-dependent, the exact size of the region is within the range of at least 4-byte and at most the default minimum page size. The coherency is enforced either by hardware coherent mechanisms or by software using CCTL instructions on this processor through an interrupt mechanism. The coherent store event can be caused by a regular store, a store_conditional, and DPREF/Cache-Line-Write instructions.
- The completion of a SCW instruction on either success or failure condition.

Portable software should avoid putting memory access or CCTL instructions between execution of LLW and SCW instructions since it may cause SCW to fail. (For example, a store word operation to the same physical address of the Locked Physical Address.)

Operations:

```

VA = Ra + (Rb << sv);
If (VA(1, 0) != 0) {
  Generate_Exception(Data_alignment_check);
}
(PA, Attributes) = Address_Translation(VA, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
  Rt = Load_Memory(PA, WORD, Attributes);
  Locked_Physical_Address = PA;
  Lock_Flag = 1;
} else {
  Generate_Exception(Excep_status);
}

```

Exceptions: Alignment check, TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Usage (Note):

A very long instruction sequence between LLW and SCW may always fail the SCW instruction due to periodic timer interrupt. Software should take this into consideration when constructing the LLW and SCW instruction sequences.

Additional Software Constraints:

For N1213 hardcore N1213_43U1HA0 (CPU_VER==0x0C010003), follow the below software constraint to ensure correct LLW/SCW operations.

- If you use LLW/SCW in an interruption handler, make sure that
 - Execution of a LLW instruction must lead to execution of a SCW instruction. An UPREDICTABLE result may happen if execution of a LLW instruction eventually leads to execution of an IRET instruction without going through a SCW instruction.

8.1.39. LMW (Load Multiple Word)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	6	5	4	3	2	10
0	LSMW 011101	Rb	Ra		Re		Enable4		LMW 0	b:0 a:1	i:0 d:1	m		00	

Syntax: LMW.{b| a}{i | d}{m?} Rb, [Ra], Re, Enable4

Purpose: Load multiple 32-bit words from sequential memory locations into multiple registers.

Description:

This instruction loads multiple 32-bit words from sequential memory addresses specified by the base address register Ra and the {b | a}{i | d} options into a continuous range or a subset of general purpose registers. The subset of general purpose registers is specified by a register list formed by Rb, Re, and the four-bit Enable4 field as follows.

<Register List> = a range from [Rb, Re] and a list from <Enable4>

- {i | d} option specifies the direction of the address change. {i} generates increasing addresses from Ra and {d} generates decreasing addresses from Ra.
- {b | a} option specifies the way how the first address is generated. {b} uses the contents of Ra as the first memory load address. {a} uses either Ra+4 or Ra-4 for the {i | d} option respectively as the first memory load address.
- {m?} option, if it is specified, indicates that the base address register will be updated to the value computed in the following formula at the completion of this instruction.

TNReg = Total number of registers loaded

Updated value = Ra + (4 * TNReg) for {i} option

Updated value = Ra - (4 * TNReg) for {d} option

- [Rb, Re] specifies a range of registers which will be loaded by this instruction. Rb(4,0)

specifies the first register number in the continuous register range and Re(4,0) specifies the last register number. In addition to the range of registers, <Enable4(3,0)> specifies the load of 4 individual registers from R28 to R31 (s9/fp, gp, lp, sp) which have special calling convention usage. The exact mapping of Enable4(3,0) bits and registers is as follows:

Bits	Enable4(3) Format(9)	Enable4(2) Format(8)	Enable4(1) Format(7)	Enable4(0) Format(6)
Registers	R28	R29	R30	R31

- Several constraints are imposed for the <Register List>:
 - If [Rb(4,0), Re(4,0)] specifies at least one register:
 - ◆ Rb(4,0) <= Re(4,0) AND
 - ◆ 0 <= Rb(4,0), Re(4,0) < 28
 - If [Rb(4,0), Re(4,0)] specifies no register at all:
 - ◆ Rb(4,0) == Re(4,0) = 0b11111 AND
 - ◆ Enable4(3,0) != 0b0000
 - If these constraints are not met, UNPREDICTABLE results will happen to the contents of all registers after this instruction.
- The registers are loaded in sequence from matching memory locations. That is, the lowest-numbered register is loaded from the lowest memory address while the highest-numbered register is loaded from the highest memory address.
- If the base address register update {m?} option is specified while the base address register Ra is also specified in the <Register Specification>, there are two source values for the final content of the base address register Ra. In this case, the final value of Ra is UNPREDICTABLE. As to the rest of the loaded registers, they should have the values as if the base address register update {m?} option is not specified.
- This instruction can handle aligned/unaligned memory address.

Operation:

```

TNReg = Count_Registers(register_list);
if ("bi") {
    B_addr = Ra;
    E_addr = Ra + (TNReg * 4) - 4;
} elseif ("ai") {
    B_addr = Ra + 4;
    E_addr = Ra + (TNReg * 4);
} elseif ("bd") {
    B_addr = Ra - (TNReg * 4) + 4;
    E_addr = Ra;
} else { // "ad"
    B_addr = Ra - (TNReg * 4);
    E_addr = Ra - 4
}
VA = B_addr;
for (i = 0 to 31) {
    if (register_list[i] == 1) {
        (PA, Attributes) = Address_Translation(VA, PSW.DT);
        Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
        If (Excep_status == NO_EXCEPTION) {
            Ri = Load_Memory(PA, Word, Attributes);
            VA = VA + 4;
        } else {
            Generate_Exception(Excep_status);
        }
    }
}
if ("im") {
    Ra = Ra + (TNReg * 4);
} else { // "dm"
    Ra = Ra - (TNReg * 4);
}

```

Exception:

TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

- If the base register update is not specified, the base register value is unchanged. This applies even if the instruction loads its own base register and the loading of the base register occurs earlier than the exception event. For example, suppose the instruction is
`LMW.bd R2, [R4], R4, 0b0000`

And the implementation loads R4, then R3, and finally R2. If an exception occurs on any of the accesses, the value in the base register R4 of the instruction is unchanged.

- If the base register update is specified, the value left in the base register is unchanged.
- If the instruction loads only one general-purpose register, the value in that register is unchanged.
- If the instruction loads more than one general-purpose register, UNPREDICTABLE values are left in destination registers which are not the base register of the instruction.

Interruption: Whether this instruction is interruptible or not is implementation-dependent.

Privilege Level: all

Note:

- (1) LMW and SMW instructions do not guarantee atomicity among individual memory access operations. Neither do they guarantee single access to a memory location during the execution. Any I/O access that has side-effects other than simple stable memory-like access behavior should not use these two instructions.
- (2) The memory access order among the words accessed by LMW/SMW is not defined here and should be implementation-dependent. However, the more likely access order in an implementation is:
 - For LMW/SMW.i : increasing memory addresses from the base address.
 - For LMW/SMW.d: decreasing memory addresses from the base address.
- (3) The memory access order within an un-aligned word accessed is not defined here and should be implementation-dependent. However, the more likely access order in an implementation is:
 - For LMW/SMW.i: the aligned low address of the word and then the aligned high address of the word. If an interruption occurs, the EVA register will contain the starting low address of the un-aligned word.

- For LMW/SMW.d: the aligned high address of the word and then the aligned low address of the word. If an interruption occurs, the EVA register will contain “base un-aligned address + 4” of the first word or the starting low address of the remaining decreasing memory word.
- (4) Based on the more likely access order of (2) and (3), upon interruption, the EVA register for un-aligned LMW/SMW is more likely to have the following value:
- For LMW/SMW.i: the starting low addresses of the accessed words or “Ra + (TNReg * 4)” where TNReg represents the total number of registers loaded or stored.
 - For LMW/SMW.d: the starting low addresses of the accessed words or “Ra + 4”.

8.1.40. LW (Load Word)

Type: 32-bit Baseline

Format:

LW												
31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LW 00000010		

LW.bi

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LW.bi 00000110		

Syntax: LW Rt, [Ra + (Rb << sv)]

LW.bi Rt, [Ra], (Rb << sv)

Purpose: Load a 32-bit word from memory into a general register.

Description:

This instruction loads a word from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + (Rb << sv) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + (Rb << sv) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + (Rb << sv);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
if (!Word_Aligned(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Wdata(31, 0) = Load_Memory(PAddr, WORD, Attributes);
    Rt = Wdata(31, 0);
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.41. LWI (Load Word Immediate)

Type: 32-bit Baseline

Format:

LWI									
31	30	25	24	20	19	15	14	0	
0	LWI 000010		Rt		Ra			imm15s	

LWI.bi

31	30	25	24	20	19	15	14	0	
0	LWI.bi 000110		Rt		Ra			imm15s	

Syntax: LWI Rt, [Ra + (imm15s << 2)]

 LWI.bi Rt, [Ra], (imm15s << 2)

(imm15s is a word offset. In assembly programming, always write a byte offset.)

Purpose: Load a 32-bit word from memory into a general register.

Description:

This instruction loads a word from memory into the general register Rt. There are two different forms to specify the memory address: the regular form uses Ra + SE(imm15s << 2) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + SE(imm15s << 2) value after the memory load operation. An UNPREDICTABLE result will be written to Rt if Rt is specified as equal to Ra in the instruction format. Note that imm15s is treated as a signed integer.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + Sign_Extend(imm15s << 2);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
if (!Word_Aligned(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Wdata(31, 0) = Load_Memory(PAddr, WORD, Attributes);
    Rt = Wdata(31, 0);
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.42. LWUP (Load Word with User Privilege Translation)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LWUP 00100010		

Syntax: LWUP Rt, [Ra + (Rb << sv)]

Purpose: Load a 32-bit word from memory into a general register with the user mode privilege address translation.

Description: This instruction loads a word from the memory address Ra + (Rb << sv) into the general register Rt with the user mode privilege address translation regardless of the current processor operation mode (i.e. PSW.POM) and the current data address translation state (i.e. PSW.DT).

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Vaddr = Ra + (Rb << sv);
if (!Word_Aligned(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, TRANSLATE);
Excep_status = Page_Exception(Attributes, USER_MODE, LOAD);
if (Excep_status == NO_EXCEPTION) {
    Wdata(31, 0) = Load_Memory(PAddr, WORD, Attributes);
    Rt = Wdata(31, 0);
} else {
    Generate_Exception(Excep_status);
}
  
```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:



8.1.43. MADD32 (Multiply and Add to Data Low)

Type: 32-bit Baseline

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	000	Dt	0	Ra	Rb	0000	MADD32 110011						

Syntax: MADD32 Dt, Ra, Rb

Purpose: Multiply the contents of two 32-bit registers and add the lower 32-bit multiplication result to the lower 32-bit content of a 64-bit data register. Write the final result back to the lower 32-bit of the 64-bit data register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb, adds the lower 32-bit multiplication result to the content of Dt.LO 32-bit data register, and writes the final result back to Dt.LO data register. The contents of Ra and Rb can be either signed or unsigned integers.

Operations:

```

Mresul t = Ra * Rb;
Dt.LO = Dt.LO + Mresul t(31, 0);
  
```

Exceptions: None

Privilege level: All

Note:

8.1.44. MADD64 (Multiply and Add Unsigned)

Type: 32-bit Baseline

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	000	Dt	0	Ra	Rb	0000	MADD64 101011						

Syntax: MADD64 Dt, Ra, Rb

Purpose: Multiply the unsigned integer contents of two 32-bit registers and add the multiplication result to the content of a 64-bit data register. Write the final result back to the 64-bit data register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb, adds the 64-bit multiplication result to the content of Dt data register, and writes the final result back to Dt data register. The contents of Ra and Rb are treated as unsigned integers.

Operations:

```
Mresul t = CONCAT(1`b0, Ra) * CONCAT(1`b0, Rb);
Dt = Dt + Mresul t(63, 0);
```

Exceptions: None

Privilege level: All

Note:

8.1.45. MADDSS64 (Multiply and Add Signed)

Type: 32-bit Baseline

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	R000	Dt	0	Ra		Rb		0000	MADDSS64 101010				

Syntax: MADDSS64 Dt, Ra, Rb

Purpose: Multiply the signed integer contents of two 32-bit registers and add the multiplication result to the content of a 64-bit data register. Write the final result back to the 64-bit data register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb, adds the 64-bit multiplication result to the content of Dt data register, and writes the final result back to Dt data register. The contents of Ra and Rb are treated as signed integers.

Operations:

```

Mresult = Ra * Rb;
Dt = Dt + Mresult(63, 0);
  
```

Exceptions: None

Privilege level: All

Note:

8.1.46. MFSR (Move From System Register)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	10	9	5	4	0
0	MISC 110010	Rt			SRIDX		00000		MFSR 00010	

Syntax: MFSR Rt, SRIDX

Purpose: Move the content of a system register into a general register.

Description: This instruction moves the content of the system register specified by the SRIDX into the general register Rt.

Operations:

$GR[Rt] = SR[SR IDX];$

Exceptions: Privileged Instruction

Privilege level: Superuser and above

Note:

8.1.47. MFUSR (Move From User Special Register)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt	USR		Group		0000		MFUSR 100000			

Syntax: MFUSR Rt, USR_Name (= USR, Group)

Purpose: Move the content of a User Special Register to a general register.

Description: This instruction moves the content of a User Special Register specified by USR and Group into a general register Rt. The USR definitions are listed in the following tables.

Table 55. Group 0 MFUSR Definitions

Group	USR value	User Special Register
0	0	D0.LO
0	1	D0.HI
0	2	D1.LO
0	3	D1.HI
0	30-4	Reserved
0	31	PC (The PC value of this instruction)

Reading group 1 registers (see Table 56) in USER mode requires permission from PRIVILEGED mode resources (i.e. PRUSR_ACC_CTL register). If the reading permission is not enabled in USER mode, reading such a register will generate a Privileged Instruction exception. Privileged software should provide means for a user mode program to request such access permission. In addition, reading reserved registers will cause Reserved Instruction exceptions. Please check

AndeStar System Privilege Architecture Manual for detailed definitions of group 1 USR registers.

Table 56. Group 1 MFUSR Definitions

Group	USR value	User Special Register
1	0	DMA_CFG
1	1	DMA_GCSW
1	2	DMA_CHNSEL
1	3	DMA_ACT (Write only register, Read As Zero)
1	4	DMA_SETUP
1	5	DMA_ISADDR
1	6	DMA_ESADDR
1	7	DMA_TCNT
1	8	DMA_STATUS
1	9	DMA_2DSET
1	10-24	Reserved
1	25	DMA_2DSCTL
1	26-31	Reserved

Reading group 2 registers (see Table 57) in USER mode requires permission from PRIVILEGED mode resources (i.e. PRUSR_ACC_CTL register). If the reading permission is not enabled in USER mode, reading such a register will generate a Privileged Instruction exception. Privileged software should provide means for a user mode program to request such access permission. In addition, reading reserved registers will cause Reserved Instruction exceptions. Please check AndeStar System Privilege Architecture Version 3 Manual for detailed definitions of group 2 USR registers.

Table 57. Group 2 MFUSR Definitions

Group	USR value	User Special Register
2	0	PFMC0
2	1	PFMC1

Group	USR value	User Special Register
2	2	PFMC2
2	3	Reserved
2	4	PFM_CTL
2	5-31	Reserved

Operations:

Rt = User_Special_Register[Group][USR];

Exceptions: Privileged Instruction, Reserved Instruction

Privilege level: All

Note:

- (1) For PC register, there is no corresponding “MTUSR Rt, PC” instruction.
- (2) PC-relative memory access operation can be synthesized using the following code sequences:

L1: mfusr Ra, PC

L2: lwi Rs, [Ra + (offset relative to L1)]

8.1.48. MOVI (Move Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	0
0	MOVI 100010	Rt			imm20s	

Syntax: MOVI Rt, imm20s

Purpose: Initialize a register with a constant.

Description: This instruction moves the sign-extended imm20s into general register Rt.

Operations:

Rt = SE(i mm20s);

Exceptions: None

Privilege level: All

Note:

8.1.49. MSUB32 (Multiply and Subtract to Data Low)

Type: 32-bit Baseline

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	000	Dt	0	Ra	Rb	0000		MSUB32 110101					

Syntax: MSUB32 Dt, Ra, Rb

Purpose: Multiply the contents of two 32-bit registers and subtract the lower 32-bit multiplication result from the lower 32-bit content of a 64-bit data register. Write the final result back to the lower 32-bit of the 64-bit data register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb, subtracts the lower 32-bit multiplication result from the content of Dt.LO 32-bit data register, and writes the final result back to Dt.LO data register. The contents of Ra and Rb can be either signed or unsigned integers.

Operations:

```
Mresul t = Ra * Rb;
Dt.LO = Dt.LO - Mresul t(31, 0);
```

Exceptions: None

Privilege level: All

Note:

8.1.50. MSUB64 (Multiply and Subtract Unsigned)

Type: 32-bit Baseline

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	000	Dt	0	Ra	Rb	0000	MSUB64 101101						

Syntax: MSUB64 Dt, Ra, Rb

Purpose: Multiply the unsigned integer contents of two 32-bit registers and subtract the multiplication result from the content of a 64-bit data register. Write the final result back to the 64-bit data register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb, subtracts the 64-bit multiplication result from the content of Dt data register, and writes the final result back to Dt data register. The contents of Ra and Rb are treated as unsigned integers.

Operations:

```
Mresul t = CONCAT(1`b0, Ra) * CONCAT(1`b0, Rb);
Dt = Dt - Mresul t(63, 0);
```

Exceptions: None

Privilege level: All

Note:

8.1.51. MSUBS64 (Multiply and Subtract Signed)

Type: 32-bit Baseline

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	000	Dt	0	Ra		Rb		0000	MSUBS64 101100				

Syntax: MSUBS64 Dt, Ra, Rb

Purpose: Multiply the signed integer contents of two 32-bit registers and subtract the multiplication result from the content of a 64-bit data register. Write the final result back to the 64-bit data register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb, subtracts the 64-bit multiplication result from the content of Dt data register, and writes the final result back to Dt data register. The contents of Ra and Rb are treated as signed integers.

Operations:

```
Mresult = Ra * Rb;
Dt = Dt - Mresult(63, 0);
```

Exceptions: None

Privilege level: All

Note:

8.1.52. MSYNC (Memory Data Coherence Synchronization)

Type: 32-bit Baseline

Format:

	31	30	25	24	20	19	8	7	5	4	0
0	MISC 110010	00000	000000000000	SubType	MSYNC 01100						

Syntax: MSYNC SubType

Purpose: This is a collection of Memory Barrier operations to ensure completion (locally or globally) of memory load/store operations in some phases. This instruction orders loads and stores for synchronizing memory accesses between multiple Andes cores or between an Andes core and a Direct Memory Access agent.

Description:

This instruction is used for any non-strongly ordered load and store operations where software wants to ensure certain memory access order. It is based on an assumption that the hardware implementation does not automatically ensure the required ordering behavior. Actually, a hardware implementation may enforce more ordering behavior and if it does, this instruction becomes a NOP.

The following table lists the MSYNC SubType definitions:

Table 58. MSYNC SubType Definitions

Sub Type	Name
0	All
1	Store
2 - 7	Reserved

If this instruction uses a Reserved SubType, it will generate a Reserved Instruction Exception.

1. SubType 0, All:

For an implementation which does not support coherent caches, this operation ensures that all loads and *non-dirty* stores prior to this instruction complete before all loads and stores later than this instruction can start. *Non-dirty* stores include non-cacheable stores, and cacheable stores which have been written back implicitly or explicitly.

For an implementation which support coherent caches, this operation ensures that all loads and stores prior to this instruction complete before all loads and stores later than this instruction can start.

Completeness for a load means the destination register is written. Completeness for a store means the stored value is visible to all memory access agents in the system.

This operation does not enforce any ordering between load and store instructions and instruction fetches.

2. SubType 1, Store:

For an implementation which does not support coherent caches, this operation ensures that all *non-dirty* stores prior to this instruction complete before all loads and stores later than this instruction can start. *Non-dirty* stores include non-cacheable stores, and cacheable stores which have been written back implicitly or explicitly.

For an implementation which support coherent caches, this operation ensures that all stores prior to this instruction complete before all loads and stores later than this instruction can start.

Completeness for a store means the stored value is visible to all memory access agents in the system.

This operation does not enforce any ordering between load and store instructions and instruction fetches.

Operation:

```
If (SubType is not supported) {  
    Reserved_Instruction_Exception()  
} else {  
    MemoryDataSynchronization(SubType)  
}
```

Exceptions: Reserved instruction exception

Privilege level: All

Note:

8.1.53. MTSR (Move To System Register)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	10	9	5	4	0
0	MISC 110010	Ra			SRIDX		00000		MTSR 00011	

Syntax: MTSR Ra, SRIDX

Purpose: Move the content of a general register into a system register.

Description: This instruction moves the content of the general register Ra into the system register specified by the SRIDX.

Operations:

$SR[SRIDX] = GR[Ra];$

Exceptions: Privileged Instruction

Privilege level: Superuser and above

Note:

8.1.54. MTUSR (Move To User Special Register)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt	USR		Group		0000		MTUSR 100001			

Syntax: MTUSR Rt, USR_Name (= USR, Group)

Purpose: Move the content of a general register to a User Special Register.

Description: This instruction moves the content of a general register Rt into a User Special Register specified by USR and Group. The USR definitions are defined in the following tables.

Table 59. Group 0 MTUSR Definitions

Group	USR value	User Special Register
0	0	D0.LO
0	1	D0.HI
0	2	D1.LO
0	3	D1.HI
0	4-31	Reserved

Writing group 1 registers (see Table 60) in USER mode requires permission from PRIVILEGED mode resources (i.e. PRUSR_ACC_CTL register). If the writing permission is not enabled in USER mode, writing such a register will generate a Privileged Instruction exception. Privileged software should provide means for a user mode program to request such access permission. In addition, writing reserved registers will cause Reserved Instruction exceptions. Please note that the data dependency serializations of group 1 registers between MTUSR DMA_CHNSEL and MTUSR <Channel register> or between MTUSR and MFUSR requires the DSB instruction inserted in the middle. Please see AndeStar System Privilege Architecture Manual for definitions

of group 1 USR registers and more details.

Table 60. Group 1 MTUSR Definitions

Group	USR value	User Special Register
1	0	DMA_CFG (Read only, Write ignored)
1	1	DMA_GCSW (Read only, Write ignored)
1	2	DMA_CHNSEL
1	3	DMA_ACT
1	4	DMA_SETUP
1	5	DMA_ISADDR
1	6	DMA_ESADDR
1	7	DMA_TCNT
1	8	DMA_STATUS (Read only, Write ignored)
1	9	DMA_2DSET
1	10-24	Reserved
1	25	DMA_2DSCTL
1	26-31	Reserved

Writing group 2 registers (see Table 61) in USER mode requires permission from PRIVILEGED mode resources (i.e. PRUSR_ACC_CTL register). If the writing permission is not enabled in USER mode, writing such a register will generate a Privileged Instruction exception. Privileged software should provide means for a user mode program to request such access permission. In addition, writing reserved registers will cause Reserved Instruction exceptions. Please note that the data dependency serializations of group 2 registers between MTUSR PFM_CTL and MTUSR <PFMCx register> or between MTUSR and MFUSR requires the DSB instruction inserted in the middle. Please see AndeStar System Privilege Architecture Version 3 Manual for definitions of group 2 USR registers and more details.

Table 61. Group 2 MTUSR Definitions

Group	USR value	User Special Register
2	0	PFMC0
2	1	PFMC1
2	2	PFMC2
2	3	Reserved
2	4	PFM_CTL
2	5-31	Reserved

Operations:

User_Special_Register[Group][USR] = Rt;

Exceptions: Privileged Instruction, Reserved Instruction**Privilege level:** All**Note:**

8.1.55. MUL (Multiply Word to Register)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		Rb		0000		MUL 100100		

Syntax: MUL Rt, Ra, Rb

Purpose: Multiply the contents of two registers and write the result to a register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb and writes the lower 32-bit of the multiplication result to Rt. The contents of Ra and Rb can be signed or unsigned numbers.

Operations:

```
Mresul t = Ra * Rb;
Rt = Mresul t(31, 0);
```

Exceptions: None

Privilege level: All

Note:

8.1.56. MULT32 (Multiply Word to Data Low)

Type: 32-bit Baseline

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	R 000	Dt 0		Ra		Rb		0000	MULT32 110001				

Syntax: MULT32 Dt, Ra, Rb

Purpose: Multiply the contents of two 32-bit registers and write the result to the lower 32-bit of a 64-bit data register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb and writes the lower 32-bit multiplication result to Dt.LO 32-bit data register. The contents of Ra and Rb can be either signed or unsigned integers.

Operations:

```
Mresult = Ra * Rb;
Dt.LO = Mresult(31, 0);
```

Exceptions: None

Privilege level: All

Note:

8.1.57. MULT64 (Multiply Word Unsigned)

Type: 32-bit Baseline

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	R 000	Dt	0	Ra		Rb		0000	MULT64 101001				

Syntax: MULT64 Dt, Ra, Rb

Purpose: Multiply the unsigned integer contents of two 32-bit registers and write the result to a 64-bit data register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb and writes the 64-bit multiplication result to Dt data register. The contents of Ra and Rb are treated as unsigned integers.

Operations:

```
Mresul t = CONCAT(1`b0, Ra) * CONCAT(1`b0, Rb);
Dt = Mresul t(63, 0);
```

Exceptions: None

Privilege level: All

Note:

8.1.58. MULTS64 (Multiply Word Signed)

Type: 32-bit Baseline

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	000	Dt	0	Ra	Rb	0000	MULTS64 101000						

Syntax: MULTS64 Dt, Ra, Rb

Purpose: Multiply the signed integer contents of two 32-bit registers and write the result to a 64-bit data register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb and writes the 64-bit multiplication result to Dt data register. The contents of Ra and Rb are treated as signed integers.

Operations:

```
Mresult = Ra * Rb;
Dt = Mresult(63, 0);
```

Exceptions: None

Privilege level: All

Note:

8.1.59. NOP (No Operation)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	NOP 00000	NOP 00000	NOP 00000					00000		SRLI 01001	

Syntax: NOP

Purpose: Perform no operation.

Description: This instruction does nothing. It is aliased to “SRLI R0, R0, 0”, but treated by an implementation as a true NOP.

Operations:

;

Exceptions: None

Privilege level: All

Note:

8.1.60. NOR (Bit-wise Logical Nor)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt	Ra	Rb	00000	NOR 00101						

Syntax: NOR Rt, Ra, Rb

Purpose: Perform a bit-wise logical NOR operation on the content of two registers.

Description: This instruction combines the content of Ra with that of Rb using a bit-wise logical NOR operation and writes the result to Rt.

Operations:

$$Rt = \sim(Ra \mid Rb);$$

Exceptions: None

Privilege level: All

Note:

8.1.61. OR (Bit-wise Logical Or)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		OR 00100		

Syntax: OR Rt, Ra, Rb

Purpose: Perform a bit-wise logical OR operation on the content of two registers.

Description: This instruction combines the content of Ra with that of Rb using a bit-wise logical OR operation and writes the result to Rt.

Operations:

$$Rt = Ra \mid Rb;$$

Exceptions: None

Privilege level: All

Note:

8.1.62. ORI (Or Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	0
0	ORI 101100	Rt		Ra				imm15u

Syntax: ORI Rt, Ra, imm15u

Purpose: Bit-wise OR of the content of a register with an unsigned constant.

Description: This instruction combines the content of Ra with the zero-extended imm15u using a bit-wise logical OR operation and writes the result to Rt.

Operations:

Rt = Ra | ZE(imm15u);

Exceptions: None

Privilege level: All

Note:

8.1.63. RET (Return from Register)

Type: 32-bit Baseline

Format:

31	30	25	24	15	14	10	9	8	7	6	5	4	0
0	JREG 100101	0000000000		Rb		DT/IT 00		00		RET 1		JR 00000	

Syntax: RET Rb

Purpose: Make an unconditional function call return to an instruction address stored in a register.

Description: This instruction branches unconditionally to an instruction address stored in Rb. Note that the architecture behavior of this instruction is the same as that of the JR instruction. Even so, software uses this instruction instead of JR for function call return. This facilitates software to distinguish the two different usages and is helpful in call stack backtracing applications. Distinguishing a function return jump from a regular jump also helps implementation performance (e.g. return address prediction).

Operations:

PC = Rb;

Exceptions: None

Privilege level: All

Note:

8.1.64. RET.xTOFF (Return from Register and Translation OFF)

Type: 32-bit Baseline (with MMU configuration)

Format:

RET.ITOFF													
31	30	25	24	15	14	10	9	8	7	6	5	4	0
0	JREG 100101	0000000000		Rb		DT/IT 01		00		RET 1		JR 00000	

RET.TOFF

31	30	25	24	15	14	10	9	8	7	6	5	4	0
0	JREG 100101	0000000000		Rb		DT/IT 11		00		RET 1		JR 00000	

Syntax: RET.[T | IT]OFF Rb

Purpose: Make an unconditional function call return to an instruction address stored in a register and turn off address translation for the target instruction.

Description: This instruction branches unconditionally to an instruction address stored in Rb. It also clears the IT (and DT if included) field of the Processor Status Word (PSW) system register to turn off the instruction (and data if included) address translation process in the memory management unit. This instruction guarantees that fetching of the target instruction sees PSW.IT as 0 (and PSW.DT as 0 if included) and thus does not go through the address translation process.

Note that the architecture behavior of this instruction is the same as that of the JR.xTOFF instruction. Even so, software uses this instruction instead of JR.xTOFF for function call return. This facilitates software to distinguish the two different usages and is helpful in call stack backtracing applications. Distinguishing a function return jump from a regular jump also helps implementation performance (e.g. return address prediction).

Operations:

```
PC = Rb;  
PSW.IT = 0;  
if (INST(9) == 1) {  
    PSW.DT = 0;  
}
```



Exceptions: Privileged Instruction, Reserved Instruction (for non-MMU configuration)

Privilege level: Superuser and above

Note: This instruction is used in an interruption handler or privileged code in a translated address space to return to a place in a non-translated address space. Please see the JRAL.xTON instruction for the reverse process of coming from a non-translated address space to a translated address space.

8.1.65. ROTR (Rotate Right)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000	ROTR 01111			

Syntax: ROTR Rt, Ra, Rb

Purpose: Perform a right rotation operation on the content of a register.

Description: This instruction right-rotates the content of Ra and writes the result to Rt. The rotation amount is specified by the low-order 5-bits of the Rb register.

Operations:

```
ra = Rb(4, 0);
Rt = CONCAT(Ra(ra-1, 0), Ra(31, ra));
```

Exceptions: None

Privilege level: All

Note:

8.1.66. ROTRI (Rotate Right Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt	Ra	imm5u	00000	ROTRI 01011						

Syntax: ROTRI Rt, Ra, imm5u

Purpose: Perform a right rotation operation on the content of a register.

Description: This instruction right-rotates the content of Ra and writes the result to Rt. The rotation amount is specified by the imm5u constant.

Operations:

Rt = CONCAT(Ra(i mm5u-1, 0), Ra(31, i mm5u));

Exceptions: None

Privilege level: All

Note:

8.1.67. SB (Store Byte)

Type: 32-bit Baseline

Format:

SB												
31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		SB 00001000		

SB.bi

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		SB.bi 00001100		

Syntax: SB Rt, [Ra + (Rb << sv)]

SB.bi Rt, [Ra], (Rb << sv)

Purpose: Store an 8-bit byte from a general register into memory.

Description: This instruction stores the least-significant 8-bit byte in the general register Rt to memory. There are two different forms to specify the memory address: the regular form uses Ra + (Rb << sv) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + (Rb << sv) value after the memory store operation.

Operations:

```

Addr = Ra + (Rb << sv);
If (.bi_form) {
  Vaddr = Ra;
} else {
  Vaddr = Addr;
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
  Store_Memory(PAddr, BYTE, Attributes, Rt(7,0));
  If (.bi_form) { Ra = Addr; }
} else {
  Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error

Privilege level: All

Note:

8.1.68. SBI (Store Byte Immediate)

Type: 32-bit Baseline

Format:

SBI									
31	30	25	24	20	19	15	14	0	
0	SBI 001000		Rt		Ra			imm15s	

SBI.bi

31	30	25	24	20	19	15	14	0	
0	SBI.bi 001100		Rt		Ra			imm15s	

Syntax: SBI Rt, [Ra + imm15s]

 SBI.bi Rt, [Ra], imm15s

Purpose: Store an 8-bit byte from a general register into a memory location.

Description: This instruction stores the least-significant 8-bit byte in the general register Rt to the memory location. There are two different forms to specify the memory address: the regular form uses Ra + SE(imm15s) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + SE(imm15s) value after the memory store operation.

Note that imm15s is treated as a signed integer.

Operations:

```

Addr = Ra + Sign_Extend(imm15s);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
    Store_Memory(PAddr, BYTE, Attributes, Rt(7,0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

8.1.69. SCW (Store Conditional Word)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt			Ra		Rb		sv		SCW 00011001	

Syntax: SCW Rt, [Ra + (Rb << sv)]

Purpose: Used as a primitive to perform atomic read-modify-write operations.

Description:

LLW and SCW instructions are basic interlocking primitives to perform an atomic read (load-locked), modify, and write (store-conditional) sequence.

LLW Rx

Modify Rx

SCW Rx

BEQZ Rx

A LLW instruction begins the sequence and a SCW instruction completes the sequence. If this sequence can be performed without any intervening interruption or an interfering write from another processor or I/O module, the SCW instruction succeeds. Otherwise the SCW instruction fails and the program has to retry the sequence. There can only be one such active read-modify-write sequence per processor at any time. If a new LLW instruction is issued before a SCW instruction completes an active sequence, the new LLW instruction will start a new sequence which replaces the previous one.

The SCW instruction conditionally stores a 32-bit word from register Rt to a word-aligned memory address calculated by adding Ra and (Rb << sv). If all the following conditions are true, the memory store operation happens and a result of 1 is written to the general register Rt to

indicate a success status.

- The Lock_Flag is 1 (Note: any exception generated by the SCW instruction will clear the Lock_Flag)
- The word-aligned store physical address is the same as the aligned address of the Locked_Physical_Address generated by the previous LLW instruction.

If the Lock_Flag is 0, the store operation is not performed and a result of 0 is written to the general register Rt to indicate a failure status.

If the word-aligned store physical address is not the same as the aligned address of the Locked_Physical_Address generated by the previous LLW instruction, it is UNPREDICTABLE (implementation-dependent) whether the store operation will be performed. However, the final status (success or failure) will be consistent with the implementation's action.

The per-processor lock flag is cleared if the following events happen:

- Any execution of an IRET instruction.
- A coherent store is completed by another processor or coherent I/O module to the “Lock Region” containing the Locked Physical Address. The definition of the “Lock Region” is an aligned power-of-2 byte memory region. Though implementation-dependent, the exact size of the region is within the range of at least 4-byte and at most the default minimum page size. The coherency is enforced either by hardware coherent mechanisms or by software using CCTL instructions on this processor through an interrupt mechanism. The coherent store event can be caused by a regular store, a store_conditional, and DPREF/Cache-Line-Write instructions.
- The completion of a SCW instruction on either success or failure condition.

Portable software should avoid putting memory access or CCTL instructions between the execution of LLW and SCW instructions since it may cause SCW to fail. (For example, a store word operation to the same physical address of the Locked Physical Address.)

Operations:

```

VA = Ra + (Rb << sv);
If (VA(1, 0) != 0) {
    Generate_Exception(Data_alignment_check);
}
(PA, Attributes) = Address_Translation(VA, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
    If (Lock_Flag == 1)
        If (PA == Locked_Physical_Address) {
            Store_Memory(PA, Attributes, Rt);
            Rt = 1;
        } else {
            Implementation-dependent for success or fail;
        }
    } else {
        Rt = 0;
    }
    Lock_Flag = 0;
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: Alignment check, TLB fill, Non-leaf PTE not present, Leaf PTE not present, Write protection, Page modified, Access bit, TLB VLPT miss.

Privilege level: All

Usage Note: A very long instruction sequence between LLW and SCW may always fail the SCW instruction due to periodic timer interrupt. Software should take this into consideration when constructing LLW and SCW instruction sequences.

Implementation Note:

The SCW instruction conditionally executes in the memory system, depending on the cache coherence design in the memory system. Thus, an implementation needs to prevent SCW from being invalidated by any interruption after the SCW request enters the memory system until the

success/failure status returns from the memory system to complete the SCW instruction.

For an implementation with a non-coherent cache system, an internal and an external flag may be needed to implement the “lock” state. For such a system, the success or failure of the SCW instruction is determined by either both flags or internal flag alone. The conditions based on different memory attributes are summarized as follows.

	Non-cacheable	Cacheable	
		Write-back	Write-through
Success/fail determined by	Internal and external flags	Internal flag	Internal flag

Note that if the memory location that SCW operates on is cacheable, only the internal flag will be used and the external flag will be ignored.

Additional Software Constraints:

For N1213 hardcore N1213_43U1HA0 (CPU_VER==0x0C010003), follow the below software constraints to ensure correct LLW/SCW operations.

- If you use LLW/SCW in an interruption handler, make sure that
 - Execution of a LLW instruction must lead to execution of a SCW instruction. An UPREDICTABLE result may happen if execution of a LLW instruction eventually leads to execution of an IRET instruction without going through a SCW instruction.

8.1.70. SEB (Sign Extend Byte)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	5	4	0
0	ALU_1 100000	Rt		Ra			0000000000		SEB 10000	

Syntax: SEB Rt, Ra

Purpose: Sign-extend the least-significant-byte of a register.

Description: This instruction sign-extends the least-significant-byte of Ra and writes the result to Rt.

Operations:

$Rt = SE(Ra(7, 0));$

Exceptions: None

Privilege level: All

Note:

8.1.71. SEH (Sign Extend Halfword)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	5	4	0
0	ALU_1 100000	Rt		Ra		0000000000		SEH 10001		

Syntax: SEH Rt, Ra

Purpose: Sign-extend the least-significant-halfword of a register.

Description: This instruction sign-extends the least-significant-halfword of Ra and writes the result to Rt.

Operations:

$Rt = SE(Ra(15, 0));$

Exceptions: None

Privilege level: All

Note:

8.1.72. SETEND (Set data endian)

Type: 32-bit Baseline

Format:

31	30	25	24	21	20	19	10	9	5	4	0
0	MISC 110010	0000	BE			PSW_IDX 0010000000		SETEND 00001		MTSR 00011	

Syntax: SETEND.B (Set data endian to big endian)

SETEND.L (Set data endian to little endian)

Purpose: Control the data endian mode in the PSW register.

Description:

This instruction has two flavors. The SETEND.B sets the data endian mode to big-endian while the SETEND.L sets the data endian mode to little-endian. Note that this instruction can be used in user mode. The BE bit in this instruction encoding distinguishes between these two flavors.

BE	Flavor
0	SETEND.L
1	SETEND.B

Operations:

```
SR[PSW].BE = 1; // SETEND.B
SR[PSW].BE = 0; // SETEND.L
```

Exceptions: None

Privilege level: All

Note: A DSB instruction must follow a SETEND instruction to guarantee that any subsequent load/store instruction can observe the just updated PSW.BE value.

8.1.73. SETGIE (Set global interrupt enable)

Type: 32-bit Baseline

Format:

31	30	25	24	21	20	19	10	9	5	4	0
0	MISC 110010	0000	EN			PSW_IDX 0010000000		SETGIE 00010		MTSR 00011	

Syntax: SETGIE.E (Enable global interrupt)

SETGIE.D (Disable global interrupt)

Purpose: Control the global interrupt enable bit in the PSW register.

Description:

This instruction has two flavors. The SETGIE.E enables the global interrupt while the SETGIE.D disables the global interrupt. The EN bit in this instruction encoding distinguishes between these two flavors.

EN	Flavor
0	SETGIE.D
1	SETGIE.E

Operations:

```
SR[PSW].GIE = 1; // SETGIE.E
SR[PSW].GIE = 0; // SETGIE.D
```

Exceptions: None

Privilege level: Superuser and above

Note: A DSB instruction must follow a SETGIE instruction to guarantee that any subsequent instruction can observe the just updated PSW.GIE value for the external interrupt interruption.

8.1.74. SETHI (Set High Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	0
0	SETHI 100011	Rt			imm20u	

Syntax: SETHI Rt, imm20u

Purpose: Initialize the high portion of a register with a constant.

Description: This instruction moves the imm20u into the upper 20-bits of general register Rt and fills the lower 12-bits of Rt with 0.

Operations:

Rt = imm20u << 12;

Exceptions: None

Privilege level: All

Note:

8.1.75. SH (Store Halfword)

Type: 32-bit Baseline

Format:

SH												
31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		SH 00001001		

SH.bi

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		SH.bi 00001101		

Syntax: SH Rt, [Ra + (Rb << sv)]

SH.bi Rt, [Ra], (Rb << sv)

Purpose: Store a 16-bit halfword from a general register into memory.

Description:

This instruction stores the least-significant 16-bit halfword in the general register Rt to memory. There are two different forms to specify the memory address: the regular form uses Ra + (Rb << sv) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + (Rb << sv) value after the memory store operation.

The memory address has to be halfword-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + (Rb << sv);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
if (!Halfword_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
    Store_Memory(PAddr, HALFWORD, Attributes, Rt(15, 0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.76. SHI (Store Halfword Immediate)

Type: 32-bit Baseline

Format:

SHI									
31	30	25	24	20	19	15	14	0	
0	SHI 001001		Rt		Ra			imm15s	

SHI.bi

31	30	25	24	20	19	15	14	0	
0	SHI.bi 001101		Rt		Ra			imm15s	

Syntax: SHI Rt, [Ra + (imm15s << 1)]

SHI.bi Rt, [Ra], (imm15s << 1)

(imm15s is a halfword offset. In assembly programming, always write a byte offset.)

Purpose: Store a 16-bit halfword from a general register into memory.

Description:

This instruction stores the least-significant 16-bit halfword in the general register Rt to memory.

There are two different forms to specify the memory address: the regular form uses Ra +

SE(imm15s << 1) as its memory address while the .bi form uses Ra. For the .bi form, the Ra

register is updated with the Ra + SE(imm15s << 1) value after the memory store operation. Note that imm15s is treated as a signed integer.

The memory address has to be half-word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + Sign_Extend(imm15s << 1);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
if (!Halfword_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
    Store_Memory(PAddr, HALFWORD, Attributes, Rt(15, 0));
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.77. SLL (Shift Left Logical)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000	SLL 01100			

Syntax: SLL Rt, Ra, Rb

Purpose: Perform a logical left shift operation on the content of a register.

Description: This instruction left-shifts the content of Ra logically and writes the result to Rt. The shifted out bits are filled with zero and the shift amount is specified by the low-order 5-bits of the Rb register.

Operations:

```
sa = Rb(4, 0);
Rt = CONCAT(Ra(31-sa, 0), sa`b0);
```

Exceptions: None

Privilege level: All

Note:

8.1.78. SLLI (Shift Left Logical Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt	Ra		imm5u	00000		SLLI 01000				

Syntax: SLLI Rt, Ra, imm5u

Purpose: Perform a logical left shift operation on the content of a register.

Description: This instruction left-shifts the content of Ra logically and writes the result to Rt. The shifted out bits are filled with zero and the shift amount is specified by the imm5u constant.

Operations:

Rt = CONCAT(Ra(31-*i* mm5u, 0), *i* mm5u` b0);

Exceptions: None

Privilege level: All

Note:

8.1.79. SLT (Set on Less Than)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt	Ra	Rb	00000	SLT 00110						

Syntax: SLT Rt, Ra, Rb

Purpose: Unsigned-compare the contents of two registers.

Description: The content of Ra is unsigned-compared with that of Rb. If the content of Ra is less than that of Rb, this instruction writes a result of 1 to Rt; otherwise, it writes a result of 0 to Rt.

Operations:

```
if (CONCAT(1`b0, Ra) < CONCAT(1`b0, Rb)) {
  Rt = 1;
} else {
  Rt = 0;
}
```

Exceptions: None

Privilege level: All

Note:

8.1.80. SLTI (Set on Less Than Immediate)

Type: 32-bit Baseline

Format:

	31	30	25	24	20	19	15	14	0
0		SLTI 101110	Rt		Ra				imm15s

Syntax: SLTI Rt, Ra, imm15s

Purpose: Unsigned-compare the content of a register with a signed constant.

Description: The content of Ra is unsigned-compared with a sign-extended imm15s. If the content of Ra is less than the sign-extended imm15s, this instruction writes the result of 1 to Rt; otherwise, it writes the result of 0 to Rt. The sign-extended imm15s will generate an unsigned constant in the following range:

$$[2^{32}-1, 2^{32}-2^{14}] \text{ and } [2^{14}-1, 0]$$

Operations:

$Rt = (Ra \text{ (unsigned)} < SE(\text{imm15s})) ? 1 : 0;$

Exceptions: None

Privilege level: All

Note:

8.1.81. SLTS (Set on Less Than Signed)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt			Ra		Rb		00000		SLTS 00111	

Syntax: SLTS Rt, Ra, Rb

Purpose: Signed-compare the contents of two registers.

Description: The content of Ra is signed-compared with that of Rb. If the content of Ra is less than that of Rb, this instruction writes a result of 1 to Rt; otherwise, it writes a result of 0 to Rt.

Operations:

```
if (Ra < Rb) {
    Rt = 1;
} else {
    Rt = 0;
}
```

Exceptions: None

Privilege level: All

Note:

8.1.82. SLTSI (Set on Less Than Signed Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	0
0	SLTSI 101111	Rt		Ra				imm15s

Syntax: SLTSI Rt, Ra, imm15s

Purpose: Signed-compare the content of a register with a constant.

Description: The content of Ra is signed-compared with the sign-extended imm15s. If the content of Ra is less than the sign-extended imm15s, this instruction writes the result of 1 to Rt; otherwise, it writes the result of 0 to Rt. The sign-extended imm15s will generate a signed constant in the following range:

$$[2^{14}-1, 0] \text{ and } [-1, -2^{14}]$$

Operations:

$Rt = (Ra \text{ (signed)} < SE(imm15s)) ? 1 : 0;$

Exceptions: None

Privilege level: All

Note:

8.1.83. SMW (Store Multiple Word)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	6	5	4	3	2	10
0	LSMW 011101	Rb	Ra		Re		Enable4		SMW 1	b:0 a:1	i:0 d:1	m		00	

Syntax: SMW.{b | a}{i | d}{m?} Rb, [Ra], Re, Enable4

Purpose: Store multiple 32-bit words from multiple registers into sequential memory locations.

Description:

This instruction stores multiple 32-bit words from a range or a subset of source general-purpose registers to sequential memory addresses specified by the base address register Ra and the {b | a}{i | d} options. The source registers are specified by a register list formed by Rb, Re, and the four-bit Enable4 field as follows.

<Register List> = a range from {Rb, Re} and a list from <Enable4>

- {i | d} option specifies the direction of the address change. {i} generates increasing addresses from Ra and {d} generates decreasing addresses from Ra.
- {b | a} option specifies the way how the first address is generated. {b} uses the contents of Ra as the first memory store address. {a} uses either Ra+4 or Ra-4 for the {i | d} option respectively as the first memory store address.
- {m?} option, if it is specified, indicates that the base address register will be updated to the value computed in the following formula at the completion of this instruction.

TNReg = Total number of registers stored

Updated value = Ra + (4 * TNReg) for {i} option

Updated value = Ra - (4 * TNReg) for {d} option

- [Rb, Re] specifies a range of registers whose contents will be stored by this instruction.

Rb(4,0) specifies the first register number in the continuous register range and Re(4,0) specifies the last register number. In addition to the range of registers, <Enable4(3,0)> specifies the store of 4 individual registers from R28 to R31 (s9/fp, gp, lp, sp) which have special calling convention usage. The exact mapping of Enable4(3,0) bits and registers is as follows:

Bits	Enable4(3) Format(9)	Enable4(2) Format(8)	Enable4(1) Format(7)	Enable4(0) Format(6)
Registers	R28	R29	R30	R31

- Several constraints are imposed for the <Register List>:
 - If [Rb, Re] specifies at least one register:
 - ◆ Rb(4,0) <= Re(4,0) AND
 - ◆ 0 <= Rb(4,0), Re(4,0) < 28
 - If [Rb, Re] specifies no register at all:
 - ◆ Rb(4,0) == Re(4,0) = 0b11111 AND
 - ◆ Enable4(3,0) != 0b0000
 - If these constraints are not met, UNPREDICTABLE results will happen to the contents of the memory range pointed to by the base register. If the {m?} option is also specified after this instruction, an UNPREDICTABLE result will happen to the base register itself too.
- The registers are stored in sequence to matching memory locations. That is, the lowest-numbered register is stored to the lowest memory address while the highest-numbered register is stored to the highest memory address.
- If the base address register Ra is specified in the <Register Specification>, the value stored to memory from the register Ra is the Ra value before this instruction is executed.
- This instruction can handle aligned/unaligned memory address.

Operation:

```

TNReg = Count_Registers(register_list);
if ("bi") {
    B_addr = Ra;
    E_addr = Ra + (TNReg * 4) - 4;
} elseif ("ai") {
    B_addr = Ra + 4;
    E_addr = Ra + (TNReg * 4);
} elseif ("bd") {
    B_addr = Ra - (TNReg * 4) + 4;
    E_addr = Ra;
} else { // "ad"
    B_addr = Ra - (TNReg * 4);
    E_addr = Ra - 4
}
VA = B_addr;
for (i = 0 to 31) {
    if (register_list[i] == 1) {
        (PA, Attributes) = Address_Translation(VA, PSW.DT);
        Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
        If (Excep_status == NO_EXCEPTION) {
            Store_Memory(PA, Word, Attributes, Ri);
            VA = VA + 4;
        } else {
            Generate_Exception(Excep_status);
        }
    }
}
if ("im") {
    Ra = Ra + (TNReg * 4);
} else { // "dm"
    Ra = Ra - (TNReg * 4);
}

```

Exception:

TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

- The base register value is left unchanged on an exception event, no matter whether the

base register update is specified or not.

Interruption: Whether this instruction is interruptible or not is implementation-dependent.

Privilege Level: all



Note:

- (1) LMW and SMW instructions do not guarantee atomicity among individual memory access operations. Neither do they guarantee single access to a memory location during the execution. Any I/O access that has side-effects other than simple stable memory-like access behavior should not use these two instructions. For non-translated AndeStar address space attributes NTC of type 0 and type 1 (non-cacheable) and for translated AndeStar address space attribute C of type 0 and type 1 (device space), LMW and SMW instructions should not be used.
- (2) The cacheability AndeStar address space attribute of the memory region updated by a SMW instruction should be the same type. If the memory region has more than one cacheability types, the result of the SMW instruction is unpredictable.
- (3) The memory access order among the words accessed by LMW/SMW is not defined here and should be implementation-dependent. However, the more likely access order in an implementation is:
 - For LMW/SMW.i : increasing memory addresses from the base address.
 - For LMW/SMW.d: decreasing memory addresses from the base address.
- (4) The memory access order within an un-aligned word accessed is not defined here and should be implementation-dependent. However, the more likely access order in an implementation is:
 - For LMW/SMW.i: the aligned low address of the word and then the aligned high address of the word. If an interruption occurs, the EVA register will contain the starting low address of the un-aligned word.
 - For LMW/SMW.d: the aligned high address of the word and then the aligned low address of the word. If an interruption occurs, the EVA register will contain “base un-aligned address + 4” of the first word or the starting low address of the remaining decreasing memory word.

(4) Based on the more likely access order of (2) and (3), upon interruption, the EVA register for un-aligned LMW/SMW is more likely to have the following value:

- For LMW/SMW.i: the starting low addresses of the accessed words or “Ra + (TNReg * 4)” where TNReg represents the total number of registers loaded or stored.
- For LMW/SMW.d: the starting low addresses of the accessed words or “Ra + 4”.



8.1.84. SRA (Shift Right Arithmetic)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		SRA 01110		

Syntax: SRA Rt, Ra, Rb

Purpose: Perform an arithmetic right shift operation on the content of a register.

Description: This instruction right-shifts the content of Ra arithmetically and writes the result to Rt. The shifted out bits are filled with the sign-bit Ra(31) and the shift amount is specified by the low-order 5-bits of the Rb register.

Operations:

```
sa = Rb(4, 0);
Rt = CONCAT(sa`bRa(31), Ra(31, sa));
```

Exceptions: None

Privilege level: All

Note:

8.1.85. SRAI (Shift Right Arithmetic Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt	Ra		imm5u	00000	SRAI 01010					

Syntax: SRAI Rt, Ra, imm5u

Purpose: Perform an arithmetic right shift operation on the content of a register.

Description: This instruction right-shifts the content of Ra arithmetically and writes the result to Rt. The shifted out bits are filled with sign-bit Ra(31) and the shift amount is specified by the imm5u constant.

Operations:

$Rt = \text{CONCAT}(\text{imm5u}^{\text{b}} \text{Ra}(31), \text{Ra}(31, \text{imm5u}))$;

Exceptions: None

Privilege level: All

Note:

8.1.86. SRL (Shift Right Logical)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		SRL 01101		

Syntax: SRL Rt, Ra, Rb

Purpose: Perform a logical right shift operation on the content of a register.

Description: This instruction right-shifts the content of Ra logically and writes the result to Rt. The shifted out bits are filled with zero and the shift amount is specified by the low-order 5-bits of the Rb register.

Operations:

```
sa = Rb(4, 0);
Rt = CONCAT(sa`b0, Ra(31, sa));
```

Exceptions: None

Privilege level: All

Note:

8.1.87. SRLI (Shift Right Logical Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		imm5u		00000		SRLI 01001		

Syntax: SRLI Rt, Ra, imm5u

Purpose: Perform a logical right shift operation on the content of a register.

Description: This instruction right-shifts the content of Ra logically and writes the result to Rt. The shifted out bits are filled with zero and the shift amount is specified by the imm5u constant.

Operations:

Rt = CONCAT(i mm5u` b0, Ra(31, i mm5u));

Exceptions: None

Privilege level: All

Note: “SRLI R0, R0, 0” is aliased to NOP and treated by an implementation as NOP.

8.1.88. STANDBY (Wait for External Event)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	10	9	7	6	5	4	0
0	MISC 110010	00000	0000000000		000	SubTy pe	STANDBY 00000					

Syntax: STANDBY SubType (= no_wake_grant, wake_grant)

Purpose: Make a core enter the standby state while waiting for external events to happen.

Description:

This instruction puts the core and its associating structures into an implementation-dependent low power standby mode where the instruction execution stops and most of the pipeline clocks can be disabled. The core has to enter the standby mode after all external memory and I/O accesses have been completed.

In general, the core leaves the standby mode when an external event happens and needs the core's attention. However, to facilitate the need for an external power manager to control the clock frequency and voltage, the wakeup action needs the external power manager's consent. Thus two flavors of the STANDBY instruction are defined to distinguish the different usages. The SubType field definitions are listed as follows.

Table 62. STANDBY Instruction SubType Definitions

SubType	Mnemonic	Wakeup Condition
0	no_wake_grant	The STANDBY instruction immediately monitors and accepts a wakeup event (e.g external interrupt), making the core leave the standby mode without waiting for a wakeup_consent notification from an external agent.

SubType	Mnemonic	Wakeup Condition
0	no_wake_grant	The STANDBY instruction immediately monitors and accepts a wakeup event (e.g external interrupt), making the core leave the standby mode without waiting for a wakeup_consent notification from an external agent.
1	wake_grant	The STANDBY instruction waits for a wakeup_consent notification from an external agent (e.g. power management unit) before monitoring and accepting a wakeup_event (e.g. external interrupt) to make the core leave the standby mode.
2	wait_done	The STANDBY instruction waits for a wakeup_consent notification from an external agent (e.g. power management unit). When the wakeup_consent notification arrives, the core leaves the standby mode immediately.

The wakeup external events include interrupt (regardless of masking condition), debug request, wakeup signal, reset etc. The instruction execution restarts either from the instruction following the STANDBY instruction or from the enabled interrupt handler which causes the core to leave the standby mode. When entering an interrupt handler, the IPC system register keeps the address of the instruction following the STANDBY instruction.

An implementation can export the standby state to an external agent such as a power/energy controller to further regulate the clock or the voltage of the processor core for maximum energy savings. However, if such clock or voltage regulation causes any core/memory state loss, software is responsible to preserve the needed states before the core enters the standby mode. If such state loss has happened, the only sensible way to bring the core into action is through a reset event.

Operations:

Enter_Standby();

Exceptions: None

Privilege level: The behaviors of STANDBY under different processor operating modes are

listed in the following table.

Privilege Level	SubType encoding	SubType behavior
User	0	0
	1	0
	2	0
Superuser	0	0
	1	1
	2	2

Note:

8.1.89. SUB (Subtraction)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		SUB 00001		

Syntax: SUB Rt, Ra, Rb

Purpose: Subtract the content of two registers.

Description: This instruction subtracts the content of Rb from that of Ra and writes the result to Rt.

Operations:

$$Rt = Ra - Rb;$$

Exceptions: None

Privilege level: All

Note:

8.1.90. SUBRI (Subtract Reverse Immediate)

Type: 32-bit Baseline

Format:

	31	30	25	24	20	19	15	14	0
0	SUBRI 101001	Rt		Ra					Imm15s

Syntax: SUBRI Rt, Ra, imm15s

Purpose: Subtract the content of a register from a signed constant.

Description: This instruction subtracts the content of Ra from the sign-extended imm15s and writes the result to Rt.

Operations:

$$Rt = SE(imm15s) - Ra;$$

Exceptions: None

Privilege level: All

Note:

8.1.91. SVA (Set on Overflow Add)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		SVA 11000		

Syntax: SVA Rt, Ra, Rb

Purpose: Generate overflow status on adding the signed contents of two registers.

Description: If an arithmetic overflow is detected in the result of adding the two 32-bit signed values in Ra and Rb, this instruction writes a result of 1 to Rt; otherwise, it writes a result of 0 to Rt.

Operations:

```

value = CONCAT(Ra(31), Ra(31, 0)) + CONCAT(Rb(31), Rb(31, 0));
if (value(32) != value(31)) {
    Rt = 1;
} else {
    Rt = 0;
}
  
```

Exceptions: None

Privilege level: All

Note:

8.1.92. SVS (Set on Overflow Subtract)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		SVS 11001		

Syntax: SVS Rt, Ra, Rb

Purpose: Generate overflow status on subtracting the signed contents of two registers.

Description: If an arithmetic overflow is detected in the result of subtracting the two 32-bit signed values in Ra and Rb, this instruction writes a result of 1 to Rt; otherwise, it writes a result of 0 to Rt.

Operations:

```

value = CONCAT(Ra(31), Ra(31, 0)) - CONCAT(Rb(31), Rb(31, 0));
if (value(32) != value(31)) {
    Rt = 1;
} else {
    Rt = 0;
}
  
```

Exceptions: None

Privilege level: All

Note:

8.1.93. SW (Store Word)

Type: 32-bit Baseline

Format:

SW												
31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		Sv		SW 00001010		

SW.bi

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		Sv		SW.bi 00001110		

Syntax: SW Rt, [Ra + (Rb << sv)]

SW.bi Rt, [Ra], (Rb << sv)

Purpose: Store a 32-bit word from a general register into memory.

Description:

This instruction stores a word from the the general register Rt into memory. There are two different forms to specify the memory address: the regular form uses Ra + (Rb << sv) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + (Rb << sv) value after the memory store operation.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + (Rb << sv);
If (.bi_form) {
  Vaddr = Ra;
} else {
  Vaddr = Addr;
}
if (!Word_Aligned(Vaddr)) {
  Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
  Store_Memory(PAddr, WORD, Attributes, Rt);
  If (.bi_form) { Ra = Addr; }
} else {
  Generate_Exception(Excep_status);
}
  
```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.94. SWI (Store Word Immediate)

Type: 32-bit Baseline

Format:

SWI									
31	30	25	24	20	19	15	14	0	
0	SWI 001010		Rt		Ra			imm15s	

SWI.bi

31	30	25	24	20	19	15	14	0	
0	SWI.bi 001110		Rt		Ra			imm15s	

Syntax: SWI Rt, [Ra + (imm15s << 2)]

SWI.bi Rt, [Ra], (imm15s << 2)

(imm15s is a word offset. In assembly programming, always write a byte offset.)

Purpose: Store a 32-bit word from a general register into memory.

Description:

This instruction stores a word from the general register Rt into memory. There are two different forms to specify the memory address: the regular form uses Ra + SE(imm15s << 2) as its memory address while the .bi form uses Ra. For the .bi form, the Ra register is updated with the Ra + SE(imm15s << 2) value after the memory store operation. Note that imm15s is treated as a signed integer.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra + Sign_Extend(imm15s << 2);
If (.bi_form) {
    Vaddr = Ra;
} else {
    Vaddr = Addr;
}
if (!Word_Aligned(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
    Store_Memory(PAddr, WORD, Attributes, Rt);
    If (.bi_form) { Ra = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```



Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.1.95. SWUP (Store Word with User Privilege Translation)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		Sv		SWUP 00101010		

Syntax: SWUP Rt, [Ra + (Rb << sv)]

Purpose: Store a 32-bit word from a general register into memory with the user mode privilege address translation.

Description: This instruction stores a word from the general register Rt into the memory address Ra + (Rb << sv) with the user mode privilege address translation regardless of the current processor operation mode (i.e. PSW.POM) and the current data address translation state (i.e. PSW.DT). The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Vaddr = Ra + (Rb << sv);
if (!Word_Aligned(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, TRANSLATE);
Excep_status = Page_Exception(Attributes, USER_MODE, STORE);
if (Excep_status == NO_EXCEPTION) {
    Store_Memory(PAddr, WORD, Attributes, Rt);
} else {
    Generate_Exception(Excep_status);
}
  
```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:



8.1.96. SYSCALL (System Call)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	5	4	0
0	MISC 110010	00000			SWID			SYSCALL 01011

Syntax: SYSCALL SWID

Purpose: Generate a System Call exception.

Description: This instruction unconditionally generates a System Call exception and transfers control to the System Call exception handler. Software uses the 15-bit SWID of the instruction as a parameter to distinguish different system call services.

Operations:

Generate_Exception(System_Call);

Exceptions: System Call

Privilege level: All

Note:

8.1.97. TEQZ (Trap if equal 0)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	5	4	0
0	MISC 110010	Ra			SWID		TEQZ 00110	

Syntax: TEQZ Ra, SWID

Purpose: Generate a conditional Trap exception.

Description: This instruction generates a Trap exception and transfers control to the Trap exception handler if the content of Ra is equal to 0. Software uses the 15-bit SWID of the instruction as a parameter to distinguish different trap features and usages.

Operations:

```
If (GR[Ra] == 0)
    Generate_Exception(Trap);
```

Exceptions: Trap

Privilege level: All

Note:

8.1.98. TNEZ (Trap if not equal 0)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	5	4	0
0	MISC 110010	Ra			SWID		TNEZ 00111	

Syntax: TNEZ Ra, SWID

Purpose: Generate a conditional Trap exception.

Description: This instruction generates a Trap exception and transfers control to the Trap exception handler if the content of Ra is not equal to 0. Software uses the 15-bit SWID of the instruction as a parameter to distinguish different trap features and usages.

Operations:

```
If (GR[Ra] != 0)
    Generate_Exception(Trap);
```

Exceptions: Trap

Privilege level: All

Note:

8.1.99. TLBOP (TLB Operation)

Type: 32-bit Baseline

Format:

	31	30	25	24	20	19	15	14	10	9	5	4	0
0	MISC 110010	Rt		Ra		00000		SubType		TLBOP 01110			

Syntax: TLBOP Ra, SubType

TLBOP Rt, Ra, PB (TLB probe operation)

TLBOP FLUA (TLB flush all operation)

Purpose: Perform various operations on processor TLB. This instruction is typically used by software to manage page table entry (PTE) information in TLB.

Description:

This instruction performs TLB control operations based on the SubType field. The definitions and encodings for the SubType field are listed in the following table. Depending on the SubType, different TLBOP instructions have different number of operands from 0 to 2.

Table 63. TLBOP SubType Definitions

SubType	Mnemonics	Operation	Rt?/Ra?
0	TargetRead (TRD)	Read targeted TLB entry	-/Ra
1	TargetWrite (TWR)	Write targeted TLB entry	-/Ra
2	RWrite (RWR)	Write a hardware-determined TLB entry	-/Ra
3	RWriteLock (RWLK)	Write a hardware-determined TLB entry and lock	-/Ra
4	Unlock (UNLK)	Unlock a TLB entry	-/Ra
5	Probe (PB)	Probe TLB entry information	Rt/Ra
6	Invalidate (INV)	Invalidate TLB entries except locked entries	-/Ra
7	FlushAll (FLUA)	Flush all TLB entries except locked entries	-/-

The operations of the cache control instruction can be grouped and described in the following categories:

a. TLB Target Read

Syntax: TLBOP Ra, TargetRead

This operation reads a specified entry in the software-visible portion of the TLB structure and places the read result in the TLB_VPN, TLB_DATA, and TLB_MISC registers. The specified entry is indicated by the Ra register.

The TLB entry number for a non-fully-associative N sets K ways TLB cache is as follows:

31	$\log_2(N*K)$	$\log_2(N*K)-1$	$\log_2(N)$	$\log_2(N)-1$	0
Ignored		Way number		Set number	

The normal instruction sequence of performing the TLB Target Read operation is as follows:

movi Ra, TLB_rd_entry // prepare read entry number
tlbop Ra, TRD // read TLB
dsb // data serialization barrier
mfsr Ry, TLB_VPN // move read result to reg

Important: The TLB_MISC register contains the current process' Context ID and Access Page Size information. Thus, any use of this instruction requires saving/restoring the TLB_MISC register if you want the current process to run correctly immediately after this operation.

b. TLB Target Write

Syntax: TLBOP Ra, TargetWrite

This operation writes a specified entry in the software-visible portion of the TLB structure. The specified entry is indicated by the Ra register. The other write operands are in the

TLB_VPN, TLB_DATA, and TLB_MISC registers.

The TLB entry number for the non-fully-associative N sets K ways TLB cache is as follows:

31	$\log_2(N*K)$	$\log_2(N*K)-1$	$\log_2(N)$	$\log_2(N)-1$	0
Ignored		Way number		Set number	

If the selected target entry is locked, this instruction will overwrite the locked entry and clear the locked flag.

The normal instruction sequence of performing the TLB Target Write operation is as follows:

```

mtsr  Ra, TLB_VPN    // may not needed
mtsr  Rb, TLB_DATA   // prepare PPN, etc.
mtsr  Rc, TLB_MISC   // may not needed
dsb               // data serialization barrier
                  // may not needed (imp-dep)
movi  Rd, TLB_wr_entry // prepare write index
tlbop Rd, TWR        // idx TLB write
isb               // inst serialization barrier

```

c. TLB Random Write (HW-determined way in a set)

Syntax: TLBOP Ra, RWrite

This operation writes a hardware-determined random TLB way in a set determined by the VA (in TLB_VPN) and page size (in TLB_MISC) in the software-visible portion of the TLB structure. The general register Ra contains the data that will be written into the TLB_DATA portion of the TLB structure. The other write operands are in the TLB_VPN and TLB_MISC registers.

If the ways in the specified set are all locked during the write operation of this instruction,

this operation may generate a precise or an imprecise “Data Machine Error” exception. Whether an exception is generated or not is depending on the setting in the TBALCK field of the MMU Control system register (MMU_CTL). Note that the default value of the TBALCK is to generate an exception.

The normal instruction sequence of performing the TLB Random Write operation is as follows:

```
... // TLB fill exception
    // TLB_VPN & TLB_MISC has been preset
... // Preparing PTE address in Rb
lw Ra, [Rb]
tlbop Ra, RWR      // Ra contains PPN, etc.
iretisb           // instruction barrier
```

d. TLB Random Write and Lock

Syntax: TLBOP Ra, RWritelock

Similar to the TLB Random Write operation, this operation writes a hardware-determined random TLB way in a set determined by the VA (in TLB_VPN) and page size (in TLB_MISC) in the software-visible portion of the TLB structure. In addition to the write operation, this instruction also locks the TLB entry.

If the ways in the specified set are all locked during the write operation of this instruction, this operation may generate a precise or an imprecise “Data Machine Error” exception. Whether an exception is generated or not is depending on the setting in the TBALCK field of the MMU Control system register (MMU_CTL). Note that the default value of the TBALCK is to generate the exception.

e. TLB Unlock

Syntax: TLBOP Ra, Unl ock

This operation unlocks a TLB entry if the VA in the general register Ra matches the VPN of a set determined by the VA (in Ra) and page size (in TLB_MISC).

f. TLB Probe

Syntax: TLBOP Rt, Ra, Probe

This operation searches all TLB structures (software-visible and software-invisible) for a specified VA, generates an entry number where the VA matches the VPN in that entry, and writes the search result to the general register Rt. The VA is specified in the general register Ra. The result stored in Rt has the following format:

31	30	29	28	n	n-1	0
NF	HW	SW	Reserved		Entry #	

If the VA can be found in the software-visible part of the TLB, set the “sw” bit. If the VA can be found in the software-invisible part of the TLB, set the “hw” bit. If the VA cannot be found in either the software-visible or software-invisible part of the TLB, set the “nf” bit.

The TLB entry number for the non-fully-associative N sets K ways TLB cache is as follows:

Log2(N*K)-1	log2(N)	Log2(N)-1	0
Way number		Set number	

The normal instruction sequence of performing the TLB probe operation is as follows:

tlbop Rt, Ra, PB // VA is in Ra <Examine> Rt

If this instruction encounters a multiple match condition when searching the TLB, a precise “Data Machine Error” exception will be generated.

g. TLB Invalidate VA

Syntax: TLBOP Ra, Inval i date

This operation flushes the TLB entry that contains the VA in the Ra register and the page size specified in the TLB_MISC register (software-visible and software-invisible) except the locked TLB entries. The match condition also involves the “G” bit of a PTE entry and the CID field of the TLB_MISC register. Their matching logic is as follows:

- If “G” is asserted, CID *is not* part of the match condition.
- If “G” is not asserted, CID *is* part of the match condition.

The normal instruction sequence for this operation is as follows:

```
// prepare VA in Ra
...
tlbop Ra, INV // Invalidate TLB entries containing VA
isb           // inst serialization barrier
```

Note that this TLB invalidate operation may flush more pages than the exact number of pages which contain this VA, up to the entire TLB structure, and the exact behavior is implementation-dependent.

If this instruction encounters a multiple match condition when searching the TLB, it invalidates all matched entries and generates no “Data Machine Error” exception.

h. TLB Invalidate All

Syntax: TLBOP FlushAll

This operation invalidates all TLB entries (software-visible and software-invisible) except the locked TLB entries.

The normal instruction sequence for this operation is as follows:

```

tlbop FLUA    // TLB invalidate All
isb          // Instruction serializati on barrier

```

Operations:

```

If (SubType is not supported)
Exception(Reserved Instruction);
If (Op(SubType) == TargetRead) {
    Entry_Addr = Ra;
    {TLB_VPN, TLB_DATA, TLB_MISC} =
    TLB_Entry_Read(Entry_Addr);
} else if (OP(SubType) == TargetWrite) {
    Entry_Addr = Ra;
    TLB_Entry_Write(Entry_Addr, TLB_VPN, TLB_DATA, TLB_MISC);
} else if (OP(SubType) == RWrite) {
    TLB_Write(TLB_VPN, Ra, TLB_MISC, noLock);
} else if (OP(SubType) == RWriteLock) {
    TLB_Write(TLB_VPN, Ra, TLB_MISC, lock);
} else if (OP(SubType) == Unlock) {
    {found, Entry} = TLB_Search(Ra);
    if (found) {
        TLB_Unlock(Entry);
    }
} else if (OP(SubType) == Probe) {
    {found, Entry} = TLB_Search(Ra);
    Rt = {found, Entry};
} else if (OP(SubType) == Invalidate) {
    {found, Entry} = TLB_Search(Ra);
    if (found) {
        TLB_Invalidate(Entry);
    }
} else if (OP(SubType) == FlushAll) {
    Foreach TLB_Entry {
        if (Is_Not_Locked(TLB_Entry)) {
            TLB_Invalidate(TLB_Entry);
        }
    }
}

```

Exceptions: Privileged Instruction, Imprecise Machine Error

Privilege level: Superuser and above

Note: All non-instruction-fetch-related exceptions generated by a TLBOP instruction should have the INST field of the ITYPE register set to 0.



8.1.100. TRAP (Trap exception)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	5	4	0
0	MISC 110010	00000			SWID		TRAP 00101	

Syntax: TRAP SWID

Purpose: Generate an unconditional Trap exception.

Description: This instruction unconditionally generates a Trap exception and transfers control to the Trap exception handler. Software uses the 15-bit SWID of the instruction as a parameter to distinguish different trap features and usages.

Operations:

```
Generate_Exception(Trap);
```

Exceptions: Trap

Privilege level: All

Note:

8.1.101. WSBH (Word Swap Byte within Halfword)

Type: 32-bit Baseline

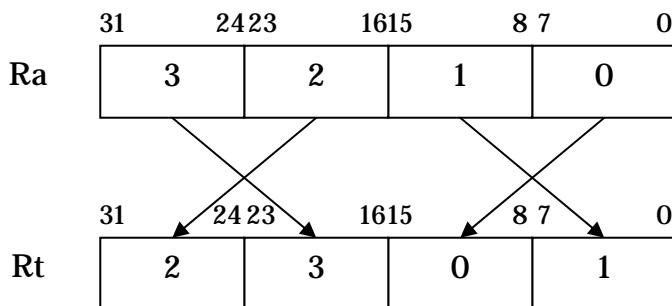
Format:

31	30	25	24	20	19	15	14	5	4	0
0	ALU_1 100000	Rt		Ra		0000000000		WSBH 10100		

Syntax: WSBH Rt, Ra

Purpose: Swap the bytes within each halfword of a register.

Description: This instruction swaps the bytes within each halfword of Ra and writes the result to Rt.



Operations:

Rt = CONCAT(Ra(23, 16), Ra(31, 24), Ra(7, 0), Ra(15, 8));

Exceptions: None

Privilege level: All

Note:

8.1.102. XOR (Bit-wise Logical Exclusive Or)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000	XOR 00011			

Syntax: XOR Rt, Ra, Rb

Purpose: Perform a bit-wise logical Exclusive OR operation on the content of two registers.

Description: This instruction combines the content of Ra with that of Rb using a bit-wise logical exclusive OR operation and writes the result to Rt.

Operations:

$$Rt = Ra \wedge Rb;$$

Exceptions: None

Privilege level: All

Note:

8.1.103. XORI (Exclusive Or Immediate)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	0
0	XORI 101011	Rt		Ra				imm15u

Syntax: XORI Rt, Ra, imm15u

Purpose: Bit-wise exclusive OR of the content of a register with an unsigned constant.

Description: This instruction bit-wise exclusive ORs the content of Ra with the zero-extended imm15u and writes the result to Rt.

Operations:

Rt = Ra ^ ZE(imm15u);

Exceptions: None

Privilege level: All

Note:

8.1.104. ZEB (Zero Extend Byte)

Type: 32-bit Baseline Pseudo OP

Alias of: ANDI Rt, Ra, 0xFF

Syntax: ZEB Rt, Ra

Purpose: Zero-extend the least-significant-byte of a register.

Description: This instruction zero-extends the least-significant-byte of Ra and writes the result to Rt.

Operations:

Rt = ZE(Ra(7, 0));

Exceptions: None

Privilege level: All

Note:

8.1.105. ZEH (Zero Extend Halfword)

Type: 32-bit Baseline

Format:

31	30	25	24	20	19	15	14	5	4	0
0	ALU_1 100000	Rt		Ra		0000000000		ZEH 10011		

Syntax: ZEH Rt, Ra

Purpose: Zero-extend the least-significant-halfword of a register.

Description: This instruction zero-extends the least-significant-halfword of Ra and writes the result to Rt.

Operations:

Rt = ZE(Ra(15, 0));

Exceptions: None

Privilege level: All

Note:

8.2. 32-bit Baseline V1 Optional Instructions

This section includes Baseline V1 optional instructions.

8.2.1. DIV (Unsigned Integer Divide)

Type: 32-bit Baseline Optional

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	000	Dt	0	Ra	Rb	0000	DIV 101111						

Syntax: DIV Dt, Ra, Rb

Purpose: Divide the unsigned integer content of a 32-bit register with that of another register.

Description:

This instruction divides the 32-bit content of Ra with that of Rb, then writes the 32-bit quotient result to Dt.LO register and the 32-bit remainder result to Dt.HI register. The contents of Ra and Rb are treated as unsigned integers.

If the content of Rb is zero and the IDIVZE bit of the INT_MASK register is 1, this instruction generates an Arithmetic exception. The IDIVZE bit of the INT_MASK register enables exception generation for the “Divide-By-Zero” condition.

Operations:

```
If (Rb != 0) {  
    quotient = Floor(CONCAT(1`b0, Ra) / CONCAT(1`b0, Rb));  
    remainder = CONCAT(1`b0, Ra) mod CONCAT(1`b0, Rb);  
    Dt.L0 = quotient;  
    Dt.HI = remainder;  
} else if (!INT_MASK.INTVZ == 0) {  
    Dt.L0 = 0;  
    Dt.HI = 0;  
} else {  
    Generate_Exception(Arithmetic);  
}
```

Exceptions: Arithmetic**Privilege level:** All**Note:**

8.2.2. DIVS (Signed Integer Divide)

Type: 32-bit Baseline Optional

Format:

31	30	25	24	22	21	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	R 000	Dt 0		Ra		Rb		0000		DIVS 101110			

Syntax: DIVS Dt, Ra, Rb

Purpose: Divide the signed integer content of a 32-bit register with that of another register.

Description:

This instruction divides the 32-bit content of Ra with that of Rb, then writes the 32-bit quotient result to Dt.LO register and the 32-bit remainder result to Dt.HI register. The contents of Ra and Rb are treated as signed integers.

If the content of Rb is zero and the IDIVZE bit of the INT_MASK register is 1, this instruction generates an Arithmetic exception. The IDIVZE bit of the INT_MASK register enables exception generation for the “Divide-By-Zero” condition. If the quotient overflows, this instruction always generates an Arithmetic exception. The overflow condition is as follows:

- Positive quotient > 0x7FFF FFFF (When Ra = 0x80000000 and Rb = 0xFFFFFFFF)

Operations:

```
If (Rb != 0) {  
    quotient = Floor(Ra / Rb);  
    if (IsPositive(quotient) && quotient > 0x7FFFFFFF) {  
        Generate_Exception(Arithmeti c);  
    }  
    remainder = Ra mod Rb;  
    Dt.L0 = quotient;  
    Dt.HI = remainder;  
} else if (INT_MASK.IDIVZE == 0) {  
    Dt.L0 = 0;  
    Dt.HI = 0;  
} else {  
    Generate_Exception(Arithmeti c);  
}
```

Exceptions: Arithmetic

Privilege level: All

Note:

8.3. 32-bit Performance Extension Instructions

8.3.1. ABS (Absolute)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	6	5	0
0	ALU_2 100001	Rt	Ra		000000000		ABS 000011			

Syntax: ABS Rt, Ra

Purpose: Get the absolute value of a signed integer in a general register.

Description: This instruction calculates the absolute value of a signed integer stored in Ra and writes the result to Rt. This instruction with the minimum negative integer input of 0x80000000 produces an output of maximum positive integer of 0x7fffffff.

Operations:

```

if (Ra >= 0) {
    Rt = Ra;
} else {
    if (Ra == 0x80000000) {
        Rt = 0x7fffffff;
    } else {
        Rt = -Ra;
    }
}

```

Exceptions: None

Privilege level: All

Note:

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8.3.2. AVE (Average with Rounding)

Type: 32-bit Performance Extension

Format:

	31	30	25	24	20	19	15	14	10	9	6	5	0
0		ALU_2 100001	Rt		Ra		Rb		0000		AVE 000010		

Syntax: AVE Rt, Ra, Rb

Purpose: Calculate the average of the contents of two general registers.

Description: This instruction calculates the average value of two signed integers stored in Ra and Rb, rounds up a half-integer result to the nearest integer, and writes the result to Rt.

Operations:

```
Sum = CONCAT(Ra(31), Ra(31, 0)) + CONCAT(Rb(31), Rb(31, 0)) + 1;
Rt = Sum(32, 1);
```

Exceptions: None

Privilege level: All

Note:

8.3.3. BCLR (Bit Clear)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		imm5u		0000		BCLR 001001		

Syntax: BCLR Rt, Ra, imm5u

Purpose: Clear an individual one bit from the content of a general register

Description: This instruction clears an individual one bit from the value stored in Ra and writes the cleared result to Rt. The bit position is specified by the imm5u value.

Operations:

```
onehot = 1 << imm5u;
Rt = Ra & ~onehot;
```

Exceptions: None

Privilege level: All

Note:

8.3.4. BSET (Bit Set)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		imm5u		0000		BSET 001000		

Syntax: BSET Rt, Ra, imm5u

Purpose: Set an individual one bit from the content of a general register

Description: This instruction sets an individual one bit from the value stored in Ra and writes the modified result to Rt. The bit position is specified by the imm5u value.

Operations:

```
onehot = 1 << imm5u;
Rt = Ra | onehot;
```

Exceptions: None

Privilege level: All

Note:

8.3.5. BTGL (Bit Toggle)

Type: 32-bit Performance Extension

Format:

	31	30	25	24	20	19	15	14	10	9	6	5	0
0		ALU_2 100001	Rt		Ra			imm5u		0000		BTGL 001010	

Syntax: BTGL Rt, Ra, imm5u

Purpose: Toggle an individual one bit from the content of a general register

Description: This instruction toggles an individual one bit from the value stored in Ra and writes the modified result to Rt. The bit position is specified by the imm5u value.

Operations:

```
onehot = 1 << imm5u;
Rt = Ra ^ onehot;
```

Exceptions: None

Privilege level: All

Note:

8.3.6. BTST (Bit Test)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		imm5u		0000		BTST 001011		

Syntax: BTST Rt, Ra, imm5u

Purpose: Test an individual one bit from the content of a general register

Description: This instruction tests an individual one bit from the value stored in Ra. The bit position is specified by the imm5u value. If the bit is set, this instruction writes the result of 1 to Rt; if the bit is cleared, it writes the result of 0 to Rt.

Operations:

```

onehot = 1 << imm5u;
Rt = Ra & onehot;
if (Rt == 0) {
    Rt = 0;
} else {
    Rt = 1;
}
  
```

Exceptions: None

Privilege level: All

Note:

8.3.7. CLIP (Clip Value)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		imm5u		0000		CLIP 000101		

Syntax: CLIP Rt, Ra, imm5u

Purpose: Limit the signed integer of a register into an unsigned range.

Description: This instruction limits the signed integer stored in Ra into an unsigned integer range between $2^{\text{imm5u}-1}$ and 0, then writes the limited result to Rt. For example, if imm5u is 0, the result should be always 0. If Ra is negative, the result is 0 as well.

Operations:

```

if (Ra > 2imm5u-1) {
  Rt = 2imm5u-1;
} else if (Ra < 0) {
  Rt = 0;
} else {
  Rt = Ra;
}

```

Exceptions: None

Privilege level: All

Note:

8.3.8. CLIPS (Clip Value Signed)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt	Ra	imm5u	0000	CLIPS 000100						

Syntax: CLIPS Rt, Ra, imm5u

Purpose: Limit the signed integer of a register into a signed range.

Description: This instruction limits the signed integer stored in Ra into a signed integer range between $2^{\text{imm}5u}-1$ and $-2^{\text{imm}5u}$, then writes the limited result to Rt. For example, if imm5u is 3, the result should be between 7 and -8.

Operations:

```

if (Ra > 2imm5u-1) {
  Rt = 2imm5u-1;
} else if (Ra < -2imm5u) {
  Rt = -2imm5u;
} else {
  Rt = Ra;
}

```

Exceptions: None

Privilege level: All

Note:

8.3.9. CLO (Count Leading Ones)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	6	5	0
0	ALU_2 100001	Rt		Ra			000000000		CLO 000110	

Syntax: CLO Rt, Ra

Purpose: Count the number of successive ones from the most significant bit of a general register.

Description: Starting from the most significant bit (bit 31) of Ra, this instruction counts the number of successive ones and writes the result to Rt. For example, if bit 31 of Ra is 0, the result is 0; if Ra has a value of 0xFFFFFFFF, the result should be 32.

Operations:

```

cnt = 0;
for (i = 31 to 0) {
    if (Ra(i) == 0) {
        break;
    } else {
        cnt = cnt + 1;
    }
}
Rt = cnt;

```

Exceptions: None

Privilege level: All

Note:

8.3.10. CLZ (Count Leading Zeros)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	6	5	0
0	ALU_2 100001	Rt		Ra				000000000	CLZ 000111	

Syntax: CLZ Rt, Ra

Purpose: Count the number of successive zeros from the most significant bit of a general register.

Description: Starting from the most significant bit (bit 31) of Ra, this instruction counts the number of successive zeros and writes the result to Rt. For example, if bit 31 of Ra is 1, the result is 0; if Ra has a value of 0, the result should be 32.

Operations:

```

cnt = 0;
for (i = 31 to 0) {
    if (Ra(i) == 1) {
        break;
    } else {
        cnt = cnt + 1;
    }
}
Rt = cnt;
  
```

Exceptions: None

Privilege level: All

Note:

8.3.11. MAX (Maximum)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		Rb		0000		MAX 000000		

Syntax: MAX Rt, Ra, Rb

Purpose: Get the larger value from the contents of two general registers.

Description: This instruction compares two signed integers stored in Ra and Rb, picks the larger value as the result, and writes the result to Rt.

Operations:

```
if (Ra >= Rb) {
    Rt = Ra;
} else {
    Rt = Rb;
}
```

Exceptions: None

Privilege level: All

Note:

8.3.12. MIN (Minimum)

Type: 32-bit Performance Extension

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		Rb		0000		MIN 000001		

Syntax: MIN Rt, Ra, Rb

Purpose: Get the smaller value from the contents of two general registers.

Description: This instruction compares two signed integers stored in Ra and Rb, picks the smaller value as the result, and writes the result to Rt.

Operations:

```
if (Ra >= Rb) {
    Rt = Rb;
} else {
    Rt = Ra;
}
```

Exceptions: None

Privilege level: All

Note:

8.4. 32-bit Performance Extension Version 2 Instructions

8.4.1. BSE (Bit Stream Extraction)

Type: 32-bit Performance Extension Version 2

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		Rb		0000		BSE 001100		

Syntax: BSE Rt, Ra, Rb

Purpose: Extract a number of bits from a register for bit stream extraction.

Description:

This instruction extracts a number of bits (1 to 32) from register Ra into lower bits of register Rt. If Rb(30), acted as an “underflow” flag, is 0, the instruction fills the non-occupied bits in Rt with 0; otherwise, it does nothing to these bits. The number of bits extracted is specified in Rb(12,8)+1, and the *distance* between Ra(31) and the starting MSB bit position of the extracted bits in Ra is specified in Rb(4,0). After the bits are extracted, Rb(4,0) is incremented to be the *distance* between Ra(31) and the (LSB-1) bit position of the just extracted bits in Ra, making successive BSE extractions flowing from bit-31 to bit-0 in Ra. You can view Rb(4,0) as the number of bits that has already been extracted from Ra starting from Ra(31).

Although the instruction does nothing to the non-occupied bits in Rt if Rb(30) is equal to 1, it updates the Rb(12,8) with Rb(20,16) which should contain the old Rb(12,8) before the underflow condition. (Please see description below.)

The extraction operation with Rb(30) equal to 0 is illustrated in Figure 3 and with Rb(30) equal to 1 is illustrated in Figure 4.

If the sum of Rb(4,0) (=N) and Rb(12,8) (=M) is equal to 31, this instruction will extract all

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

remaining M+1 bits in Ra and set Rb(5,0) to 0x20 in preparation for the next bit stream BSE extraction. In addition, Rb(31), acted as a bitstream register “refill” flag, will be set to 1. This is illustrated in Figure 5. Note that setting Rb(5) to 1 can re-adjust the Rb(4,0) if some bits that have been extracted are re-extracted again.

If the sum of Rb(4,0) (=N) and Rb(12,8) (=M) is greater than 31, this instruction will extract all remaining 32-N bits in Ra to Rt(M,M+N-31) and fill Rt(M+N-32,0) with 0. In this case, it will set Rb(5,0) to 0x20 and Rb(12,8) to M+N-32 in preparation for the next bit stream BSE extraction. In addition, Rb(31), acted as a bitstream register “refill” flag, will be set to 1, and Rb(30), acted as an “underflow” flag, will also be set to 1 indicating that not all required bits are extracted; the old Rb(12,8) will be saved in Rb(20,16) for recovery after the underflow processing. This is illustrated in Figure 6. .

If register number Rt is equal to register number Rb, an UNPREDICTABLE result will be written to the register for this instruction is intened to write different data to different Rt and Rb registers.

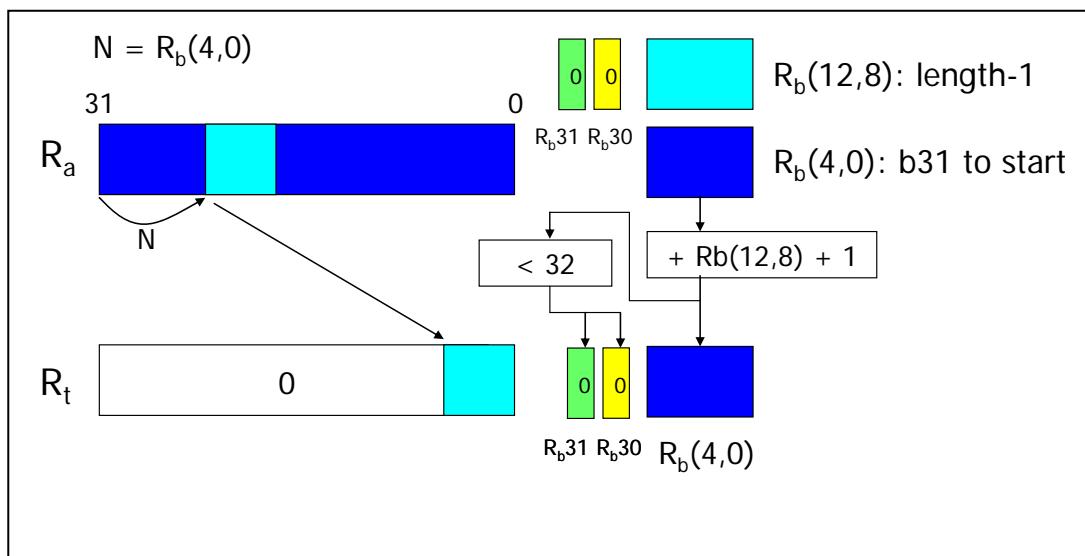
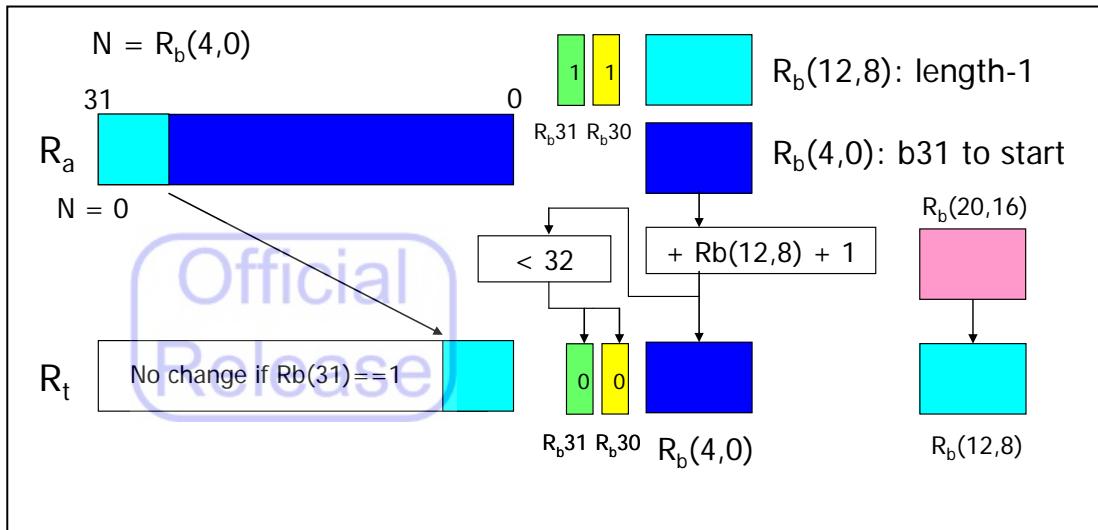
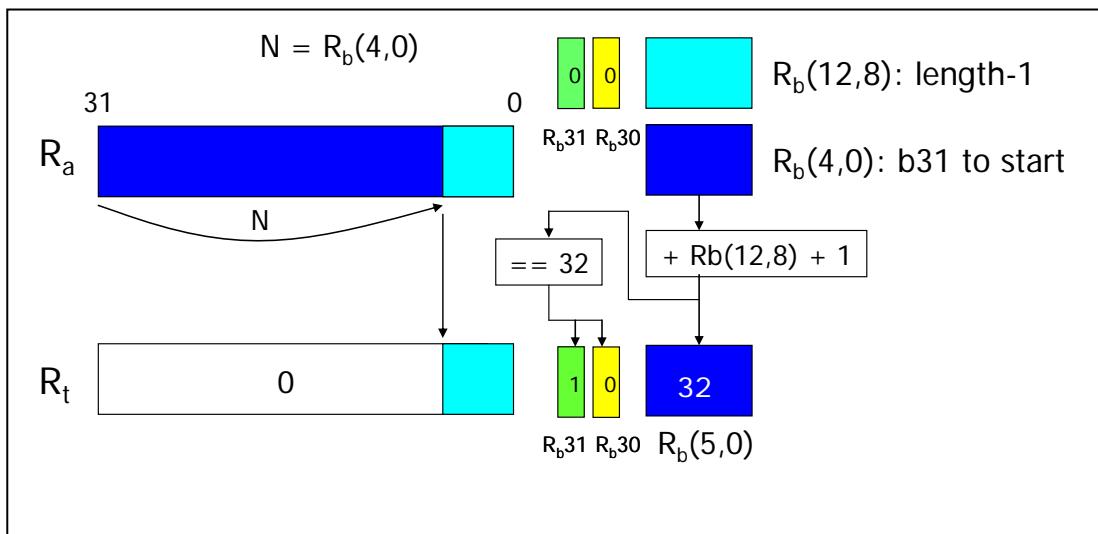


Figure 3. Basic BSE Operation with Rb(30) == 0

Figure 4. Basic BSE Operation with $Rb(30) == 1$ Figure 5. BSE Operation Extracting All Remaining Bits with $Rb(30) == 0$

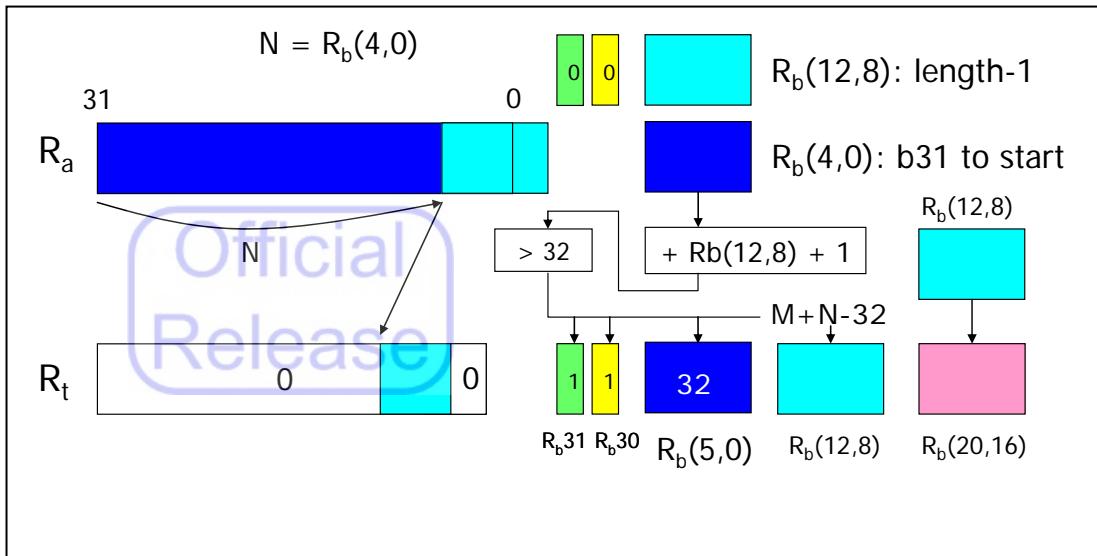


Figure 6. BSE Operation with the “Underflow” Condition with $Rb(30) == 0$

Operations:

```

M = Rb[12: 8];
if (Rb[30] == 0) {
    Len = M + 1;
    Rt[31: Len] = 0;
}
N = Rb[4: 0];
D = M + N;
Rb[7: 5] = 1;
if (31 > D) { // normal condition
    Rb[4: 0] = D + 1;
    Rb[31] = 0;
    Rt[M: 0] = Ra[31-N: 31-N-M];
    if (Rb[30] == 1) {
        Rb[12: 8] = Rb[20: 16];
        Rb[15: 13] = 0;
    }
    Rb[30] = 0;
} else if (31 == D) { // empty condition
    Rb[4: 0] = 0;
    Rb[30] = 0;
    Rb[31] = 1;
    Rt[M: 0] = Ra[M: 0];
}

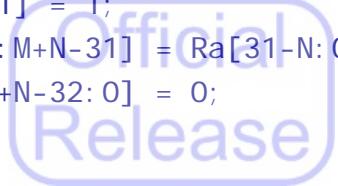
```

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

```

} else if (31 < D) {      // underflow condition
  Rb[20:16] = M;
  Rb[12:8] = D - 32;
  Rb[4:0] = 0;
  Rb[30] = 1;
  Rb[31] = 1;
  Rt[M:M+N-31] = Ra[31-N:0];
  Rt[M+N-32:0] = 0;
}

```



Exceptions: None

Privilege level: All;

Note:

- (1) You can use several baseline instructions to perform a normal multiple-bit extraction like below:
 - R4 is the bit stream register (i.e. Ra).
 - R2 is the number of bits to be extracted (i.e. Rb(12,8)+1).
 - R1 is the distance from MSB of the bit stream register to the starting bit position of the extraction (i.e. Rb(4,0)).

```

sll r3, r4, r1      // squeez out msb
subri r5, r2, 32    // calculate lsb squeeze count
srl r3, r3, r5      // squeeze out lsb
add r1, r1, r2      // move pointer to starting bit

```

So this instruction saves at least 3 instructions when compared to a normal extraction flow.

Following is an example of Huffman decoding loop code using the BSE instruction:

```

LW Ra, [BitStream], 4 # Load 32b from BitStream
L1:   Prepare Rb
L2:   BSE Rt, Ra, Rb
      BGTZ Rb, LOOKUP    # branch if no refill
      LW Ra, [BitStream], 4 # Load 32b from BitStream

```

```
BTST Rk, Rb, 30      # 1 if underflow
BNEZ Rk, L2          # branch if underflow
LOOKUP: Load Huffman table with index Rt
If end of code word {
    Get the decoded data
    Re-adjust Rb(4,0) if necessary
}
Adjust Rb(12,8) if necessary
J L2
```

A semi-transparent watermark in the center of the page. It features the words "Official Release" in a stylized, rounded font. The "O" and "R" are particularly large and prominent. The entire watermark is surrounded by a faint circular border.

- (2) You can use BSP instructions to update the Rb(12,8) and Rb(4,0) values.
- (3) You can use basic addition and subtraction to adjust Rb(4,0) value.

8.4.2. BSP (Bit Stream Packing)

Type: 32-bit Performance Extension Version 2

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		Rb		0000		BSP 001101		

Syntax: BSP Rt, Ra, Rb

Purpose: Insert a number of bits to a register for bit stream packing.

Description:

This instruction inserts a number of bits (1 to 32) from lower bits of register Ra (LSB side) into register Rt. The number of bits inserted is specified in Rb(12,8)+1, and the *distance* from Rt(31) to the MSB bit position of the inserted bits in Rt is specified in Rb(4,0). After the bits are inserted, Rb(4,0) is incremented to be the *distance* from Rt(31) to the (LSB-1) bit position of the just inserted bits in Rt, making successive BSP insertions flowing from bit-31 to bit-0 in Rt. This is illustrated in Figure 7.

If the sum of Rb(4,0) (=N) and Rb(12,8) (=M) is smaller than 31 and the Rb(30) is 1, it indicates that a previous BSP operation is “overflown.” In this case, in addition to the usual packing operation, this instruction updates Rb(12,8) with Rb(20,16) which stores the original packing length before the overflow condition. (See description below.) This is illustrated in Figure 8.

If the sum of Rb(4,0) (=N) and Rb(12,8) (=M) is equal to 31, this instruction will insert all M+1 bits in Ra to Rt, fill up all bits in Rt, and set Rb(5,0) to 0x20 in preparation for the next bit stream BSP packing operation. In addition, Rb(31), acted as a bitstream register “output” flag, will be set to 1. This is illustrated in Figure 9.

If the sum of Rb(4,0) (=N) and Rb(12,8) (=M) is greater than 31, the remaining number of bits in Rt is not enough to accommodate the number of bits needed for packing in Ra. In this case,

this instruction will only insert 32-N bits from Ra to Rt and leave the remaining M+N-31 bits in Ra to be inserted in the next BSP instruction. It will set Rb(5,0) to 0x20 and Rb(12,8) to M+N-32 in preparation for the next bit stream BSP packing operation. In addition, Rb(31), acted as a bitstream register “output” flag, will be set to 1, and R(30), acted as an “overflow” flag, will also be set to 1 indicating that not all required bits are packed; the old Rb(12,8) will be saved in Rb(20,16) for recovery after the overflow condition has been resolved. This is illustrated in Figure 10.

If register number Rt is equal to register number Rb, an UNPREDICTABLE result will be written to the register since this instruction is intended to write different data to different Rt and Rb registers.

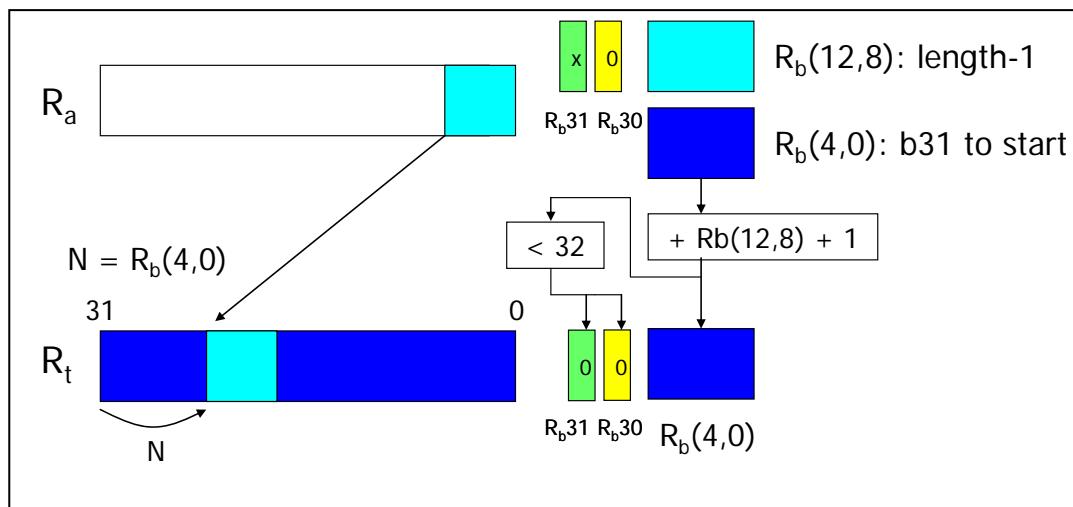
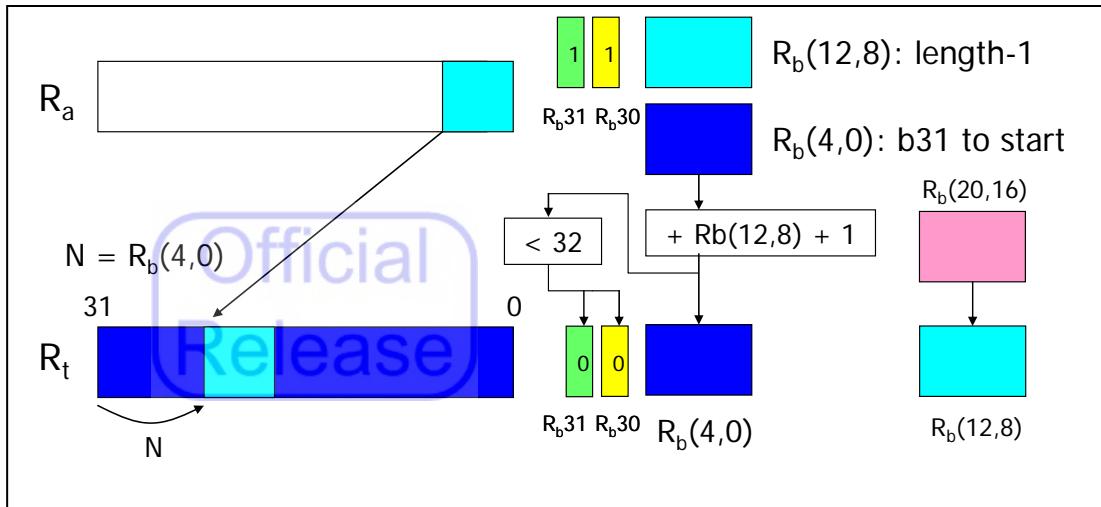
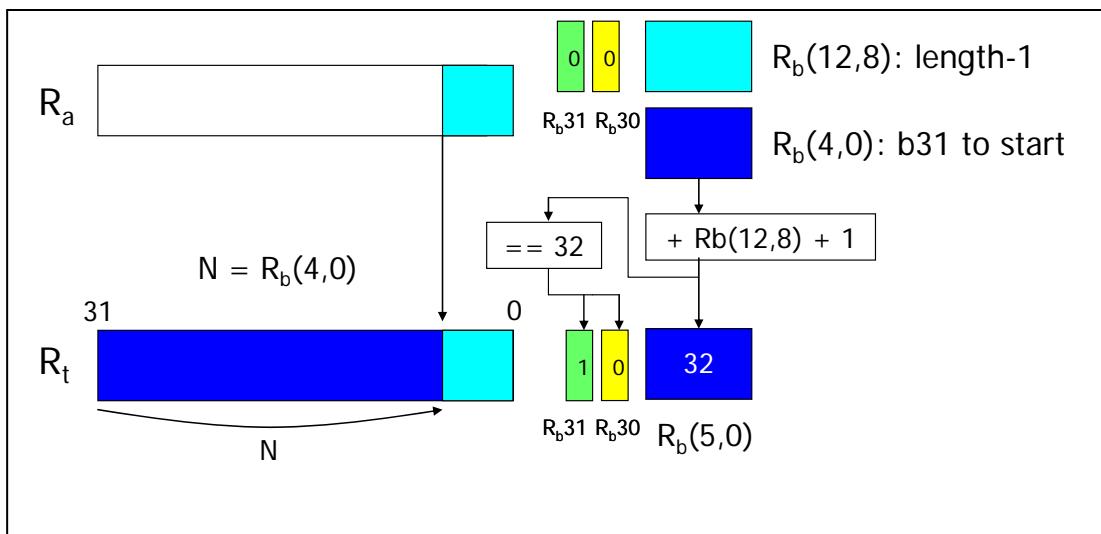


Figure 7. Basic BSP Operation

Figure 8. BSP Operation with $Rb(30) == 1$ Figure 9. BSP Operation Filling up R_t .

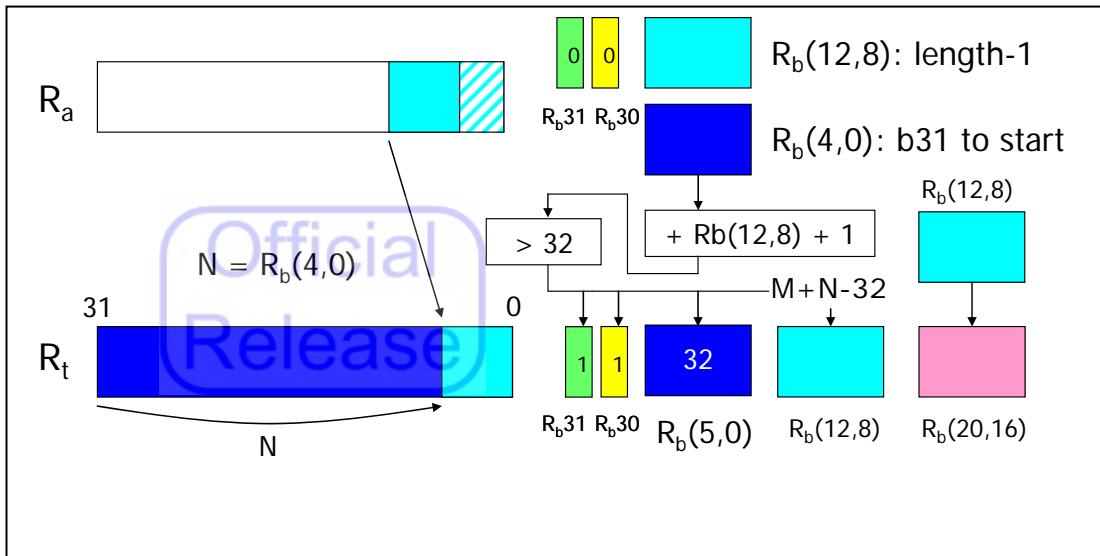


Figure 10. BSP Operation with the "Overflow" Condition.

Operations:

```

M = Rb[12: 8];
N = Rb[4: 0];
D = M + N;
Rb[7: 5] = 1;
if (31 > D) { // normal condition
    Rb[4: 0] = D + 1;
    Rb[31] = 0;
    Rt[31-N: 31-N-M] = Ra[M: 0];
    if (Rb[30] == 1) {
        Rb[12: 8] = Rb[20: 16];
        Rb[15: 13] = 0;
    }
    Rb[30] = 0;
} else if (31 == D) { // full condition
    Rb[4: 0] = 0;
    Rb[30] = 0;
    Rb[31] = 1;
    Rt[M: 0] = Ra[M: 0];
} else if (31 < D) { // overflow condition
    Rb[20: 16] = M;
    Rb[12: 8] = D - 32;
    Rb[4: 0] = 0;
}

```

```

Rb[30] = 1;
Rb[31] = 1;
Rt[31-N:0] = Ra[M:M+N-31];
}
  
```

Exceptions: None



Privilege level: All

Note:

- (1) You can use several baseline instructions to perform a normal multiple-bit packing operation like below:
 - R4 contains the bits for packing (i.e. Ra).
 - R5 is the bit stream register (i.e. Rt)
 - R2 contains the number of bits for packing (i.e. Rb(12,8)+1).
 - R1 is the distance from MSB of the bit stream register to the starting bit position of the packing operation (i.e. Rb(4,0)).

```

subri r3, r2, 32
sll r3, r4, r3
srl r3, r3, r1
or r5, r5, r3
  
```

So this instruction saves at least 3 instructions when compared to a normal bit-packing flow.

Following is an example of Huffman encoding loop code using the BSP instruction:

```

L1:    Prepare Rb
        Prepare code word in Ra
L2:    BSP Rt, Ra, Rb
        BGTZ Rb, L1      # branch if no output
        SW Rt, [BitStream], 4 # store 32b to BitStream
        BTST Rt, Rb, 30    # 1 if overflow
        BNEZ Rt, L2
        J L1
  
```

- (2) You can use BSP instructions to update the Rb(12,8) and Rb(4,0) values.
- (3) You can use basic addition and subtraction to adjust Rb(4,0) value.

8.4.3. PBSAD (Parallel Byte Sum of Absolute Difference)

Type: 32-bit Performance Extension Version 2

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	SIMD 111000	Rt		Ra		Rb		00000		PBSAD 00000		

Syntax: PBSAD Rt, Ra, Rb

Purpose: Calculate the sum of absolute difference of four unsigned 8-bit data elements.

Description: This instruction subtracts the four un-signed 8-bit elements of Ra from those of Rb. It adds the absolute value of each difference together and writes the result to Rt.

Operations:

```

a(7, 0) = ABS(Ra(7, 0) - Rb(7, 0));
b(7, 0) = ABS(Ra(15, 8) - Rb(15, 8));
c(7, 0) = ABS(Ra(23, 16) - Rb(23, 16));
d(7, 0) = ABS(Ra(31, 24) - Rb(31, 24));
Rt = a(7, 0) + b(7, 0) + c(7, 0) + d(7, 0);
  
```

Exceptions: None

Privilege level: All

Note:

8.4.4. PBSADA (Parallel Byte Sum of Absolute Difference Accum)

Type: 32-bit Performance Extension Version 2

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	SIMD 111000	Rt		Ra		Rb		00000	PBSADA 00001			

Syntax: PBSADA Rt, Ra, Rb

Purpose: Calculate the sum of absolute difference of four unsigned 8-bit data elements and accumulate it into a register.

Description: This instruction subtracts the four un-signed 8-bit elements of Ra from those of Rb. It adds the absolute value of each difference together along with the content of Rt and writes the accumulated result back to Rt.

Operations:

```

a(7, 0) = ABS(Ra(7, 0) - Rb(7, 0));
b(7, 0) = ABS(Ra(15, 8) - Rb(15, 8));
c(7, 0) = ABS(Ra(23, 16) - Rb(23, 16));
d(7, 0) = ABS(Ra(31, 24) - Rb(31, 24));
Rt = Rt + a(7, 0) + b(7, 0) + c(7, 0) + d(7, 0);

```

Exceptions: None

Privilege level: All

Note:

8.5. 32-bit STRING Extension Instructions

8.5.1. FFB (Find First Byte)

Type: 32-bit STRING Extension

Format:

31	30	25	24	20	19	15	14	10	9	7	6	0
0	ALU_2 100001	Rt		Ra		Rb		000		FFB 0001110		

Syntax: FFB Rt, Ra, Rb

Purpose: Find the first byte in a word that matches a value in a register.

Description: This instruction matches each byte in Ra with the value in Rb(7,0). If any matching byte is found, this instruction writes a non-zero position indication of the first matching byte based on the current data endian (PSW.BE) mode to Rt; if no matching byte is found, it writes a zero to Rt.

Operations:

```

Match1 = (Ra(7, 0) == Rb(7, 0));
Match2 = (Ra(15, 8) == Rb(7, 0));
Match3 = (Ra(23, 16) == Rb(7, 0));
Match4 = (Ra(31, 24) == Rb(7, 0));
found = Match1 || Match2 || Match3 || Match4;
if (!found) {
  Rt = 0;
} else if (PSW.BE == 0) {
  if (Match1) {
    Rt = -4;
  } else if (Match2) {
    Rt = -3;
  } else if (Match3) {
    Rt = -2;
  }
}

```

```
Rt = -2;  
} else {  
    Rt = -1;  
}  
} else { // PSW.BE == 1  
    If (Match4) {  
        Rt = -4;  
    } else if (Match3) {  
        Rt = -3;  
    } else if (Match2) {  
        Rt = -2;  
    } else {  
        Rt = -1;  
    }  
}
```



Exceptions: None

Privilege level: All

Note:

8.5.2. FFBI (Find First Byte Immediate)

Type: 32-bit STRING Extension

Format:

31	30	25	24	20	19	15	14	7	6	0
0	ALU_2 100001	Rt	Ra	imm8	FFBI 1001110					

Syntax: FFBI Rt, Ra, imm8

Purpose: Find the first byte in a word that matches a constant.

Description: This instruction matches each byte in Ra with the constant imm8. If any matching byte is found, the instruction writes a non-zero position indication of the first matching byte based on the current data endian (PSW.BE) mode to Rt; if no matching byte is found, it writes a zero to Rt.

Operations:

```

Match1 = (Ra(7, 0) == i mm8);
Match2 = (Ra(15, 8) == i mm8);
Match3 = (Ra(23, 16) == i mm8);
Match4 = (Ra(31, 24) == i mm8);
found = Match1 || Match2 || Match3 || Match4;
If (!found) {
  Rt = 0;
} else if (PSW.BE == 0) {
  If (Match1) {
    Rt = -4;
  } else if (Match2) {
    Rt = -3;
  } else if (Match3) {
    Rt = -2;
  } else {
    Rt = -1;
}

```

```
    }
} else { // PSW.BE == 1
    If (Match4) {
        Rt = -4;
    } else if (Match3) {
        Rt = -3;
    } else if (Match2) {
        Rt = -2;
    } else {
        Rt = -1;
    }
}
```



Exceptions: None

Privilege level: All

Note:

8.5.3. FFMISM (Find First Mis-match)

Type: 32-bit STRING Extension

Format:

31	30	25	24	20	19	15	14	10	9	7	6	0
0	ALU_2 100001	Rt	Release	Ra		Rb		000	FFMISM 0001111			

Syntax: FFMISM Rt, Ra, Rb

Purpose: Find the first byte in a word that fails a 32-bit word comparison.

Description: This instruction matches each byte in Ra with each corresponding byte in Rb. If any mis-matching byte is found, the instruction writes a non-zero position indication of the first mis-matching byte based on the current data endian (PSW.BE) mode to Rt; if no mis-matching byte is found, it writes a zero to Rt.

Operations:

```

Mi sM1 = (Ra(7, 0) != Rb(7, 0));
Mi sM2 = (Ra(15, 8) != Rb(15, 8));
Mi sM3 = (Ra(23, 16) != Rb(23, 16));
Mi sM4 = (Ra(31, 24) != Rb(31, 24));
found = Mi sM1 || Mi sM2 || Mi sM3 || Mi sM4;
If (!found) {
  Rt = 0;
} else if (PSW.BE == 0) {
  If (Mi sM1) {
    Rt = -4;
  } else if (Mi sM2) {
    Rt = -3;
  } else if (Mi sM3) {
    Rt = -2;
  } else {
    Rt = -1;
}

```

```
    }
} else { // PSW.BE == 1
    If (MisM4) {
        Rt = -4;
    } else if (MisM3) {
        Rt = -3;
    } else if (MisM2) {
        Rt = -2;
    } else {
        Rt = -1;
    }
}
```



Exceptions: None

Privilege level: All

Note:

8.5.4. FLMISM (Find Last Mis-match)

Type: 32-bit STRING Extension

Format:

	31	30	25	24	20	19	15	14	10	9	7	6	0
0		ALU_2 100001	Rt		Ra		Rb		000	FLMISM 1001111			

Syntax: FLMISM Rt, Ra, Rb

Purpose: Find the last byte in a word that fails a 32-bit word comparison.

Description: This instruction matches each byte in Ra with each corresponding byte in Rb. If any mis-matching byte is found, the instruction writes a non-zero position indication of the last mis-matching byte based on the current data endian (PSW.BE) mode to Rt; if no mis-matching byte is found, it writes a zero to Rt.

Operations:

```

Mi sM1 = (Ra(7, 0) != Rb(7, 0));
Mi sM2 = (Ra(15, 8) != Rb(15, 8));
Mi sM3 = (Ra(23, 16) != Rb(23, 16));
Mi sM4 = (Ra(31, 24) != Rb(31, 24));
found = Mi sM1 || Mi sM2 || Mi sM3 || Mi sM4;
If (!found) {
  Rt = 0;
} else if (PSW.BE == 0) {
  If (Mi sM4) {
    Rt = -1;
  } else if (Mi sM3) {
    Rt = -2;
  } else if (Mi sM2) {
    Rt = -3;
  } else {
    Rt = -4;
}

```

```
    }
} else { // PSW.BE == 1
    If (MisM1) {
        Rt = -1;
    } else if (MisM2) {
        Rt = -2;
    } else if (MisM3) {
        Rt = -3;
    } else {
        Rt = -4;
    }
}
```



Exceptions: None

Privilege level: All

Note:

8.6. 16-bit Baseline V1 Instructions

Definitions of register number encoding conversion functions:

- $3T5(x): y(4:0) = 3T5(x(2:0))$

x	0	1	2	3	4	5	6	7
y	0	1	2	3	4	5	6	7

- $4T5(x): y(4:0) = 4T5(x(3:0))$

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
y	0	1	2	3	4	5	6	7	8	9	10	11	16	17	18	19

8.6.1. ADD (Add Register)

Type: 16-bit Baseline

Format:

ADD333								
15	14	9	8	6	5	3	2	0
1		ADD333 001100	Rt3		Ra3		Rb3	

ADD45

15	14	9	8	5	4	0
1		ADD45 000100	Rt4		Rb5	

Syntax: ADD333 Rt3, Ra3, Rb3

ADD45 Rt4, Rb5

32-bit Equivalent: ADD 3T5(Rt3), 3T5(Ra3), 3T5(Rb3) // ADD333

ADD 4T5(Rt4), 4T5(Rt4), Rb5 // ADD45

Purpose: Add the contents of two registers.

Description: For ADD333, the instruction adds the contents of Ra3 and Rb3, then writes the result to Rt3; for ADD45, it adds the contents of Rt4 and Rb5, then writes the result to the source register Rt4.

Operations:

Rt3 = Ra3 + Rb3; // ADD333

Rt4 = Rt4 + Rb5; // ADD45

Exceptions: None

Privilege level: All

Note:



8.6.2. ADDI (Add Immediate)

Type: 16-bit Baseline

Format:

ADDI333							
15	14	9	8	6	5	3	2 0
1		ADDI333 001110		Rt3		Ra3	imm3u

ADDI45

15	14	9	8	5	4	0
1		ADDI45 000110		Rt4		imm5u

Syntax: ADDI333 Rt3, Ra3, imm3u

ADDI45 Rt4, imm5u

32-bit Equivalent: ADDI 3T5(Rt3), 3T5(Ra3), ZE(imm3u) // ADDI333

ADDI 4T5(Rt4), 4T5(Rt4), ZE(imm5u) // ADDI45

Purpose: Add a zero-extended immediate to the content of a register.

Description: For ADDI333, this instruction adds the zero-extended 3-bit immediate “imm3u” to the content of Ra3, then writes the result to Rt3; for ADDI45, it adds the zero-extended 5-bit immediate to the content of Rt4. then writes the result to the source register Rt4.

Operations:

```
Rt3 = Ra3 + ZE(imm3u); // ADDI 333
Rt4 = Rt4 + ZE(imm5u); // ADDI 45
```

Exceptions: None

Privilege level: All

Note:



8.6.3. BEQS38 (Branch on Equal Implied R5)

Type: 16-bit Baseline

Format:

15	14	11	10	8	7	0
1	BEQS38 1010		Rt3 (#Rt3 != 5)		imm8s	

Syntax: BEQS38 Rt3, imm8s

32-bit Equivalent: BEQ 3T5(Rt3), R5, SE(imm8s)

(next sequential PC = PC + 2)

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with that of the implied register R5.

Description: If the content of the implied register R5 is equal to that of Rt3 (#Rt3 != 5), this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm8s << 1) value. The branch range is ± 256 bytes.

Operations:

```
TAddr = PC + Sign_Extend(imm8s << 1);
If (R5 == Rt3) {
  PC = TAddr;
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.6.4. BEQZ38 (Branch on Equal Zero)

Type: 16-bit Baseline

Format:

15	14	11	10	8	7	0
1	BEQZ38 1000		Rt3		imm8s	

Syntax: BEQZ38 Rt3, imm8s

32-bit Equivalent: BEQZ 3T5(Rt3), SE(imm8s)

(next sequential PC = PC + 2)

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt3 is equal to zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm8s << 1) value. The branch range is ± 256 bytes.

Operations:

```
TAddr = PC + Sign_Extend(imm8s << 1);
If (Rt3 == 0) {
  PC = TAddr;
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.6.5. BEQZS8 (Branch on Equal Zero Implied R15)

Type: 16-bit Baseline

Format:

15	14	8	7	0
1	 BEQZS8 1101000		imm8s	

Syntax: BEQZS8 imm8s

32-bit Equivalent: BEQZ R15, SE(imm8s)

(next sequential PC = PC + 2)

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of R15 with zero.

Description: If the content of R15 is equal to zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm8s << 1) value. The branch range is ± 256 bytes.

Operations:

```

TAddr = PC + Sign_Extend(imm8s << 1);
If (R15 == 0) {
  PC = TAddr;
}
  
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.6.6. BNES38 (Branch on Not Equal Implied R5)

Type: 16-bit Baseline

Format:

15	14	11	10	8	7	0
1	BNES38 1011		Rt3 (#Rt3 != 5)		imm8s	

Syntax: BNES38 Rt3, imm8s

32-bit Equivalent: BNE 3T5(Rt3), R5, SE(imm8s)

(next sequential PC = PC + 2)

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with that of the implied register R5.

Description: If the content of the implied register R5 is not equal to that of Rt3 (#Rt3 != 5), this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm8s << 1) value. The branch range is ± 256 bytes.

Operations:

```

TAddr = PC + Sign_Extend(imm8s << 1);
If (R5 != Rt3) {
  PC = TAddr;
}
  
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.6.7. BNEZ38 (Branch on Not Equal Zero)

Type: 16-bit Baseline

Format:

15	14	11	10	8	7	0
1	BNEZ38 1001		Rt3		imm8s	

Syntax: BNEZ38 Rt3, imm8s

32-bit Equivalent: BNEZ 3T5(Rt3), SE(imm8s)

(next sequential PC = PC + 2)

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with zero.

Description: If the content of Rt3 is not equal to zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm8s << 1) value. The branch range is ± 256 bytes.

Operations:

```

TAddr = PC + Sign_Extend(imm8s << 1);
If (Rt3 != 0) {
    PC = TAddr;
}

```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.6.8. BNEZS8 (Branch on Not Equal Zero Implied R15)

Type: 16-bit Baseline

Format:

15	14	8	7	0
1	BNEZS8 1101001			imm8s

Syntax: BNEZS8 imm8s

32-bit Equivalent: BNEZ R15, SE(imm8s)
(next sequential PC = PC + 2)

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of R15 with zero.

Description: If the content of R15 is not equal to zero, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm8s << 1) value. The branch range is ± 256 bytes.

Operations:

```
TAddr = PC + Sign_Extend(imm8s << 1);
If (R15 != 0) {
    PC = TAddr;
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.6.9. BREAK16 (Breakpoint)

Type: 16-bit Baseline

Format:

15	14	9	8	0
1	BREAK16 110101			SWID9

Syntax: BREAK16 SWID9

32-bit Equivalent: BREAK ZE(SWID9)

Purpose: Generate a Breakpoint exception.

Description: This instruction unconditionally generates a Breakpoint exception and transfers control to the Breakpoint exception handler. Software uses the 9-bit SWID of the instruction as a parameter to distinguish different breakpoint features and usages.

Operations:

```
Generate_Exception(Breakpoint);
```

Exceptions: Breakpoint

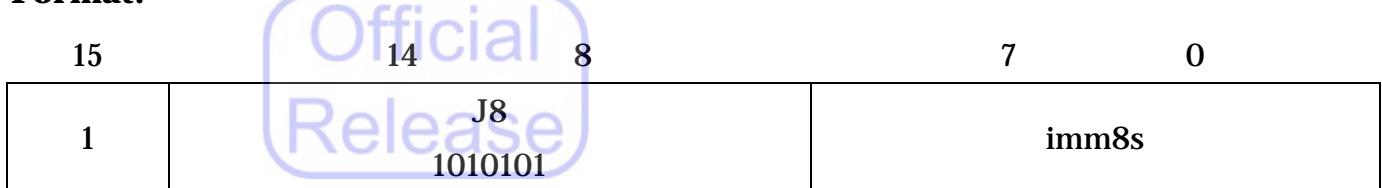
Privilege level: All

Note:

8.6.10. J8 (Jump Immediate)

Type: 16-bit Baseline

Format:



Syntax: J8 imm8s

32-bit Equivalent: J SE(imm8s)

Purpose: Perform unconditional PC-relative branching.

Description: This instruction jumps unconditionally to the target address calculated by adding the current instruction address and the sign-extended (imm8s << 1) value. The branch range is ± 256 bytes.

Operations:

```
TAddr = PC + Sign_Extend(imm8s << 1);
PC = TAddr;
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.6.11. JR5 (Jump Register)

Type: 16-bit Baseline

Format:

15	14	5	4	0
1	JR5 1011101000			Rb5

Syntax: JR5 Rb5

32-bit Equivalent: JR Rb5

Purpose: Unconditionally branch to an address stored in a general register.

Description: This instruction jumps unconditionally to the target address stored in the register Rb5.

Operations:

```
TAddr = Rb5;
PC = TAddr;
```

Exceptions: None

Privilege level: All

Note:

8.6.12. JRAL5 (Jump Register and Link)

Type: 16-bit Baseline

Format:

15	14	5	4	0
1	JRAL5 1011101001			Rb5

Syntax: JRAL5 Rb5

32-bit Equivalent: JRAL Rb5

$$(R30 = PC + 2)$$

Purpose: Unconditionally branch to a function entry point.

Description: This instruction jumps unconditionally to the target address stored in the register Rb5 and writes the next sequential address ($PC + 2$) of the current instruction to R30.

Operations:

```
TAddr = Rb5;
R30 = PC + 2;
PC = TAddr;
```

Exceptions: None

Privilege level: All

Note:

8.6.13. LBI333 (Load Byte Immediate Unsigned)

Type: 16-bit Baseline

Format:

15	14	9	8	6	5	3	2	0
1	LBI333 010011		Rt3		Ra3		imm3u	

Syntax: LBI333 Rt3, [Ra3, imm3u]

32-bit Equivalent: LBI 3T5(Rt3), [3T5(Ra3), ZE(imm3u)]

Purpose: Load a zero-extended 8-bit byte from memory into a general register.

Description: This instruction loads a byte from the memory address specified by adding the content of Ra3 and the zero-extended imm3u value. It zero-extends the loaded byte to the width of the general register and then writes the result to Rt3.

Operations:

```

VAddr = Ra3 + Zero_Extend(imm3u);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Bdata(7,0) = Load_Memory(PAddr, BYTE, Attributes);
    Rt3 = Zero_Extend(Bdata(7,0));
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

8.6.14. LHI333 (Load Halfword Immediate Unsigned)

Type: 16-bit Baseline

Format:

15	14	9	8	6	5	3	2	0
1	LHI333 010010	Rt3		Ra3		imm3u		

Syntax: LHI333 Rt3, [Ra3 + (imm3u << 1)]

32-bit Equivalent: LHI 3T5(Rt3), [3T5(Ra3) + ZE((imm3u << 1))]

(imm3u is a halfword offset. In assembly programming, always write a byte offset.)

Purpose: Load a zero-extended 16-bit halfword from memory into a general register.

Description:

This instruction loads a halfword from the memory address specified by adding the content of Ra3 and the zero-extended (imm3u << 1) value. It zero-extends the loaded halfword to the width of the general register and then writes the result into Rt3. Note that imm3u is a halfword-aligned offset.

The memory address has to be halfword-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

VAddr = Ra3 + Zero_Extend((imm3u << 1));
if (!Halfword_Alignment(Vaddr)) {
  Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
  Hdata(15, 0) = Load_Memory(PAddr, HALFWORD, Attributes);
  Rt3 = Zero_Extend(Hdata(15, 0));
} else {
  Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.6.15. LWI333 (Load Word Immediate)

Type: 16-bit Baseline

Format:

LWI333								
15	14	9	8	6	5	3	2	0
1	LWI333 010000		Rt3		Ra3		imm3u	

LWI333.bi

15	14	9	8	6	5	3	2	0
1	LWI333.bi 010001		Rt3		Ra3		imm3u	

Syntax: LWI333 Rt3, [Ra3 + (imm3u << 2)]

 LWI333.bi Rt3, [Ra3], (imm3u << 2)

32-bit Equivalent: LWI 3T5(Rt3), [3T5(Ra3) + ZE(imm3u << 2)]

 LWI.bi 3T5(Rt3), [3T5(Ra3)], ZE(imm3u << 2)

(imm3u is a word offset. In assembly programming, always write a byte offset.)

Purpose: Load a 32-bit word from memory into a general register.

Description:

This instruction loads a word from memory into the general register Rt3. There are two different forms to specify the memory address: the regular form uses Ra3 + ZE(imm3u << 2) as its memory address while the .bi form uses Ra3. For the .bi form, the Ra3 register is updated with the Ra3 + ZE(imm3u << 2) value after the memory load operation.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra3 + Zero_Extend((imm3u << 2));
If (.bi_form) {
  Vaddr = Ra3;
} else {
  Vaddr = Addr;
}
if (!Word_Aligned(Vaddr)) {
  Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
  Wdata(31,0) = Load_Memory(PAddr, WORD, Attributes);
  Rt3 = Wdata(31,0);
  If (.bi_form) { Ra3 = Addr; }
} else {
  Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.6.16. LWI37 (Load Word Immediate with Implied FP)

Type: 16-bit Baseline

Format:

15	14	11	10	8	7	6	0
1	XWI37 0111	Rt3		LWI37 0		imm7u	

Syntax: LWI37 Rt3, [FP + (imm7u << 2)]

32-bit Equivalent: LWI 3T5(Rt3), [FP + ZE(imm7u << 2)]

(imm7u is a word offset. In assembly programming, always write a byte offset.)

Purpose: Load a 32-bit word from memory into a general register.

Description:

This instruction loads a word from memory into the general register Rt3. The memory address is specified by adding the implied FP (i.e. R28) register and the zero-extended (imm7u << 2) value. Note that the imm7u is a word-aligned offset.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

VAddr = FP (i.e. R28) + Zero_Extend(imm7u << 2);
if (!Word_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
if (Excep_status == NO_EXCEPTION) {
    Wdata(31, 0) = Load_Memory(PAddr, WORD, Attributes);
    Rt3 = Wdata(31, 0);
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.6.17. LWI450 (Load Word Immediate)

Type: 16-bit Baseline

Format:

15	14	9	8	5	4	0
1	LWI450 011010		Rt4		Ra5	

Syntax: LWI450 Rt4, [Ra5]

32-bit Equivalent: LWI 4T5(Rt4), [Ra5 + 0]

Purpose: Load a 32-bit word from memory into a general register.

Description:

This instruction loads a word from memory into the general register Rt4. The memory address is specified in Ra5.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

VAddr = Ra5;
if (!Word_Alignment(Vaddr)) {
  Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
if (Excep_status == NO_EXCEPTION) {
  Wdata(31, 0) = Load_Memory(PAddr, WORD, Attributes);
  Rt4 = Wdata(31, 0);
} else {
  Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.6.18. MOV55 (Move Register)

Type: 16-bit Baseline

Format:

15	14	10	9	5	4	0
1	MOV55 00000		Rt5		Ra5	

Syntax: MOV55 Rt5, Ra5

32-bit Equivalent: ADDI/ORI Rt5, Ra5, 0

Purpose: Move contents between general-purpose registers.

Description: This instruction moves the content of Ra5 into Rt5.

Operations:

Rt5 = Ra5;

Exceptions: None

Privilege level: All

Note:

8.6.19. MOVI55 (Move Immediate)

Type: 16-bit Baseline

Format:

15	14	10	9	5	4	0
1	MOVI55 00001		Rt5		imm5s	

Syntax: MOVI55 Rt5, imm5s

32-bit Equivalent: MOVI Rt5, SE(imm5s)

Purpose: Move a sign-extended immediate into a general-purpose register.

Description: This instruction moves the sign-extended 5-bit immediate “imm5s” into Rt5.

Operations:

```
Rt5 = Sign_Extend(imm5s);
```

Exceptions: None

Privilege level: All

Note:

8.6.20. NOP16 (No Operation)

Type: 16-bit Baseline

Format:

15	14	9	8	5	4	0
1	SRLI45 001001		NOP16 0000		NOP16 00000	

Syntax: NOP16

(SRLI45 R0, 0)

32-bit Equivalent: NOP

(SRL R0, R0, 0)

Purpose: Perform no operation. It may align program code for any specific purpose.

Description: This instruction is aliased to “SRLI45 R0, 0” instruction in hardware.

Operations:

None

Exceptions: None

Privilege level: All

Note:

8.6.21. RET5 (Return from Register)

Type: 16-bit Baseline

Format:

15	14	5	4	0
1	RET5 1011101100			Rb5

Syntax: RET5 Rb5

32-bit Equivalent: RET Rb5

Purpose: Make an unconditional function call return to an address stored in a general register.

Description: This instruction jumps unconditionally to the target address stored in the register Rb5. Note that the architecture behavior of this instruction is the same as that of the JR5 instruction. Even so, software uses this instruction instead of JR5 for function call return. This facilitates software to distinguish the two different usages and is helpful in call stack backtracing applications. Distinguishing a function return jump from a regular jump also helps implementation performance (e.g. return address prediction).

Operations:

```
TAddr = Rb5;
PC = TAddr;
```

Exceptions: None

Privilege level: All

Note:

8.6.22. SBI333 (Store Byte Immediate)

Type: 16-bit Baseline

Format:

15	14	9	8	6	5	3	2	0
1	SBI333 010111	Rt3		Ra3		imm3u		

Syntax: SBI333 Rt3, [Ra3 + imm3u]

32-bit Equivalent: SBI 3T5(Rt3), [3T5(Ra3) + ZE(imm3u)]

Purpose: Store an 8-bit byte from a general register into a memory location.

Description: This instruction stores the least-significant 8-bit byte in the general register Rt3 to the memory location whose address is specified by adding the content of Ra3 and the zero-extended imm3u value.

Operations:

```

VAddr = Ra3 + Zero_Extend(imm3u);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_Status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_Status == NO_EXCEPTION) {
    Store_Memory(PAddr, BYTE, Attributes, Rt3(7,0));
} else {
    Generate_Exception(Excep_Status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8.6.23. SEB33 (Sign Extend Byte)

Type: 16-bit Baseline

Format:

15	14	9	8	6	5	3	2	0
1		BFMI333 001011		Rt3		Ra3		SEB33 010

Syntax: SEB33 Rt3, Ra3

32-bit Equivalent: SEB 3T5(Rt3), 3T5(Ra3)

Purpose: Sign-extend the least-significant byte of Ra3 and write the result to Rt3.

Description: This instruction sign-extends the least-significant byte of Ra3 to the width of a general register and writes the result to the register Rt3.

Operations:

Rt3 = Sign_Extend(Ra3(7, 0));

Exceptions: None

Privilege level: All

Note:

8.6.24. SEH33 (Sign Extend Halfword)

Type: 16-bit Baseline

Format:

15	14	9	8	6	5	3	2	0
1		BFMI333 001011		Rt3		Ra3		SEH33 011

Syntax: SEH33 Rt3, Ra3

32-bit Equivalent: SEH 3T5(Rt3), 3T5(Ra3)

Purpose: Sign-extend the least-significant halfword of Ra3 and write the result to Rt3.

Description: This instruction sign-extends the least-significant halfword of Ra3 to the width of a general register and writes the result to the register Rt3.

Operations:

Rt3 = Sign_Extend(Ra3(15, 0));

Exceptions: None

Privilege level: All

Note:

8.6.25. SHI333 (Store Halfword Immediate)

Type: 16-bit Baseline

Format:

15	14	9	8	6	5	3	2	0
1	SHI333 010110	Rt3		Ra3		imm3u		

Syntax: SHI333 Rt3, [Ra3 + (imm3u << 1)]

32-bit Equivalent: SHI 3T5(Rt3), [3T5(Ra3) + ZE(imm3u << 1)]

(imm3u is a halfword offset. In assembly programming, always write a byte offset.)

Purpose: Store a 16-bit halfword from a general register into a memory location.

Description:

This instruction stores the least-significant 16-bit halfword in the general register Rt3 to the memory location whose address is specified by adding the content of Ra3 with the zero-extended (imm3u << 1) value. Note that imm3u is a halfword-aligned offset.

The memory address has to be halfword-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

VAddr = Ra3 + Zero_Extend(imm3u << 1);
if (!Halfword_Alignment(Vaddr)) {
  Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
  Store_Memory(PAddr, HALFWORD, Attributes, Rt3(15, 0));
} else {
  Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.6.26. SLLI333 (Shift Left Logical Immediate)

Type: 16-bit Baseline

Format:

15	14	9	8	6	5	3	2	0
1	SLLI333 001010		Rt3		Ra3		imm3u	

Syntax: SLLI333 Rt3, Ra3, imm3u

32-bit Equivalent: SLLI 3T5(Rt3), 3T5(Ra3), ZE(imm3u)

Purpose: Left-shift a register content to a fixed number of bits in the range of 0 to 7.

Description: The instruction left-shifts the content of Ra3 by a fixed number of bits between 0 and 7 specified by the “imm3u” constant, fills the shifted-out bits with zero, and writes the shifted result to Rt3.

Operations:

```
Rt3 = Ra3 << imm3u; // or
Rt3 = Ra3(31-imm3u, 0).Duplicate(0, imm3u);
```

Exceptions: None

Privilege level: All

Note:

8.6.27. SLT45 (Set on Less Than Unsigned)

Type: 16-bit Baseline

Format:

15	14	9	8	5	4	0
1		SLT45 110001		Ra4		Rb5

Syntax: SLT45 Ra4, Rb5

32-bit Equivalent: SLT R15, 4T5(Ra4), Rb5

Purpose: Set a result on R15 for an unsigned less-than comparison.

Description: This instruction compares the contents of Ra4 and Rb5 as unsigned integers. If Ra4 is less than Rb5, the instruction writes R15 with 1; otherwise, it writes R15 with 0.

Operations:

```
If (Ra4 < Rb5) {
    R15 = 1;
} else {
    R15 = 0;
}
```

Exceptions: None

Privilege level: All

Note:

8.6.28. SLTI45 (Set on Less Than Unsigned Immediate)

Type: 16-bit Baseline

Format:

15	14	9	8	5	4	0
1		SLTI45 110011		Ra4		imm5u

Syntax: SLTI45 Ra4, imm5u

32-bit Equivalent: SLTI R15, 4T5(Ra4), ZE(imm5u)

Purpose: Set a result on R15 for an unsigned less-than comparison.

Description: This instruction compares the contents of Ra4 and a zero-extended imm5u as unsigned integers. If Ra4 is less than the zero-extended imm5u, this instruction writes R15 with 1; otherwise, it writes R15 with 0.

Operations:

```
If (Ra4 (unsigned) < Zero_Extend(imm5u)) {
    R15 = 1;
} else {
    R15 = 0;
}
```

Exceptions: None

Privilege level: All

Note:

8.6.29. SLTS45 (Set on Less Than Signed)

Type: 16-bit Baseline

Format:

15	14	9	8	5	4	0
1	SLTS45 110000		Ra4		Rb5	

Syntax: SLTS45 Ra4, Rb5

32-bit Equivalent: SLTS R15, 4T5(Ra4), Rb5

Purpose: Set a result on R15 for a signed less-than comparison.

Description: This instruction compares the contents of Ra4 and Rb5 as signed integers. If Ra4 is less than Rb5, this instruction writes R15 with 1; otherwise, it writes R15 with 0.

Operations:

```
If (Ra4 (signed) < Rb5) {
    R15 = 1;
} else {
    R15 = 0;
}
```

Exceptions: None

Privilege level: All

Note:

8.6.30. SLTSI45 (Set on Less Than Signed Immediate)

Type: 16-bit Baseline

Format:

15	14	9	8	5	4	0
1	SLTSI45 110010		Ra4		imm5u	

Syntax: SLTSI45 Ra4, imm5u

32-bit Equivalent: SLTSI R15, 4T5(Ra4), ZE(imm5u)

Purpose: Set a result on R15 for a signed less-than comparison.

Description: This instruction compares the contents of Ra4 and a zero-extended imm5u as signed integers. If Ra4 is less than the zero-extended imm5u, this instruction writes R15 with 1; otherwise, it writes R15 with 0.

Operations:

```
If (Ra4 (signed) < Zero_Extend(imm5u)) {
  R15 = 1;
} else {
  R15 = 0;
}
```

Exceptions: None

Privilege level: All

Note:

8.6.31. SRAI45 (Shift Right Arithmetic Immediate)

Type: 16-bit Baseline

Format:

15	14	9	8	5	4	0
1	SRAI45 001000		Rt4		imm5u	

Syntax: SRAI45 Rt4, imm5u

32-bit Equivalent: SRAI 4T5(Rt4), 4T5(Rt4), imm5u

Purpose: Right-shift a register content arithmetically (i.e. maintaining the sign of the original value) to a fixed number of bits in the range of 0 to 31.

Description: This instruction right-shifts the content of Rt4 by a fixed number of bits between 0 and 31 specified by the “imm5u” constant, duplicates the sign bit of Rt4, i.e. Rt4(31), to fill the shifted-out bits, and writes the shifted result to the source register Rt4.

Operations:

```
Rt4 = Rt4 (sign)>> imm5u; // or
Rt4 = Duplicate(Rt4(31), imm5u).Rt4(31, imm5u);
```

Exceptions: None

Privilege level: All

Note:

8.6.32. SRLI45 (Shift Right Logical Immediate)

Type: 16-bit Baseline

Format:

15	14	9	8	5	4	0
1	SRLI45 001001		Rt4		imm5u	

Syntax: SRLI45 Rt4, imm5u

32-bit Equivalent: SRLI 4T5(Rt4), 4T5(Rt4), imm5u

Purpose: Right-shift a register content logically (i.e. filling in zeros at the shifted-out bits) to a fixed number of bits in the range of 0 to 31.

Description: This instruction right-shifts the content of Rt4 by a fixed number of bits between 0 and 31 specified by the “imm5u” constant, fills the shifted-out bits with zeros, and writes the shifted result to the source register Rt4.

Operations:

```
Rt4 = Rt4 (0)>> imm5u; // or
Rt4 = Duplicate(0, imm5u).Rt4(31, imm5u);
```

Exceptions: None

Privilege level: All

Note:

8.6.33. SUB (Subtract Register)

Type: 16-bit Baseline

Format:

SUB333								
15	14	9	8	6	5	3	2	0
1	SUB333 001101		Rt3		Ra3		Rb3	

SUB45

15	14	9	8	5	4	0
1	SUB45 000101		Rt4		Rb5	

Syntax: SUB333 Rt3, Ra3, Rb3

SUB45 Rt4, Rb5

32-bit Equivalent: SUB 3T5(Rt3), 3T5(Ra3), 3T5(Rb3) // SUB333

SUB 4T5(Rt4), 4T5(Rt4), Rb5 // SUB45

Purpose: Subtract the contents of two registers.

Description: For SUB333, this instruction subtracts the content of Rb3 from Ra3 and writes the result to Rt3; for SUB45, it subtracts the content of Rb5 from Rt4 and writes the result to the source register Rt4.

Operations:

Rt3 = Ra3 - Rb3; // SUB333

Rt4 = Rt4 - Rb5; // SUB45

Exceptions: None

Privilege level: All

Note:



8.6.34. SUBI (Subtract Immediate)

Type: 16-bit Baseline

Format:

SUBI33							
15	14	9	8	6	5	3	2 0
1	SUBI33 001111		Rt3		Ra3		imm3u

SUBI45

15	14	9	8	5	4	0
1	SUBI45 000111		Rt4			imm5u

Syntax: SUBI333 Rt3, Ra3, imm3u

SUBI45 Rt4, imm5u

32-bit Equivalent: ADDI 3T5(Rt3), 3T5(Ra3), NEG(imm3u) // SUBI333

ADDI 4T5(Rt4), 4T5(Rt4), NEG(imm5u) // SUBI45

Purpose: Subtract a zero-extended immediate from the content of a register.

Description: For SUBI333, this instruction subtracts the zero-extended 3-bit immediate “imm3u” from the content of Ra3 and writes the result to Rt3; for SUBI45, it subtracts the zero-extended 5-bit immediate “imm5u” from the content of Rt4 and writes the result to the source register Rt4.

Operations:

Rt3 = Ra3 - ZE(imm3u); // SUBI 333

Rt4 = Rt4 - ZE(imm5u); // SUBI 45

Exceptions: None

Privilege level: All

Note:



8.6.35. SWI333 (Store Word Immediate)

Type: 16-bit Baseline

Format:

SWI333								
15	14	9	8	6	5	3	2	0
1	SWI333 010100		Rt3		Ra3		imm3u	

SWI333.bi

15	14	9	8	6	5	3	2	0
1	SWI333.bi 010101		Rt3		Ra3		imm3u	

Syntax: SWI333 Rt3, [Ra3 + (imm3u << 2)]

SWI333.bi Rt3, [Ra3], (imm3u << 2)

32-bit Equivalent: SWI 3T5(Rt3), [3T5(Ra3) + ZE(imm3u << 2)]

SWI.bi 3T5(Rt3), [3T5(Ra3)], ZE(imm3u << 2)

(imm3u is a word offset. In assembly programming, always write a byte offset.)

Purpose: Store a 32-bit word from a general register into memory.

Description:

This instruction stores a word from the general register Rt3 into memory. There are two different forms to specify the memory address: the regular form uses Ra3 + ZE(imm3u << 2) as its memory address while the .bi form uses Ra3. For the .bi form, the Ra3 register is updated with the Ra3 + ZE(imm3u << 2) value after the memory store operation.

The memory address has to be word-aligned. Otherwise, it cause a Data Alignment Check exception to occur.

Operations:

```

Addr = Ra3 + Zero_Extend((imm3u << 2));
If (.bi_form) {
    Vaddr = Ra3;
} else {
    Vaddr = Addr;
}
if (!Word_Aligned(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
    Store_Memory(PAddr, WORD, Attributes, Rt3);
    If (.bi_form) { Ra3 = Addr; }
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.6.36. SWI37 (Store Word Immediate with Implied FP)

Type: 16-bit Baseline

Format:

15	14	11	10	8	7	6	0
1	XWI37 0111	Rt3		SWI37 1		imm7u	

Syntax: SWI37 Rt3, [FP + (imm7u << 2)]

32-bit Equivalent: SWI 3T5(Rt3), [FP + ZE(imm7u << 2)]

(imm7u is a word offset. In assembly programming, always write a byte offset.)

Purpose: Store a 32-bit word from a general register into memory.

Description: This instruction stores a word from the general register Rt3 into memory. The memory address is specified by adding the implied FP (i.e. R28) register and the zero-extended (imm7u << 2) value. Note that the imm7u is a word-aligned offset.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

VAddr = FP (i.e. R28) + Zero_Extend(imm7u << 2);
if (!Word_Alignment(Vaddr)) {
  Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
  Store_Memory(PAddr, WORD, Attributes, Rt3);
} else {
  Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.6.37. SWI450 (Store Word Immediate)

Type: 16-bit Baseline

Format:

15	14	9	8	5	4	0
1	SWI450 011011		Rt4		Ra5	

Syntax: SWI450 Rt4, [Ra5]

32-bit Equivalent: SWI 4T5(Rt4), [Ra5 + 0]

Purpose: Store a 32-bit word from a general register into memory.

Description:

This instruction stores a word from the general register Rt4 into memory. The memory address is specified in Ra5.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

VAddr = Ra5;
if (!Word_Aligned(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
if (Excep_status == NO_EXCEPTION) {
    Store_Memory(PAddr, WORD, Attributes, Rt4);
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:



8.6.38. X11B33 (Extract the Least 11 Bits)

Type: 16-bit Extension

Format:

15	14	9	8	6	5	3	2	0
1		BFMI333 001011		Rt3		Ra3		X11B33 101

Syntax: X11B33 Rt3, Ra3

32-bit Equivalent: ANDI 3T5(Rt3), 3T5(Ra3), 0x7FF

Purpose: Extract the least-significant 11 bits of Ra3 and write the result to Rt3.

Description: This instruction extracts the least-significant 11 bits of Ra3 and writes the result to the register Rt3.

Operations:

Rt3 = Ra3 & 0x7FF;

Exceptions: None

Privilege level: All

Note:

8.6.39. XLSB33 (Extract LSB)

Type: 16-bit Extension

Format:

15	14	9	8	6	5	3	2	0
1		BFMI333 001011		Rt3		Ra3		XLSB33 100

Syntax: XLSB33 Rt3, Ra3

32-bit Equivalent: ANDI 3T5(Rt3), 3T5(Ra3), 0x1

Purpose: Extract the least-significant bit of Ra3 and write the result to Rt3.

Description: This instruction extracts the least-significant bit of Ra3 and writes the result to the register Rt3.

Operations:

Rt3 = Ra3 & 0x1;

Exceptions: None

Privilege level: All

Note:

8.6.40. ZEB33 (Zero Extend Byte)

Type: 16-bit Baseline

Format:

15	14	9	8	6	5	3	2	0
1		BFMI333 001011	Rt3		Ra3		ZEB33 000	

Syntax: ZEB33 Rt3, Ra3

32-bit Equivalent: ANDI 3T5(Rt3), 3T5(Ra3), 0xFF

Purpose: Zero-extend the least-significant byte of Ra3 and write the result to Rt3.

Description: This instruction zero-extends the least-significant byte of Ra3 to the width of a general register and writes the result to the register Rt3.

Operations:

Rt3 = Zero_Extend(Ra3(7, 0));

Exceptions: None

Privilege level: All

Note:

8.6.41. ZEH33 (Zero Extend Halfword)

Type: 16-bit Baseline

Format:

15	14	9	8	6	5	3	2	0
1	BFMI333 001011		Rt3		Ra3		ZEH33 001	

Syntax: ZEH33 Rt3, Ra3

32-bit Equivalent: ZEH 3T5(Rt3), 3T5(Ra3)

Purpose: Zero-extend the least-significant halfword of Ra3 and write the result to Rt3.

Description: This instruction zero-extends the least-significant halfword of Ra3 to the width of a general register and writes the result to the register Rt3.

Operations:

`Rt3 = Zero_Extend(Ra3(15, 0));`

Exceptions: None

Privilege level: All

Note:

8.7. 16-bit and 32-bit Baseline Version 2 Instructions

The following Baseline Version 2 instructions are added to improve code density.

8.7.1. ADDI10S (Add Immediate with Implied Stack Pointer)

Type: 16-bit Baseline Version 2

Format:

ADDI10S

15	14	10	9	0
1	ADDI10S 11011			imm10s

Syntax: ADDI10.sp imm10s

32-bit Equivalent: ADDI r31, r31, SE(imm10s)

Purpose: Add a sign-extended immediate to the content of the stack pointer register (R31).

Description: This instruction adds the sign-extended 10-bit immediate “imm10s” to the content of R31 (stack pointer) and writes the result back to R31.

Operations:

$$R31 = R31 + SE(i\ mm10s);$$

Exceptions: None

Privilege level: All

Note:

8.7.2. LWI37SP (Load Word Immediate with Implied SP)

Type: 16-bit Baseline Version 2

Format:

15	14	11	10	8	7	6	0
1	XWI37SP 1110	Rt3		LWI37SP 0		imm7u	

Syntax: LWI37.sp Rt3, [+ (imm7u << 2)]

32-bit Equivalent: LWI 3T5(Rt3), [SP + ZE(imm7u << 2)]

(imm7u is a word offset. In assembly programming, always write a byte offset.)

Purpose: Load a 32-bit word from memory into a general register.

Description:

This instruction loads a word from memory into the general register Rt3. The memory address is specified by adding the implied SP (i.e. R31) register and the zero-extended (imm7u << 2) value. Note that the imm7u is a word-aligned offset.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

VAddr = SP (i.e. R31) + Zero_Extend(imm7u << 2);
if (!Word_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
if (Excep_status == NO_EXCEPTION) {
    Wdata(31, 0) = Load_Memory(PAddr, WORD, Attributes);
    Rt3 = Wdata(31, 0);
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.7.3. SWI37SP (Store Word Immediate with Implied SP)

Type: 16-bit Baseline Version 2

Format:

15	14	11	10	8	7	6	0
1	XWI37SP 1110	Rt3	SWI37SP 1	imm7u			

Syntax: SWI37.sp Rt3, [+ (imm7u << 2)]

32-bit Equivalent: SWI 3T5(Rt3), [SP + ZE(imm7u << 2)]

(imm7u is a word offset. In assembly programming, always write a byte offset.)

Purpose: Store a 32-bit word from a general register into memory.

Description:

This instruction stores a word from the general register Rt3 into memory. The memory address is specified by adding the implied SP (i.e. R31) register and the zero-extended (imm7u << 2) value. Note that the imm7u is a word-aligned offset.

The memory address has to be word-aligned. Otherwise, it causes a Data Alignment Check exception to occur.

Operations:

```

VAddr = SP (i.e. R31) + Zero_Extend(imm7u << 2);
if (!Word_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_status == NO_EXCEPTION) {
    Store_Memory(PAddr, WORD, Attributes, Rt3);
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Note:

8.7.4. ADDI.gp (GP-implied Add Immediate)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	18	0
0	SBGP 011111	Rt	1 ADDI			imm19s	

Syntax: ADDI.gp Rt, imm19s

Purpose: Add the content of the implied GP (R29) register and a signed constant.

Description: This instruction adds the content of Gp (R29) and the sign-extended imm19s, then writes the result to Rt. The imm19s value covers a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

$$Rt = R29 + SE(i\ mm19s);$$

Exceptions: None

Privilege level: All

Note:

8.7.5. DIVR (Unsigned Integer Divide to Registers)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		Rs		DIVR 10111		

Syntax: DIVR Rt, Rs, Ra, Rb

Purpose: Divide the unsigned integer content of one register with that of another register.

Description:

This instruction divides the 32-bit content of Ra with that of Rb. It writes the 32-bit quotient result to Rt and the 32-bit remainder result to Rs. The contents of Ra and Rb are treated as unsigned integers. If Rt and Rs point to the same register, this instruction only writes the quotient result to that register.

If the content of Rb is zero and the IDIVZE bit of the INT_MASK register is 1, this instruction generates an Arithmetic exception. The IDIVZE bit of the INT_MASK register enables exception generation for the “Divide-By-Zero” condition.

Operations:

```
If (Rb != 0) {  
    quotient = Floor(CONCAT(1`b0, Ra) / CONCAT(1`b0, Rb));  
    remainder = CONCAT(1`b0, Ra) mod CONCAT(1`b0, Rb);  
    Rt = quotient;  
    If (Rs != Rt) {  
        Rs = remainder;  
    }  
} else if (INT_MASK.IDIVZE == 0) {  
    Rt = 0;  
    Rs = 0;  
} else {  
    Generate_Exception(ArithmetiC);  
}
```

Exceptions: Arithmetic**Privilege level:** All**Note:**

8.7.6. DIVSR (Signed Integer Divide to Registers)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		Rs		DIVSR 10110		

Syntax: DIVSR Rt, Rs, Ra, Rb

Purpose: Divide the signed integer content of one register with that of another register.

Description:

This instruction divides the 32-bit content of Ra with that of Rb. It writes the 32-bit quotient result to Rt and the 32-bit remainder result to Rs. The contents of Ra and Rb are treated as signed integers. If Rt and Rs point to the same register, the instruction only writes the quotient result to that register.

If the content of Rb is zero and the IDIVZE bit of the INT_MASK register is 1, this instruction generates an Arithmetic exception. The IDIVZE bit of the INT_MASK register enables exception generation for the “Divide-By-Zero” condition. If the quotient overflows, this instruction always generates an Arithmetic exception. The overflow condition is as follows:

- Positive quotient > 0x7FFF FFFF (When Ra = 0x80000000 and Rb = 0xFFFFFFFF)

Operations:

```
If (Rb != 0) {  
    quotient = Floor(Ra / Rb);  
    if (IsPositive(quotient) && quotient > 0x7FFFFFFF) {  
        Generate_Exception(Arithmeti c);  
    }  
    remainder = Ra mod Rb;  
    Rt = quotient;  
    If (Rs != Rt) {  
        Rs = remainder;  
    }  
} else if (INT_MASK.IDIVZE == 0) {  
    Rt = 0;  
    Rs = 0;  
} else {  
    Generate_Exception(Arithmeti c);  
}
```

Exceptions: Arithmetic**Privilege level:** All**Note:**

8.7.7. LBI.gp (GP-implied Load Byte Immediate)

Type: 32-bit Baseline Version 2

Format:

31	30	24	20	19	18	0
0	LBGP 010111	Rt	0	LBI	imm19s	

Syntax: LBI.gp Rt, [+ imm19s]

Purpose: Load a zero-extended 8-bit byte from memory into a general register.

Description: This instruction loads a zero-extended byte from memory into the general register Rt. The memory address is specified by the implied GP register (R29) plus a sign-extended imm19s value. The imm19s value covers a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

```

Vaddr = R29 + Sign_Extend(imm19s);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Bdata(7,0) = Load_Memory(PAddr, BYTE, Attributes);
    Rt = Zero_Extend(Bdata(7,0));
} else {
    Generate_Exception(Excep_status);
}
  
```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:



8.7.8. LBSI.gp (GP-implied Load Byte Signed Immediate)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	18	0
0	LBGP 010111	Rt	1	LBSI		imm19s	

Syntax: LBSI.gp Rt, [+ imm19s]

Purpose: Load a sign-extended 8-bit byte from memory into a general register.

Description: This instruction loads a sign-extended byte from memory into the general register Rt. The memory address is specified by the implied GP register (R29) plus a sign-extended imm19s value. The imm19s value covers a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

```

Vaddr = R29 + Sign_Extend(imm19s);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_Status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_Status == NO_EXCEPTION) {
    Bdata(7,0) = Load_Memory(PAddr, BYTE, Attributes);
    Rt = Sign_Extend(Bdata(7,0));
} else {
    Generate_Exception(Excep_Status);
}
  
```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8.7.9. LBUP (Load Byte with User Privilege Translation)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		LBUP 00100000		

Syntax: LBUP Rt, [Ra + (Rb << sv)]

Purpose: Load a zero-extended 8-bit byte from memory into a general register with the user mode privilege address translation.

Description: This instruction loads a zero-extended byte from memory address Ra + (Rb << sv) into the general register Rt with the user mode privilege address translation regardless of the current processor operation mode (i.e. PSW.POM) and the current data address translation state (i.e. PSW.DT).

Operations:

```

Vaddr = Ra + (Rb << sv);
(PAddr, Attributes) = Address_Translation(Vaddr, TRANSLATE);
Excep_status = Page_Exception(Attributes, USER_MODE, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Bdata(7,0) = Load_Memory(PAddr, Byte, Attributes);
    Rt = Zero_Extend(Bdata(7,0));
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8.7.10. LHI.gp (GP-implied Load Halfword Immediate)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	18	17	0
0	HWGP 011110	Rt	00 LHI					imm18s

Syntax: LHI.gp Rt, [+ (imm18s << 1)]

(imm18s is a halfword offset. In assembly programming, always write a byte offset.)

Purpose: Load a zero-extended 16-bit halfword from memory into a general register.

Description: This instruction loads a zero-extended halfword from memory into the general register Rt. The memory address is specified by the implied GP register (R29) plus a sign-extended (imm18s << 1) value. The (imm18s << 1) value covers a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

```

Vaddr = R29 + Sign_Extend(imm18s << 1);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_Status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_Status == NO_EXCEPTION) {
    Hdata(15, 0) = Load_Memory(PAddr, HALFWORD, Attributes);
    Rt = Zero_Extend(Hdata(15, 0));
} else {
    Generate_Exception(Excep_Status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8.7.11. LHSI.gp (GP-implied Load Signed Halfword Immediate)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	18	17	0
0	HWGP 011110	Rt		01 LHSI				imm18s

Syntax: LHSI.gp Rt, [+ (imm18s << 1)]

(imm18s is a halfword offset. In assembly programming, always write a byte offset.)

Purpose: Load a sign-extended 16-bit halfword from memory into a general register.

Description: This instruction loads a sign-extended halfword from memory into the general register Rt. The memory address is specified by the implied GP register (R29) plus a sign-extended (imm18s << 1) value. The (imm18s << 1) value covers a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

```

Vaddr = R29 + Sign_Extend(imm18s << 1);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_Status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_Status == NO_EXCEPTION) {
    Hdata(15, 0) = Load_Memory(PAddr, HALFWORD, Attributes);
    Rt = Sign_Extend(Hdata(15, 0));
} else {
    Generate_Exception(Excep_Status);
}
  
```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8.7.12. LMWA (Load Multiple Word with Alignment Check)

Type: 32-bit Baseline Version 2

Format:

LMWA															
31	30	25	24	20	19	15	14	10	9	6	5	4	3	2	10
0	LSMW 011101		Rb		Ra		Re		Enable4		LMW 0	b:0 a:1	i:0 d:1	m	01

Syntax: LMWA.{b| a}{i | d}{m?} Rb, [Ra], Re, Enable4

Purpose: Load multiple 32-bit words from sequential memory locations into multiple registers.

Description:

This instruction loads multiple 32-bit words from sequential memory addresses specified by the base address register Ra and the {b | a}{i | d} options into a continuous range or a subset of general purpose registers. The subset of general purpose registers is specified by a register list formed by Rb, Re, and the four-bit Enable4 field as follows.

<Register List> = a range from [Rb, Re] and a list from <Enable4>

- {b | a} option specifies the way how the first address is generated. {b} uses the contents of Ra as the first memory load address. {a} uses either Ra+4 or Ra-4 for the {i | d} option respectively as the first memory load address.
- {i | d} option specifies the direction of the address change. {i} generates increasing addresses from Ra and {d} generates decreasing addresses from Ra.
- {m?} option, if it is specified, indicates that the base address register will be updated to the value computed in the following formula at the completion of this instruction.

TNReg = Total number of registers loaded

Updated value = Ra + (4 * TNReg) for {i} option

Updated value = Ra - (4 * TNReg) for {d} option

- [Rb, Re] specifies a range of registers which will be loaded by this instruction. Rb(4,0) specifies the first register number in the continuous register range and Re(4,0) specifies the last register number. In addition to the range of registers, <Enable4(3,0)> specifies the load of 4 individual registers from R28 to R31 (s9/fp, gp, lp, sp) which have special calling convention usage. The exact mapping of Enable4(3,0) bits and registers is as follows:

Bits	Enable4(3) Format(9)	Enable4(2) Format(8)	Enable4(1) Format(7)	Enable4(0) Format(6)
Registers	R28	R29	R30	R31

- Several constraints are imposed for the <Register List>:
 - If [Rb(4,0), Re(4,0)] specifies at least one register:
 - ◆ Rb(4,0) <= Re(4,0) AND
 - ◆ 0 <= Rb(4,0), Re(4,0) < 28
 - If [Rb(4,0), Re(4,0)] specifies no register at all:
 - ◆ Rb(4,0) == Re(4,0) = 0b11111 AND
 - ◆ Enable4(3,0) != 0b0000
 - If these constraints are not met, UNPREDICTABLE results will happen to the contents of all registers after this instruction.
- The registers are loaded in sequence from matching memory locations with one exception. That is, the lowest-numbered register is loaded from the lowest memory address while the highest-numbered register is loaded from the highest memory address with the following exception.
 - The matching memory locations of R28 (fp) and R31 (sp) are swapped. That is, the memory locations for R28-R31 are as follows.

R28 ----- High memory location
 R30
 R29
 R31 ----- Low memory location

Note that the load sequence of this instruction involving R28 and R31 is different from

the load/store sequence of LMW/SMW.

- If the base address register update {m?} option is specified while the base address register Ra is also specified in the <Register Specification>, there are two source values for the final content of the base address register Ra. In this case, the final value of Ra is UNPREDICTABLE. As to the rest of the loaded registers, they should have the values as if the base address register update {m?} option is not specified.
- This instruction can only handle word-aligned memory address.

Operation:

```

TNReg = Count_Registers(register_list);
if ("bi") {
    B_addr = Ra;
    E_addr = Ra + (TNReg * 4) - 4;
} elseif ("ai") {
    B_addr = Ra + 4;
    E_addr = Ra + (TNReg * 4);
} elseif ("bd") {
    B_addr = Ra - (TNReg * 4) + 4;
    E_addr = Ra;
} else { // "ad"
    B_addr = Ra - (TNReg * 4);
    E_addr = Ra - 4
}
VA = B_addr;
if (!word-aligned(VA)) {
    Generate_Exception(Data_Alignment_Check);
}
for (i = 0 to 27, 31, 29, 30, 28) {
    if (register_list[i] == 1) {
        (PA, Attributes) = Address_Translation(VA, PSW.DT);
        Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
        If (Excep_status == NO_EXCEPTION) {
            Ri = Load_Memory(PA, Word, Attributes);
            VA = VA + 4;
        } else {
            Generate_Exception(Excep_status);
        }
    }
}
  
```

```

    }
}

if ("im") {
  Ra = Ra + (TNReg * 4);
} else { // "dm"
  Ra = Ra - (TNReg * 4);
}

```

Exception: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

- If the base register update is not specified, the base register value is unchanged. This applies even if the instruction loads its own base register and the loading of the base register occurs earlier than the exception event. For example, suppose the instruction is
`LMW.bd R2, [R4], R4, 0b0000`
 And the implementation loads R4, then R3, and finally R2. If an exception occurs on any of the accesses, the value in the base register R4 of the instruction is unchanged.
- If the base register update is specified, the value left in the base register is unchanged.
- If the instruction loads only one general-purpose register, the value in that register is unchanged.
- If the instruction loads more than one general-purpose register, UNPREDICTABLE values are left in destination registers which are not the base register of the instruction.

Interruption: Whether this instruction is interruptible or not is implementation-dependent.

Privilege Level: all

Note:

- (1) LMW and SMW instructions do not guarantee atomicity among individual memory access operations. Neither do they guarantee single access to a memory location during the execution. Any I/O access that has side-effects other than simple stable memory-like access behavior should not use these two instructions.
- (2) The memory access order among the words accessed by LMW/SMW is not defined here and should be implementation-dependent. However, the more likely access order in an implementation is:

- For LMW/SMW.i : increasing memory addresses from the base address.
 - For LMW/SMW.d: decreasing memory addresses from the base address.
- (3) The memory access order within an un-aligned word accessed is not defined here and should be implementation-dependent. However, the more likely access order in an implementation is:
- For LMW/SMW.i: the aligned low address of the word and then the aligned high address of the word. If an interruption occurs, the EVA register will contain the starting low address of the un-aligned word.
 - For LMW/SMW.d: the aligned high address of the word and then the aligned low address of the word. If an interruption occurs, the EVA register will contain “base un-aligned address + 4” of the first word or the starting low address of the remaining decreasing memory word.
- (4) Based on the more likely access order of (2) and (3), upon interruption, the EVA register for un-aligned LMW/SMW is more likely to have the following value:
- For LMW/SMW.i: the starting low addresses of the accessed words or “Ra + (TNReg * 4)” where TNReg represents the total number of registers loaded or stored.
 - For LMW/SMW.d: the starting low addresses of the accessed words or “Ra + 4”.

8.7.13. LWI.gp (GP-implied Load Word Immediate)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	17	16	0
0	HWGP 011110	Rt		110 LWI				imm17s

Syntax: LWI.gp Rt, [+ (imm17s << 2)]

(imm17s is a word offset. In assembly programming, always write a byte offset.)

Purpose: Load a 32-bit word from memory into a general register.

Description: This instruction loads a 32-bit word from memory into the general register Rt. The memory address is specified by the implied GP register (R29) plus a sign-extended (imm17s << 2) value. The (imm17s << 2) value covers a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

```

Vaddr = R29 + Sign_Extend(imm17s << 2);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_Status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_Status == NO_EXCEPTION) {
    Wdata(31, 0) = Load_Memory(PAddr, WORD, Attributes);
    Rt = Wdata(31, 0);
} else {
    Generate_Exception(Excep_Status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8.7.14. MADDR32 (Multiply and Add to 32-Bit Register)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		Rb		0001 GPR		MADD32 110011		

Syntax: MADDR32 Rt, Ra, Rb

Purpose: Multiply the contents of two 32-bit registers and add the lower 32-bit multiplication result to the 32-bit content of a destination register. Write the final result back to the destination register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb. It adds the lower 32-bit multiplication result to the content of Rt and writes the final result back to Rt. The contents of Ra and Rb can be either signed or unsigned integers.

Operations:

```
Mresul t = Ra * Rb;
Rt = Rt + Mresul t(31, 0);
```

Exceptions: None

Privilege level: All

Note:

8.7.15. MSUBR32 (Multiply and Subtract from 32-Bit Register)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		Rb		0001 GPR	MSUB32 110101			

Syntax: MSUBR32 Rt, Ra, Rb

Purpose: Multiply the contents of two 32-bit registers and subtract the lower 32-bit multiplication result from the 32-bit content of a destination register. Write the final result back to the destination register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb, subtracts the lower 32-bit multiplication result from the content of Rt, then writes the final result back to Rt. The contents of Ra and Rb can be either signed or unsigned integers.

Operations:

```
Mresult = Ra * Rb;
Rt = Rt - Mresult(31, 0);
```

Exceptions: None

Privilege level: All

Note:

8.7.16. MULR64 (Multiply Word Unsigned to Registers)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		Rb		0001 GPR	MULT64 101001			

Syntax: MULR64 Rt, Ra, Rb

Purpose: Multiply the unsigned integer contents of two 32-bit registers and write the 64-bit result to an even/odd pair of 32-bit registers.

Description:

This instruction multiplies the 32-bit content of Ra with that of Rb and writes the 64-bit multiplication result to an even/odd pair of registers containing Rt. Rt(4,1) index d determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

How the register pair contains the 64-bit result depends on the current data endian. When the data endian is *big*, the even register of the pair contains the high 32-bit of the result and the odd register of the pair contains the low 32-bit of the result. In contrast, when the data endian is *little*, the odd register of the pair contains the high 32-bit of the result and the even register of the pair contains the low 32-bit of the result.

The contents of Ra and Rb are treated as unsigned integers.

Operations:

```
Mresul t = CONCAT(1`b0, Ra) * CONCAT(1`b0, Rb);  
If (PSW.BE == 1) {  
    R[Rt(4, 1).0(0)](31, 0) = Mresul t(63, 32);  
    R[Rt(4, 1).1(0)](31, 0) = Mresul t(31, 0);  
} else {  
    R[Rt(4, 1).1(0)](31, 0) = Mresul t(63, 32);  
    R[Rt(4, 1).0(0)](31, 0) = Mresul t(31, 0);  
}
```

Exceptions: None**Privilege level:** All**Note:**

8.7.17. MULSR64 (Multiply Word Signed to Registers)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	15	14	10	9	6	5	0
0	ALU_2 100001	Rt		Ra		Rb		0001 GPR	MULTS64 101000			

Syntax: MULSR64 Rt, Ra, Rb

Purpose: Multiply the signed integer contents of two 32-bit registers and write the 64-bit result to an even/odd pair of 32-bit registers.

Description:

This instruction multiplies the 32-bit content of Ra with the 32-bit content of Rb and writes the 64-bit multiplication result to an even/odd pair of registers containing Rt. Rt(4,1) index d determines the even/odd pair group of the two registers. Specifically, the register pair includes register $2d$ and $2d+1$.

How the register pair contains the 64-bit result depends on the current data endian. When the data endian is *big*, the even register of the pair contains the high 32-bit of the result and the odd register of the pair contains the low 32-bit of the result. In contrast, when the data endian is *little*, the odd register of the pair contains the high 32-bit of the result and the even register of the pair contains the low 32-bit of the result.

The contents of Ra and Rb are treated as signed integers.

Operations:

```
Mresul t = Ra * Rb;  
If (PSW.BE == 1) {  
    R[Rt(4, 1).0(0)](31, 0) = Mresul t(63, 32);  
    R[Rt(4, 1).1(0)](31, 0) = Mresul t(31, 0);  
} else {  
    R[Rt(4, 1).1(0)](31, 0) = Mresul t(63, 32);  
    R[Rt(4, 1).0(0)](31, 0) = Mresul t(31, 0);  
}
```

Exceptions: None**Privilege level:** All**Note:**

8.7.18. SBI(gp) (GP-implied Store Byte Immediate)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	18	0
0	SBGP 011111	Rt	0	SBI		imm19s	

Syntax: SBI(gp) Rt, [+ imm19s]

Purpose: Store an 8-bit byte from a general register into a memory location.

Description: This instruction stores the least-significant 8-bit byte in the general register Rt to the memory location. The memory address is specified by the implied GP register (R29) plus a sign-extended imm19s value. The imm19s value covers a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

```

Vaddr = R29 + Sign_Extend(imm19s);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_Status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_Status == NO_EXCEPTION) {
    Store_Memory(PAddr, BYTE, Attributes, Rt(7,0));
} else {
    Generate_Exception(Excep_Status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

8.7.19. SBUP (Store Byte with User Privilege Translation)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	15	14	10	9	8	7	0
0	MEM 011100	Rt		Ra		Rb		sv		SBUP 00101000		

Syntax: SB Rt, [Ra + (Rb << sv)]

Purpose: Store an 8-bit byte from a general register into memory with the user mode privilege address translation.

Description: This instruction stores the least-significant 8-bit byte in the general register Rt to the memory Ra + (Rb << sv) with the user mode privilege address translation regardless of the current processor operation mode (i.e. PSW.POM) and the current data address translation state (i.e. PSW.DT).

Operations:

```

Vaddr = Ra + (Rb << sv);
(PAddr, Attributes) = Address_Translation(Vaddr, TRANSLATE);
Excep_status = Page_Exception(Attributes, USER_MODE, STORE);
If (Excep_status == NO_EXCEPTION) {
  Store_Memory(PAddr, BYTE, Attributes, Rt(7, 0));
} else {
  Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error

Privilege level: All

Note:

8.7.20. SHI.gp (GP-implied Store Halfword Immediate)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	18	17	0
0	HWGP 011110	Rt		10 SHI				imm18s

Syntax: SHI.gp Rt, [+ (imm18s << 1)]

(imm18s is a halfword offset. In assembly programming, always write a byte offset.)

Purpose: Store a 16-bit halfword from a general register into a memory location.

Description: This instruction stores the least-significant 16-bit halfword in the general register Rt to the memory location. The memory address is specified by the implied GP register (R29) plus a sign-extended (imm18s << 1) value. The (imm18s << 1) value covers a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

```

Vaddr = R29 + Sign_Extend(imm18s << 1);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_Status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_Status == NO_EXCEPTION) {
    Store_Memory(PAddr, HALFWORD, Attributes, Rt(15, 0));
} else {
    Generate_Exception(Excep_Status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8.7.21. SMWA (Store Multiple Word with Alignment Check)

Type: 32-bit Baseline Version 2

Format:

SMWA															
31	30	25	24	20	19	15	14	10	9	6	5	4	3	2	10
0	LSMW 011101	Rb		Ra			Re		Enable4		SMW 1	b:0 a:1	i:0 d:1	m	01

Syntax: SMWA.{b | a}{i | d}{m?} Rb, [Ra], Re, Enable4

Purpose: Store multiple 32-bit words from multiple registers into sequential memory locations.

Description:

This instruction stores multiple 32-bit words from a range or a subset of source general-purpose registers to sequential memory addresses specified by the base address register Ra and the {b | a}{i | d} options. The source registers are specified by a register list formed by Rb, Re, and the four-bit Enable4 field as follows.

<Register List> = a range from {Rb, Re} and a list from <Enable4>

- {b | a} option specifies the way how the first address is generated. {b} uses the contents of Ra as the first memory store address. {a} uses either Ra+4 or Ra-4 for the {i | d} option respectively as the first memory store address.
- {i | d} option specifies the direction of the address change. {i} generates increasing addresses from Ra and {d} generates decreasing addresses from Ra.
- {m?} option, if it is specified, indicates that the base address register will be updated to the value computed in the following formula at the completion of this instruction.

TNReg = Total number of registers stored

Updated value = Ra + (4 * TNReg) for {i} option

Updated value = Ra - (4 * TNReg) for {d} option

- [Rb, Re] specifies a range of registers whose contents will be stored by this instruction. Rb(4,0) specifies the first register number in the continuous register range and Re(4,0) specifies the last register number. In addition to the range of registers, <Enable4(3,0)> specifies the store of 4 individual registers from R28 to R31 (s9/fp, gp, lp, sp) which have special calling convention usage. The exact mapping of Enable4(3,0) bits and registers is as follows:

Bits	Enable4(3) Format(9)	Enable4(2) Format(8)	Enable4(1) Format(7)	Enable4(0) Format(6)
Registers	R28	R29	R30	R31

- Several constraints are imposed for the <Register List>:
 - If [Rb, Re] specifies at least one register:
 - ◆ Rb(4,0) <= Re(4,0) AND
 - ◆ 0 <= Rb(4,0), Re(4,0) < 28
 - If [Rb, Re] specifies no register at all:
 - ◆ Rb(4,0) == Re(4,0) = 0b11111 AND
 - ◆ Enable4(3,0) != 0b0000
 - If these constraints are not met, UNPREDICTABLE results will happen to the contents of the memory range pointed to by the base register. If the {m?} option is also specified after this instruction, an UNPREDICTABLE result will happen to the base register itself too.
- The registers are stored in sequence to matching memory locations with one exception. That is, the lowest-numbered register is stored to the lowest memory address while the highest-numbered register is stored to the highest memory address with the following exception.
 - The matching memory locations of R28 (fp) and R31 (sp) are swapped. That is, the memory locations for R28-R31 are as follows.

R28 ----- High memory location

R30

R29

R31 ----- Low memory location

Note that the load sequence of this instruction involving R28 and R31 is different from the load/store sequence of LMW/SMW.

- If the base address register Ra is specified in the <Register Specification>, the value stored to memory from the register Ra is the Ra value before this instruction is executed.
- This instruction can only handle word-aligned memory address.

Operation:

```

TNReg = Count_Registers(register_list);
if ("bi") {
    B_addr = Ra;
    E_addr = Ra + (TNReg * 4) - 4;
} elseif ("ai") {
    B_addr = Ra + 4;
    E_addr = Ra + (TNReg * 4);
} elseif ("bd") {
    B_addr = Ra - (TNReg * 4) + 4;
    E_addr = Ra;
} else { // "ad"
    B_addr = Ra - (TNReg * 4);
    E_addr = Ra - 4
}
VA = B_addr;
if (!word_aligned(VA)) {
    Generate_Exception(Data_Alignment_Check);
}
for (i = 0 to 27, 31, 29, 30, 28) {
    if (register_list[i] == 1) {
        (PA, Attributes) = Address_Translation(VA, PSW.DT);
        Excep_status = Page_Exception(Attributes, PSW.POM, STORE);
    }
}

```

```

        If (Excep_status == NO_EXCEPTION) {
            Store_Memory(PA, Word, Attributes, Ri);
            VA = VA + 4;
        } else {
            Generate_Exception(Excep_status);
        }
    }

if ("im") {
    Ra = Ra + (TNReg * 4);
} else { // "dm"
    Ra = Ra - (TNReg * 4);
}

```



Exception: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

- The base register value is left unchanged on an exception event, no matter whether the base register update is specified or not.

Interruption: Whether this instruction is interruptible or not is implementation-dependent.

Privilege Level: all

Note:

- (1) LMW and SMW instructions do not guarantee atomicity among individual memory access operations. Neither do they guarantee single access to a memory location during the execution. Any I/O access that has side-effects other than simple stable memory-like access behavior should not use these two instructions.
- (2) The memory access order among the words accessed by LMW/SMW is not defined here and should be implementation-dependent. However, the more likely access order in an implementation is:
 - For LMW/SMW.i : increasing memory addresses from the base address.
 - For LMW/SMW.d: decreasing memory addresses from the base address.
- (3) The memory access order within an un-aligned word accessed is not defined here and should be implementation-dependent. However, the more likely access order in an

implementation is:

- For LMW/SMW.i: the aligned low address of the word and then the aligned high address of the word. If an interruption occurs, the EVA register will contain the starting low address of the un-aligned word.
 - For LMW/SMW.d: the aligned high address of the word and then the aligned low address of the word. If an interruption occurs, the EVA register will contain “base un-aligned address + 4” of the first word or the starting low address of the remaining decreasing memory word.
- (4) Based on the more likely access order of (2) and (3), upon interruption, the EVA register for un-aligned LMW/SMW is more likely to have the following value:
- For LMW/SMW.i: the starting low addresses of the accessed words or “Ra + (TNReg * 4)” where TNReg represents the total number of registers loaded or stored.
 - For LMW/SMW.d: the starting low addresses of the accessed words or “Ra + 4.”

8.7.22. SWI.gp (GP-implied Store Word Immediate)

Type: 32-bit Baseline Version 2

Format:

31	30	25	24	20	19	17	16	0
0	HWGP 011110	Rt		111 SWI			imm17s	

Syntax: SWI.gp Rt, [+ (imm17s << 2)]

(imm17s is a word offset. In assembly programming, always write a byte offset.)

Purpose: Store a 32-bit word from a general register into a memory location.

Description: This instruction stores the 32-bit word in the general register Rt to the memory location. The memory address is specified by the implied GP register (R29) plus a sign-extended (imm17s << 2) value. The (imm17s << 2) value covers a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

```

Vaddr = R29 + Sign_Extend(imm17s << 2);
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_Status = Page_Exception(Attributes, PSW.POM, STORE);
If (Excep_Status == NO_EXCEPTION) {
    Store_Memory(PAddr, WORD, Attributes, Rt(31, 0));
} else {
    Generate_Exception(Excep_Status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error.

Privilege level: All

Note:

The information contained herein is the exclusive property of Andes Technology Co. and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Andes Technology Corporation.

8.8. 16-bit and 32-bit Baseline Version 3 Instructions

The following Baseline Version 3 instructions are added to improve code density and TLS implementation efficiency. Some of these instructions can also be classified as V3m instructions.

Mnemonic	Instruction	V3m instruction	16 or 32 bit
ADD_SLLI	Addition with Shift Left Logical Immediate		32-bit
ADD_SRRI	Addition with Shift Right Logical Immediate		32-bit
ADDRI36.SP	Add Immediate to Register with Implied Stack Register	Yes	16-bit
ADD5.PC	Add with Implied PC		16-bit
AND_SLLI	Logical And with Shift Left Logical Immediate		32-bit
AND_SRRI	Logical And with Shift Right Logical Immediate		32-bit
AND33	Bit-wise Logical And	Yes	16-bit
BITC	Bit Clear		32-bit
BITCI	Bit Clear Immediate		32-bit
BEQC	Branch on Equal Constant	Yes	32-bit
BNEC	Branch on Not Equal Constant	Yes	32-bit
BMASKI33	Bit Mask Immediate	Yes	16-bit
CCTL L1D_WBALL	L1D Cache Writeback All		32-bit
FEXTI33	Field Extraction Immediate	Yes	16-bit
JRALNEZ	Jump Register and Link on Not Equal Zero		32-bit
JRNEZ	Jump Register on Not Equal Zero		32-bit
LWI45.FE	Load Word Immediate with Implied Function Entry Point (R8)	Yes	16-bit
MOVD44	Move Double Registers	Yes	16-bit
MOVPI45	Move Positive Immediate	Yes	16-bit
MUL33	Multiply Word to Register	Yes	16-bit
NEG33	Negative	Yes	16-bit
NOT33	Logical Not	Yes	16-bit
OR_SLLI	Logical Or with Shift Left Logical Immediate		32-bit

Mnemonic	Instruction	V3m instruction	16 or 32 bit
OR_SRLI	Logical Or with Shift Right Logical Immediate		32-bit
OR33	Bit-wise Logical Or	Yes	16-bit
POP25	Pop Multiple Words from Stack and Return	Yes	16-bit
PUSH25	Push Multiple Words to Stack and Set Function Entry Point (R8)	Yes	16-bit
SUB_SLLI	Subtraction with Shift Left Logical Immediate		32-bit
SUB_SRLI	Subtraction with Shift Right Logical Immediate		32-bit
XOR_SLLI	Logical Exclusive Or with Shift Left Logical Immediate		32-bit
XOR_SRLI	Logical Exclusive Or with Shift Right Logical Immediate		32-bit
XOR33	Bit-wise Logical Exclusive Or	Yes	16-bit

8.8.1. ADD_SLLI (Addition with Shift Left Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		sh		ADD_SLLI 00000		

Syntax: ADD_SLLI Rt, Ra, Rb, sh

ADD Rt, Ra, Rb (sh=0)

Purpose: Add the content of a register and a logical shift left register value.

Description: This instruction adds the content of Ra and a logical shift left register value of Rb, then writes the result to Rt. The shift amount is specified by sh. .

Operations:

$Rt = Ra + (Rb \ll sh);$

Exceptions: None

Privilege level: All

Note:

8.8.2. ADD_SRLI (Addition with Shift Right Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		sh		ADD_SRLI 11100		

Syntax: ADD_SRLI Rt, Ra, Rb, sh

ADD Rt, Ra, Rb (sh=0)

Purpose: Add the content of a register and a logical shift right register value.

Description: This instruction adds the content of Ra and a logical shift right register value of Rb, then writes the result to Rt. The shift amount is specified by sh.

Operations:

$Rt = Ra + (Rb \gg sh);$

Exceptions: None

Privilege level: All

Note:

8.8.3. ADDR136.SP (Add Immediate to Register with Implied Stack Pointer)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	6	5	0
1	ADDR136.SP 011000		Rt3		imm6u	

Syntax: ADDR136.SP Rt3, imm8u

where the imm8u is the immediate value to be added to SP and its allowed values are multiple of 4 in the range of 0-252. The mapping equation of imm8u and imm6u is:

$$\text{imm8u} = \text{imm6u} \ll 2$$

Purpose: Add the content of the implied SP (R31) register and an unsigned constant.

Description: This instruction adds the content of SP (R29) and the zero-extended imm8u, then writes the result to Rt.

Operations:

$$\text{Rt3} = \text{R31} + \text{imm6u} \ll 2;$$

Exceptions: None

Privilege level: All

Note:

8.8.4. ADD5.PC (Add with Implied PC)

Type: 16-bit Baseline Version 3

Format:

15	14	5	4	0
1	ADD5.PC 1011101101			Rt5

Syntax: ADD5.PC Rt5

Purpose: Add the content of the implied PC (Program Counter) register and that of a general register.

Description: This instruction adds the contents of PC and Rt, then writes the result to Rt.

Operations:

$Rt5 = PC + Rt5;$

Exceptions: None

Privilege level: All

Note:

8.8.5. AND_SLLI (Logical And with Shift Left Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt	Ra		Rb		sh		AND_SLLI 00010			

Syntax: AND_SLLI Rt, Ra, Rb, sh

AND Rt, Ra, Rb (sh=0)

Purpose: Perform a bit-wise logical AND operation on the content of a register and a logically left-shifted register value.

Description: This instruction performs a bit-wise logical AND operation on the content of Ra and a logically left-shifted register value of Rb, then writes the result to Rt. The shift amount is specified by sh.

Operations:

$Rt = Ra \& (Rb << sh);$

Exceptions: None

Privilege level: All

Note:

8.8.6. AND_SRLI (Logical And with Shift Right Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		sh		AND_SRLI 11110		

Syntax: AND_SRLI Rt, Ra, Rb, sh

AND Rt, Ra, Rb (sh=0)

Purpose: Perform a bit-wise logical AND operation on the content of a register and a logically right-shifted register value.

Description: This instruction performs a bit-wise logical AND operation on the content of Ra and a logically right-shifted register value of Rb, then writes the result to Rt. The shift amount is specified by sh.

Operations:

Rt = Ra & (Rb >> sh);

Exceptions: None

Privilege level: All

Note:

8.8.7. AND33 (Bit-wise Logical And)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	6	5	3	2	0
1	MISC33 111111		Rt3		Ra3		AND33 110	

Syntax: AND33 Rt3, Ra3

32-bit Equivalent: AND 3T5(Rt3), 3T5(Rt3), 3T5(Ra3)

Purpose: Perform a bit-wise logical AND operation on the contents of two registers.

Description: This instruction uses a bit-wise logical AND operation to combine the content of Ra with that of Rt, then writes the result Rt.

Operations:

Rt = Ra & Rb;

Exceptions: None

Privilege level: All

Note:

8.8.8. BEQC (Branch on Equal Constant)

Type: 32-bit Baseline Version 3(m)

Format:

31	30	25	24	20	19	18	8	7	0
0	BR3 101101	Rt		BEQC 0		imm11s		imm8s	

Syntax: BEQC Rt, imm11s, imm8s

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with a constant specified by imm11s.

Description: If the content of Rt is equal to the sign-extended (imm11s) value, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm8s << 1) value. The branch range is ± 256 bytes.

Operations:

```
if (Rt == SE(imm11s)) {
  PC = PC + SE(imm8s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.8.9. BNEC (Branch on Not Equal Constant)

Type: 32-bit Baseline Version 3(m)

Format:

31	30	25	24	20	19	18	8	7	0
0	BR3 101101	Rt			BNEC 1	imm11s			imm8s

Syntax: BNEC Rt, imm11s, imm8s

Purpose: Perform conditional PC-relative branching based on the result of comparing the content of a register with a constant specified by imm11s.

Description: If the content of Rt is not equal to the sign-extended (imm11s) value, this instruction branches to the target address calculated by adding the current instruction address and the sign-extended (imm8s << 1) value. The branch range is ± 256 bytes.

Operations:

```
if (Rt != SE(imm11s)) {
    PC = PC + SE(imm8s << 1);
}
```

Exceptions: None

Privilege level: All

Note: The assembled/disassembled instruction format displayed by tools may be different from the encoding syntax shown here. Please consult "Andes Programming Guide" to get the correct meaning of the displayed syntax.

8.8.10. BITC (Bit Clear)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		00000		BITC 10010		

Syntax: BITC Rt, Ra, Rb

Purpose: Clear some bits of a register.

Description: This instruction performs a bit-wise AND operation on the content of Ra and the complement of Rb, then writes the result to Rt.

Operations:

$Rt = Ra \And (\sim Rb);$

Exceptions: None

Privilege level: All

Note:

8.8.11. BITCI (Bit Clear Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	0
0	BITCI 110011	Rt		Ra				imm15u

Syntax: BITCI Rt, Ra, imm15u

Purpose: Clear low order bits of a register.

Description: This instruction performs a bit-wise AND operation on the content of Ra and the complement of zero-extended imm15u, then writes the result to Rt.

Operations:

$Rt = Ra \& (\sim(ZE(imm15u)))$;

Exceptions: None

Privilege level: All

Note:

8.8.12. BMSKI33 (Bit Mask Immediate)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	6	5	3	2	0
1	001011		Rt3		imm3u		BMSKI33	110

Syntax: BMSKI33 Rt3, imm3u

32-bit Equivalent: ANDI 3T5(Rt3), 3T5(Rt3), # 2^{imm3u}

Purpose: Extract a bit from a register.

Description: This instruction extracts a bit from a register. The constant imm3u specifies the location of the extracted bit in Rt.

Operations:

$$Rt = Rt \& 2^{\text{imm3u}}$$

Exceptions: None

Privilege level: All

Note:

8.8.13. CCTL L1D_WBALL (L1D Cache Writeback All)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	11	10	9	5	4	0
0	MISC 110010	00000	00000		0000	level	L1D_WBALL 01111		CCTL 00001				

Syntax: CCTL L1D_WBALL, level

Purpose: Perform a writeback operation on processor L1D cache.

Description: This instruction writebacks the L1D cache blocks if the cache block state is valid and dirty in a write-back cache. It does not affect the lock state of the cache block. The multi-level cache management behavior is the same as L1D_VA_WB of the CCTL instruction.

Exceptions: Imprecise bus error exception.

Privilege level: All

Note:

8.8.14. FEXTI33 (Field Extraction Immediate)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	6	5	3	2	0
1	001011		Rt3		imm3u		FEXTI33 111	

Syntax: FEXTI33 Rt3, imm3u

32-bit Equivalent: ANDI 3T5(Rt3), 3T5(Rt3), # $2^{\text{imm3u}+1}-1$

Purpose: Extract the lowest field of certain (n) bits from a register.

Description: This instruction extracts the lowest field of certain (n) bits from a register where n is specified by the value of imm3u plus 1 .

Operations:

$$Rt = Rt \& (2^{\text{imm3u}+1}-1)$$

Exceptions: None

Privilege level: All

Note:

8.8.15. JRALNEZ (Jump Register and Link on Not Equal Zero)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	8	7	5	4	0
0	JREG 100101	Rt		00000		Rb		DT/IT 00	000		JRALNEZ 00011			

Syntax: JRALNEZ Rt, Rb

JRALNEZ Rb (implied Rt == R30, Link Pointer register)

Purpose: Make a conditional function call to an instruction address stored in a register.

Description: If the content of Rb is equal to zero, then this instruction behaves as a NOP. Otherwise, this instruction stores its sequential PC to Rt and then branches to an instruction address stored in Rb.

Operations:

```
If (Rb != 0) {
  Rt = PC + 4;
  PC = Rb;
} Else {
  PC = PC + 4;
}
```

Exceptions: None

Privilege level: All

Note:

8.8.16. JRNEZ (Jump Register on Not Equal Zero)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	15	14	10	9	8	7	6	5	4	0
0	JREG 100101	0000000000		Rb		DT/IT 00		00		JR hint 0		JRNEZ 00010	

Syntax: JRNEZ Rb

Purpose: Conditionally branch to an instruction address stored in a register.

Description: If the content of Rb is equal to zero, then this instruction behaves as a NOP.

Otherwise, this instruction branches to an instruction address stored in Rb.

Operations:

```
If (Rb != 0)
  PC = Rb;
Else
  PC = PC + 4;
```

Exceptions: None

Privilege level: All

Note:

8.8.17. LWI45.FE (Load Word Immediate with Implied Function Entry Point)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	5	4	0
1	LWI45.FE 011001		Rt4		imm5u	

Syntax: lwi45.fe Rt4, [imm7n]

where the imm7n is the immediate value to be added to the function entry point (R8) and its allowed values are multiple of 4 in the range from -128 to -4. The mapping equation of imm7n and imm5u is:

$$\text{imm7n} = -((32 - \text{imm5u}) \ll 2)$$

32-bit Equivalent: LWI 4T5(Rt4), [R8, (imm5u – 32) << 2]

Purpose: Load a 32-bit word from a memory address based on the function entry point (R8) into a general register.

Description: This instruction loads a word from a memory address into the general register Rt. The memory address is specified by R8 + (imm5u – 32) << 2.

Operations:

```

Vaddr = R8 + Sign_Extend((imm5u - 32) << 2);
if (!Word_Alignment(Vaddr)) {
    Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
If (Excep_status == NO_EXCEPTION) {
    Wdata(31, 0) = Load_Memory(PAddr, WORD, Attributes);
    Rt = Wdata(31, 0);
} else {
    Generate_Exception(Excep_status);
}

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

Privilege level: All

Implementation Note:

The address offset value of this instruction is negative ranging from -32 to -1. The term “imm5u – 32” means negative part of imm6s. In 2’s complement, the 32-bit offset value can be generated by

$$\text{“imm5u - 32”} = \text{SE(concat (1`b1, imm5u))}$$

8.8.18. MOVD44 (Move Double Registers)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	8	7	4	3	0
1	MOVD44 1111101		Rt5e		Ra5e	

Syntax: MOVD44 Rt1, Ra1

where Rt1 is the first destination register and its allowed value is an even number in the range between 0 to 31. The mapping equation of Rt and Rt5e is:

$$Rt1 = Rt5e * 2$$

Ra1 is the first source register and its allowed value is an even number in the range between 0 to 31. The mapping equation of Ra and Ra5e is:

$$Ra1 = Ra5e * 2$$

Purpose: Move the content of two consecutive registers to another two consecutive registers.

Description: This instruction moves the content of two consecutive registers to another two consecutive registers. The indices of the two source registers (Ra1, Ra2) and two destination registers (Rt1, Rt2) are computed as shown in the following table:

Register	Index	Implementation Note of Register Index
Ra1	$Ra5e * 2$	concat(Ra5e,1`b0)
Ra2	$Ra5e * 2 + 1$	concat(Ra5e,1`b1)
Rt1	$Rt5e * 2$	concat(Rt5e,1`b0)
Rt2	$Rt5e * 2 + 1$	concat(Rt5e,1`b1)

Operations:

Rt1 = Ra1; Rt2 = Ra2;

Exceptions: None



Privilege level: All

Note:

8.8.19. MOVPI45 (Move Positive Immediate)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	5	4	0
1	MOVPI45 111101		Rt4		imm5u	

Syntax: MOVPI45 Rt4, imm6u

where the imm6u is the immediate value to be moved into the destination register and its allowed values are in the range of 16-47. The mapping equation of imm6u and imm5u is:

$$\text{imm6u} = \text{imm5u} + 16$$

Purpose: Move a constant value (ranging from 16 to 47) into a general-purpose register.

Description: This instruction moves the value of “imm5u + 16” into Rt.

Operations:

$$\text{Rt4} = \text{imm5u} + 16;$$

Exceptions: None

Privilege level: All

Implementation Note:

The 32-bit value of “imm5u + 16” can be generated by

```

If (imm5u[4] == 0)
    ZE(concat(2`b01, imm5u(3, 0)));
Else
    ZE(concat(2`b10, imm5u(3, 0)));

```

8.8.20. MUL33 (Multiply Word to Register)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	6	5	3	2	0
1	MISC33 111111		Rt3		Ra3		MUL33 100	

Syntax: MUL33 Rt3, Ra3

32-bit Equivalent: MUL 3T5(Rt3), 3T5(Rt3), 3T5(Rb3)

MUL 3T5(Rt3), 3T5(Ra3), 3T5(Rt3)

Purpose: Multiply the contents of two registers and write the result to a register.

Description: This instruction multiplies the 32-bit content of Ra with that of Rb and writes the lower 32-bit of the multiplication result to Rt. The contents of Ra and Rb can be signed or unsigned numbers.

Operations:

```
Mresult = Ra * Rb;
Rt = Mresult(31, 0);
```

Exceptions: None

Privilege level: All

Note:

8.8.21. NEG33 (Negative)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	6	5	3	2	0
1	MISC33 111111		Rt3		Ra3		NEG33 010	

Syntax: NEG33 Rt3, Ra3

32-bit Equivalent: SUBRI 3T5(Rt3), 3T5(Ra3), 0

Purpose: Perform a negation operation to a register and write the result to a register.

Description: This instruction negates the content of Ra and writes the result into Rt.

Operations:

Rt = -Ra;

Exceptions: None

Privilege level: All

Note:

8.8.22. NOT33 (Not)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	6	5	3	2	0
1	MISC33 111111		Rt3		Ra3		NOT33 011	

Syntax: NOT33 Rt3, Ra3

32-bit Equivalent: NOR 3T5(Rt3), 3T5(Ra3), 3T5(Ra3)

Purpose: Perform a bit-wise NOT operation to the content of a register and write the result to a register.

Description: This instruction performs a bit-wise NOT operation to the content of Ra and writes the result into Rt.

Operations:

Rt = ~Ra;

Exceptions: None

Privilege level: All

Note:

8.8.23. OR_SLLI (Logical Or with Shift Left Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		sh		OR_SLLI 00100		

Syntax: OR_SLLI Rt, Ra, Rb, sh
 OR Rt, Ra, Rb(sh=0)

Purpose: Perform a bit-wise logical OR operation on the content of a register and a logically left-shifted register value.

Description: This instruction performs a bit-wise logical OR operation on the content of Ra and a logically left-shifted register value of Rb, then writes the result to Rt. The shift amount is specified by sh.

Operations:

$Rt = Ra \mid (Rb \ll sh);$

Exceptions: None

Privilege level: All

Note:

8.8.24. OR_SRLI (Logical Or with Shift Right Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		sh		OR_SRLI 10101		

Syntax: OR_SRLI Rt, Ra, Rb, sh

OR Rt, Ra, Rb (sh=0)

Purpose: Perform a bit-wise logical OR operation on the content of a register and a logically right-shifted register value.

Description: This instruction performs a bit-wise logical OR operation on the content of Ra and a logically right-shifted register value of Rb, then writes the result to Rt. The shift amount is specified by sh.

Operations:

$Rt = Ra \mid (Rb >> sh);$

Exceptions: None

Privilege level: All

Note:

8.8.25. OR33 (Bit-wise Logical Or)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	6	5	3	2	0
1	MISC33 111111		Rt3		Ra3		OR33 111	

Syntax: OR33 Rt3, Ra3

32-bit Equivalent: OR 3T5(Rt3), 3T5(Rt3), 3T5(Ra3)

Purpose: Perform a bit-wise logical OR operation on the contents of two registers.

Description: This instruction perform a bit-wise logical OR operation on the contents of Ra and Rt, and then writes the result into Rt.

Operations:

$Rt = Ra \mid Rt;$

Exceptions: None

Privilege level: All

Note:

8.8.26. POP25 (Pop Multiple Words from Stack and Return)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	7	6	5	4	0
1	POP25 1111001		Re			imm5u

Syntax: POP25 Re, imm8u

where the imm8u is the immediate value to be added to SP for starting address calculation and its allowed values are multiple of 8 in the range from 0-248. The mapping equation of imm8u and imm5u is:

$$\text{imm8u} = \text{imm5u} \ll 3$$

32-bit Equivalent:

Purpose: Pop multiple 32-bit words from stack, adjust stack pointer, and return

Description: This instruction performs three operations –

Operation 1: Load multiple 32-bit words from sequential memory locations specified by the stack pointer (R31) and imm5u into a list of GPRs specified as follows:

Re(Value)	Re(Syntax)	<GPRs list>	Starting address
0	R6	R6, fp, gp, lp	sp + imm5u << 3
1	R8	R6, R7, R8, fp, gp, lp	sp + imm5u << 3
2	R10	R6, R7, R8, R9, R10, fp, gp, lp	sp + imm5u << 3
3	R14	R6, R7, R8, R9, R10, R11, R12, R13, R14, fp, gp, lp	sp + imm5u << 3

Operation 2: Modify stack pointer according to Re and (imm5u << 3). The final stack pointer value will be:

Re(Value)	Re(Syntax)	Stack Pointer
0	R6	$sp = sp + imm5u << 3 + 16$
1	R8	$sp = sp + imm5u << 3 + 24$
2	R10	$sp = sp + imm5u << 3 + 32$
3	R14	$sp = sp + imm5u << 3 + 48$

Operation 3: Return

$pc = lp$

This instruction can only handle word aligned memory access.

Operations:

Operation 1:

```

TNReg = Count_Registers(GPRs_List);
VA = sp + imm5u << 3;
if (!word-aligned(VA)) {
    Generate_Exception(Data_Alignment_Check);
}
for (i = 0 to 31) {
    if (GPRs_List[i] == 1) {
        (PA, Attributes) = Address_Translation(VA, PSW.DT);
        Excep_Status = Page_Exception(Attributes, PSW.POM, LOAD);
        If (Excep_Status == NO_EXCEPTION) {
            Ri = Load_Memory(PA, Word, Attributes);
            VA = VA + 4;
        } else {
            Generate_Exception(Excep_Status);
        }
    }
}

```

Operation 2:

```

sp = sp + imm5u << 3;
sp = sp + (TNReg * 4);

```

Operation 3:

```

pc = lp;

```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

- On an exception event, the value left in sp is unchanged.
- On an exception event, UNPREDICTABLE values are left in destination

registers

Interruption: When an interrupt comes before operation 2, the instruction may be interruptible depending on implementation. When an interrupt comes upon or after operation 2, the instruction is non-interruptible.

Privilege level: All

Note:

8.8.27. PUSH25 (Push Multiple Words to Stack and Set FE)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	7	6	5	4	0
1	PUSH25 11111000		Re			imm5u

Syntax: PUSH25 Re, imm8u

where the imm8u is the immediate value to be subtracted with SP for the SP adjustment and its allowed values are multiple of 8 in the range from 0 to 248. The mapping equation of imm8u and imm5u is:

$$\text{imm8u} = \text{imm5u} \ll 3$$

32-bit Equivalent:

Purpose: Push multiple 32-bit words to stack, adjust stack pointer, and move pc value to R8.

Description: This instruction performs three operations –

Operation 1: This operation is exactly the same as smw.adm R6, [sp], Re, #0xe

- A. Store multiple 32-bit words from a list of GPRs specified as follows into sequential memory locations specified by the stack pointer (R31):

Re(Value)	Re(Syntax)	<GPRs list>
0	R6	R6, fp, gp, lp
1	R8	R6, R7, R8, fp, gp, lp
2	R10	R6, R7, R8, R9, R10, fp, gp, lp
3	R14	R6, R7, R8, R9, R10, R11, R12, R13, R14, fp, gp, lp

- B. Modify stack pointer value according to register counts in <GPRs list>

Re(Value)	Re(Syntax)	Stack Pointer
0	R6	sp = sp - 16
1	R8	sp = sp - 24
2	R10	sp = sp - 32
3	R14	sp = sp - 48

Operation 2: sp = sp - imm5u<<3

Operation 3: If (Re >= 1) R8 = concat (PC(31,2), 2`b0)

This instruction can only handle word aligned memory access.

Operations:

Operation 1A:

```

TNReg = Count_Registers(GPRs_List);
VA = sp - (TNReg * 4)
if (!word_aligned(VA)) {
    Generate_Exception(Data_Alignment_Check);
}
for (i = 0 to 31) {
    if (GPRs_List[i] == 1) {
        (PA, Attributes) = Address_Translation(VA, PSW.DT);
        Excep_status = Page_Exception(Attributes, PSW.POM, LOAD);
        if (Excep_status == NO_EXCEPTION) {
            Ri = Store_Memory(PA, Word, Attributes);
            VA = VA + 4;
        } else {
            Generate_Exception(Excep_status);
        }
    }
}

```

Operation 1B:

```
sp = sp - (TNReg * 4);
```

Operation 2:

```
sp = sp - imm5u << 3;
```

Operation 3:

```
if (Re >= 1)
    R8 = concat (PC(31, 2), 2`b0);
```

Exceptions: TLB fill, Non-leaf PTE not present, Leaf PTE not present, Read protection, Page modified, Access bit, TLB VLPT miss, Machine error, Data alignment check.

- The value left in sp is unchanged on an exception event as if this instruction has not been executed.

Interruption: When an interrupt comes before operation 1B, the instruction may be interruptible depending on implementation. When an interrupt comes during or after operation 1B, the instruction is non-interruptible.

Privilege level: All

Note:

8.8.28. SUB_SLLI (Subtraction with Shift Left Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		sh		SUB_SLLI 00001		

Syntax: SUB_SLLI Rt, Ra, Rb, sh

 SUB Rt, Ra, Rb (sh=0)

Purpose: Subtract the content of a register and a logically left-shifted register value.

Description: This instruction subtracts a logically left-shifted register value of Rb from the content of Ra, then writes the result to Rt. The shift amount is specified by sh.

Operations:

$Rt = Ra - (Rb \ll sh);$

Exceptions: None

Privilege level: All

Note:

8.8.29. SUB_SRLI (Subtraction with Shift Right Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		sh		SUB_SRLI 11101		

Syntax: SUB_SRLI Rt, Ra, Rb, sh
 SUB Rt, Ra, Rb (sh=0)

Purpose: Subtract the content of a register and a logically right-shifted register value.

Description: This instruction subtracts a logically right-shifted register value of Rb from the content of Ra, then writes the result to Rt. The shift amount is specified by sh.

Operations:

$$Rt = Ra - (Rb \gg sh);$$

Exceptions: None

Privilege level: All

Note:

8.8.30. XOR_SLLI (Logical Exclusive Or with Shift Left Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt		Ra		Rb		sh		XOR_SLLI 00011		

Syntax: XOR_SLLI Rt, Ra, Rb, sh
 XOR Rt, Ra, Rb (sh=0)

Purpose: Perform a bit-wise logical XOR operation on the content of a register and a logically left-shifted register value.

Description: This instruction performs a bit-wise logical XOR operation on the content of Ra and a logically left-shifted register value of Rb, then writes the result to Rt. The shift amount is specified by sh.

Operations:

`Rt = Ra ^ (Rb << sh);`

Exceptions: None

Privilege level: All

Note:

8.8.31. XOR_SRLI (Logical Exclusive Or with Shift Right Logical Immediate)

Type: 32-bit Baseline Version 3

Format:

31	30	25	24	20	19	15	14	10	9	5	4	0
0	ALU_1 100000	Rt	Ra	Rb	sh	XOR_SRLI 11111						

Syntax: XOR_SRLI Rt, Ra, Rb, sh
 XOR Rt, Ra, Rb (sh=0)

Purpose: Perform a bit-wise logical XOR operation on the content of a register and a logically right-shifted register value.

Description: This instruction performs a bit-wise logical XOR operation on the content of Ra and a logically right-shifted register value of Rb, then writes the result to Rt. The shift amount is specified by sh.

Operations:

$Rt = Ra \wedge (Rb >> sh);$

Exceptions: None

Privilege level: All

Note:

8.8.32. XOR33 (Bit-wise Logical Exclusive Or)

Type: 16-bit Baseline Version 3(m)

Format:

15	14	9	8	6	5	3	2	0
1	MISC33 111111		Rt3		Ra3		xor33 101	

Syntax: XOR33 Rt3, Ra3

32-bit Equivalent: XOR 3T5(Rt3), 3T5(Rt3), 3T5(Ra3)

Purpose: Perform a bit-wise logical Exclusive OR operation on the content of two registers.

Description: This instruction performs a bit-wise logical Exclusive OR operation on the content of Ra and Rt, and then writes the result to Rt.

Operations:

$Rt = Ra \wedge Rt;$

Exceptions: None

Privilege level: All

Note:

9. Instruction Latency for AndesCore Implementations

This chapter lists the instruction latency information for the AndesCore families in the following sections:

- 
- The logo consists of a blue rounded rectangle containing the words "Official" and "Release". "Official" is in a larger, bold font, and "Release" is in a smaller font below it.
- 9.1 N12 Family Implementation on page 384
 - 9.2 N10 Family Implementation on page 389
 - 9.3 N8 Family Implementation on page 397

9.1. N12 Family Implementation

9.1.1. Instruction Latency due to Resource Dependency

This section describes the AndesCore N12 instruction latency between a producer instruction and a corresponding consumer instruction. This information is useful for compiler optimization.

Terminology

- Producer: An instruction that produces a new register state.
- Consumer: An instruction that consumes the new register state produced by a producer.
- Latency: The minimum number of cycles between the completion of a producer and that of a consumer. Assuming a producer and its corresponding consumer cannot complete at the same time, the smallest possible latency is 1.
- Bubble: The minimum number of extra cycles that exceeds the smallest possible latency (i.e. 1) between the completion of a producer and that of a consumer. Thus it is equal to $(\text{latency} - 1)$.

Producer/Consumer Instruction Group

- Producer/Consumer Instruction Group

Producer / Consumer Group	Instructions	Note
ALU	ADDI, SUBRI, ANDI, ORI, XORI, SLTI, SLTSI, MOVI, SETHI, ADD, SUB, AND, OR, NOR, XOR, SLT, SLTS, SVA, SVS, SEB, SHE, ZEB, ZEH, WSBH, CMOVZ, CMOVN, SLLI, SRLI, SRAI, ROTRI, SLL, SRL, SRA, ROTR	Result in general register R; sources from general register R
MUL	MUL	Result in general register R
M2D	MULTS64, MULT64, MULT32, MADD64, MADD64, MSUBS64, MSUB64, MADD32, MSUB32	Result in accumulator register D

- Unique Producer Instruction Group

Producer Group	Instructions	Note
LD_D	LWI[.bi], LHI[.bi], LHSI[.bi], LBI[.bi], LBSI[.bi], LW[.bi], LH[.bi], LHS[.bi], LB[.bi], LBS[.bi], LWUP, LLW	Load instructions which have the data register as the produced state
LD_A	LWI.bi, LHI.bi, LHSI.bi, LBI.bi, LBSI.bi, LW.bi, LH.bi, LHS.bi, LB.bi, LBS.bi	Load instructions which have the base address register as the produced state
SCW	SCW	
MISC_E2	MFUSR, MFSR, JAL, JRAL, BGEZAL, BLTZAL	
DIV	DIV, DIVS	

- Unique Consumer Instruction Group

Consumer Group	Instructions	Note
ST_D	SWI[.bi], SHI[.bi], SBI[.bi], SW[.bi], SH[.bi], SB[.bi], SWUP, SCW	Store instructions which have the data register as the consumed state
MEM_A	LWI[.bi], LHI[.bi], LHSI[.bi], LBI[.bi], LBSI[.bi], LW[.bi], LH[.bi], LHS[.bi], LB[.bi], LBS[.bi], LWUP, LLW, LMW, SWI[.bi], SHI[.bi], SBI[.bi], SW[.bi], SH[.bi], SB[.bi], SWUP, SCW, SMW, DPREF, DPREFI	LD/ST instructions which have the base address register as the consumed state
BR	JR, RET, JRAL, BEQ, BNE, BEQZ, BNEZ, BGEZ, BLTZ, BGTZ, BLEZ, BGEZAL, BLTZAL	
MFUSR	MFUSR	
MISC_E1	TLBOP TRD, TLBOP TWR, TLBOP RWR, TLBOP RWLK, TLBOP UNLK, TLBOP PB, TLBOP INV, CCTL, ISYNC, MTSR, MTUSR	

In AndesCore N12, a producer and its corresponding consumer normally have a latency of 1.

Cases that need special attention or deviate from this general rule of thumb are described below.

No	Producer	Consumer	Latency
Dependency on general register Rx			
1	LD_D	ALU, ST_D, BR	2
2		MUL, M2D, MEM_A, MISC_E1	3
3	MUL	ALU, ST_D, BR	2
4		MUL, M2D, MEM_A, MISC_E1	3
5a	ALU, MISC_E2	MUL, M2D, MEM_A, MISC_E1	2
5b*	LD_A	MUL, M2D, MEM_A, MISC_E1	2
6	SCW	ALU, ST_D, BR	3
7		MUL, M2D, MEM_A, MISC_E1	2
Dependency on accumulator register Dx or multiplication E1 resource			
8	M2D	M2D, MFUSR	2
9	DIV	All Dx consumer	Variable (32 – CLZ(ra)) + 3
For the following LMW related producers, assumes LMW loading N registers:			
Aligned LMW			
Dependency on the highest-numbered register in register list for LMW.i or			
Dependency on the lowest-numbered register in register list for LMW.d			
10	LMW.i or LMW.d base not in list,	ALU, ST_D, BR	N+1
11	LMW.i or LMW.d base is the dependency reg, LMW.im or LWM.dm	MUL, M2D, MEM_A, MISC_E1	N+2
Dependency NOT on the highest-numbered register in register list for LMW.i or			
Dependency NOT on the lowest-numbered register in register list for LMW.d			
12	LMW.i or LMW.d base not in list, LMW.im	ALU, ST_D, BR	N
13		MUL, M2D, MEM_A, MISC_E1	N+1

No	Producer	Consumer	Latency
Fixed latency despite dependency relationship			
14	LMW.i base in list, but not highest-numbered reg, LMW.d base in list, but not lowest-numbered reg	ALL	N+3
Un-Aligned LMW			
15	Rule 9-13	Rule 9-13	(Rule 9-13 latency)+1

*Note: Rule 5b exists when the configuration flag “NDS_POSTWRITE_E1_BYPASS” is turned off. Turning on configuration flag “NDS_POSTWRITE_E1_BYPASS” makes all LD_A consumers have a latency of 1.

The following instructions have a fixed latency without considering any resource dependency relationships.

Instruction	Latency
RET (correct target prediction)	3
DSB	5
ISB, IRET	10
DIV, DIVS	Variable (3 – 34) [i.e. 3+floor(log2(abs(Rb)))]
TLBOP Invalidate VA	10
TLBOP Invalidate All	98
DPREF/DPREFI (miss dcache)	4

9.1.2. Cycle Penalty due to N12 Pipeline Control Mishaps Recovery

This section describes the AndesCore N12 pipeline execution cycle penalties caused by recovering certain unlucky events.

Event Type	Penalty (Bubble)
un-aligned 32-bit instruction fetch after pipeline start/flush	1
branch mis-prediction	5/6
uTLB miss/MTLB hit on small page PTE	4
uTLB miss/MTLB hit on large page PTE	6
uTLB miss/MTLB miss/HPTWK prefetch buffer hit (no large page in use)	7
uTLB miss/MTLB miss/HPTWK prefetch buffer hit (with large page in use)	9

9.2. N9/N10 Family Implementation

9.2.1. Dependency-related Instruction Latency

This section describes the AndesCore N10 instruction latency between a producer instruction and a corresponding consumer instruction. This information is useful for compiler optimization.

Terminology

- Producer: An instruction that produces a new register state.
- Consumer: An instruction that consumes the new register state produced by a producer.
- Latency: The minimum number of cycles between the completion of a producer and that of a consumer. Assuming a producer and its corresponding consumer cannot complete at the same time, the smallest possible latency is 1.
- Bubble: The minimum number of extra cycles that exceeds the smallest possible latency (i.e. 1) between the completion of a producer and that of a consumer. Thus it is equal to $(\text{latency} - 1)$.

Producer/Consumer Instruction Groups

- Producer/Consumer Instruction Group

Producer / Consumer Group	Instructions	Note
ALU	ADDI, SUBRI, ANDI, ORI, XORI, SLTI, SLTSI, MOVI, SETHI, ADD, SUB, AND, OR, NOR, XOR, SLT, SLTS, SVA, SVS, SEB, SHE, ZEB, ZEH, WSBH, CMOVZ, CMOVN, SLLI, SRRI, SRAI, ROTRI, SLL, SRL, SRA, ROTR, BITC, BITCI	Result in general register R; sources from general register R
MUL	MUL, MADDR32, MSUBR32, MULR64, MULSR64	Result in general register R
M2D	MULTS64, MULT64, MULT32, MADDS64, MADD64, MSUBS64, MSUB64, MADD32, MSUB32	Result in accumulator register D
MOVD_O	MOVD44	Odd Register of MOVD44

MOVD_E	MOVD44	Even Register of MOVD44
--------	--------	-------------------------

- Unique Producer Instruction Group

Producer Group	Instructions	Note
LD_D.bi	LWI.bi, LHI.bi, LHSI.bi, LBI.bi, LBSI.bi, LW.bi, LH.bi, LHS.bi, LB.bi, LBS.bi, LWUP, LLW	Load instructions which have the data register as the produced state
LD_D_!bi	LWI, LHI, LHSI, LBI, LBSI, LW, LH, LHS, LB, LBS, LWUP, LLW	Load instructions which have the data register as the produced state
MFUSR_GR	MFUSR GR ← D	Result in general register R
SCW	SCW	Generate success/failure status in general register R
DIV	DIV, DIVS, DIVR, DIVSR	
LMW_iHdL_nup	LMW.i with dependency on the highest-numbered register in register list, LMW.d with dependency on the lowest-numbered register in register list, and both have no base register update.	

- Unique Consumer Instruction Group

Consumer Group	Instructions	Note
ST_D_RI!bi	SWI, SHI, SBI	Store instructions which have the data register as the consumed state
ST_D_RR!bi	SW, SH, SB, SWUP, SCW	Store instructions which have the data register as the consumed state

Consumer Group	Instructions	Note
ST_D_Ribi	SWI.bi, SHI.bi, SBI.bi	Store instructions which have the data register as the consumed state
ST_D_RRbi	SW.bi, SH.bi, SB.bi	Store instructions which have the data register as the consumed state
MEM_A_!bi	LWI, LHI, LHSI, LBI, LBSI, LW, LH, LHS, LB, LBS, LWUP, LLW, LMW, SWI, SHI, SBI, SW, SH, SB, SWUP, SCW, SMW, DPREF, DPREFI	LD/ST instructions (non-bi form) which have the base address registers (Ra or Rb) as the consumed state
MEM_A_bi_Ra	LWI.bi, LHI.bi, LHSI.bi, LBI.bi, LBSI.bi, LW.bi, LH.bi, LHS.bi, LB.bi, LBS.bi, SWI.bi, SHI.bi, SBI.bi, SW.bi, SH.bi, SB.bi	LD/ST instructions (bi form) which have the base address register Ra as the consumed state
MEM_A_bi_Rb	LW.bi, LH.bi, LHS.bi, LB.bi, LBS.bi, SW.bi, SH.bi, SB.bi	LD/ST instructions (bi form) which have the base address register Rb as the consumed state
BR	JR, RET, JRAL, BEQ, BNE, BEQZ, BNEZ, BGEZ, BLTZ, BGTZ, BLEZ, BGEZAL, BLTZAL, BEQC, BNEC	
MISC	TLBOP TRD, TLBOP TWR, TLBOP RWR, TLBOP RWLK, TLBOP UNLK, TLBOP PB, TLBOP INV, CCTL, ISYNC, MTSR, MTUSR	Consumes general register R
ALU_SHIFT_Ra	ADD_SLLI, ADD_SRLI, AND_SLLI, AND_SRLI, OR_SLLI, OR_SRLI, SUB_SLLI, SUB_SRLI, XOR_SLLI, XOR_SRLI	Operand Ra of ALU_SHIFT instructions

Consumer Group	Instructions	Note
ALU_SHIFT_Rb	ADD_SLLI, ADD_SRRI, AND_SLLI, AND_SRRI, OR_SLLI, OR_SRRI, SUB_SLLI, SUB_SRRI, XOR_SLLI, XOR_SRRI	Operand Rb of ALU_SHIFT instructions
MAC_RaRb	MUL, MADDR32, MSUBR32, MULR64, MULSR64	Operand Ra or Rb of multiply instructions
MAC_Rt	MADDR32, MSUBR32, MULR64, MULSR64	Operand Rt of MAC instructions

In AndesCore N9/N10, a producer and its corresponding consumer normally have a latency of 1. Cases that need special attention or deviate from this general rule of thumb are described below.

No	Producer	Consumer	Latency	
			2R1W	3R2W
1	LD_D_!bi, MUL, MFUSR_GR, LMW_iHdL_nup*	MEM_A_!bi, MEM_A_bi_Ra, ALU, MAC_RaRb, BR, ALU_SHIFT_Rb, MISC, DIV, MOVD_E	2	2
2		MEM_A_bi_Rb, MOVD_O	1	2
3		ALU_SHIFT_Ra	1	1
4		ST_D_RI!bi	2	1
		ST_D_RR!bi	1	1
		ST_D_RIbi	2	1
5		ST_D_RRbi	2	1
6	SCW	MEM_A_!bi, MEM_A_bi_Ra, ALU, MUL, BR, MISC	3	3
7		MEM_A_bi_Rb	2	3
8		ST_D_!bi	2	2
9		ST_D.bi	3	2

* Note that this dependency latency does not include the fixed self-stalling latency described in the next section for LMW instructions.

9.2.2. Self-stall-related Instruction Latency

The following instructions have a fixed latency caused by self-stalling without considering any resource dependency relationships for all register file configurations.

Instruction / Instruction Category	Latency
RET (correct target prediction)	1
DSB	3
ISB, IRET	5
MUL (Slow config)	18
M2D (Slow config)	20

Instruction / Instruction Category	Latency
DIV, DIVS	Variable (4 – 35) [i.e. $4 + \text{floor}(\log_2(\text{abs}(Rb)))$]
LSMW1_A	N
LSMW2_A	N+1
LSMW1_U	N+1
LSMW2_U	N+2
TLBOP Invalidate VA	3
TLBOP Invalidate All	65

Note:

- (1) All LMW/SMW instructions here are loading N registers.
- (2) *LSMW1_A* denotes a LMW/SMW instruction *with no* base register update, and accessing word-aligned addresses.
- (3) *LSMW2_A* denotes a LMW/SMW instruction *with* base register update, and accessing word-aligned addresses.
- (4) *LSMW1_U* denotes a LMW/SMW instruction *with no* base register update, and accessing word-unaligned addresses.
- (5) *LSMW2_U* denotes a LMW/SMW instruction *with* base register update, and accessing word-unaligned addresses.
- (6) *M2D* indicates instructions in the instruction group described in the previous section.
- (7) *MUL* and *M2D* of the “fast configuration” have a latency of 1.

The following instructions have a fixed latency caused by self-stalling without considering any resource dependency relationships for the 2R1W register file configuration.

Instruction	Latency
Load.bi	2
Store(.bi) [R+R]	2

9.2.3. Cycle Penalty due to N9/N10 Pipeline Control Mishap Recover

This section describes the AndesCore N9/N10 pipeline execution cycle penalties caused by recovering certain unlucky events.

Event Type	Penalty (Bubble)
un-aligned 32-bit instruction fetch after pipeline start/flush	1
branch mis-prediction	2
uTLB miss/MTLB hit on small page PTE	4
uTLB miss/MTLB hit on large page PTE	6
uTLB miss/MTLB miss/HPTWK prefetch buffer hit (no large page in use)	7
uTLB miss/MTLB miss/HPTWK prefetch buffer hit (with large page in use)	9

9.2.4. Cycle Penalty due to Resource Contention

This section describes the cycle penalties, caused by resource contention, of instructions after a data prefetch instruction if the data prefetch instruction misses the cache. The following instructions incur additional cycle penalties if they follow the data prefetch instruction too closely.



Instruction
(cause resource contention with a previous missed DPREF)
LD/ST instructions
CCTL (D-type)
ISYNC
MSYNC

The number of cycle penalties depends on the current state of LSU which represents different levels of LSU resource contention.

LSU FSM State	Penalty (Bubble)
IDLE	0
FILL	2
DRAIN/WB	$M+1-N$
Others	1

Note:

- (1) M means the number of the words in a cache line.
- (2) N means the number of the instructions between the data prefetch instruction and any instruction in the above instruction table.
- (3) The cycle penalty of $M+1-N$ is the worst case scenario assuming that the N instructions in (2) do not cause pipeline stall.

9.3. N8 Family Implementation

9.3.1. Dependency-related Instruction Latency

This section describes the AndesCore N8 instruction latency between a producer instruction and a corresponding consumer instruction. This information is useful for compiler optimization.

Terminology

- Producer: An instruction that produces a new register state.
- Consumer: An instruction that consumes the new register state produced by a producer.
- Latency: The minimum number of cycles between the completion of a producer and that of a consumer. Assuming a producer and its corresponding consumer cannot complete at the same time, the smallest possible latency is 1.
- Bubble: The minimum number of extra cycles that exceeds the smallest possible latency (i.e. 1) between the completion of a producer and that of a consumer. Thus it is equal to $(\text{latency} - 1)$.

Producer/Consumer Instruction Groups

- Producer/Consumer Instruction Group

Producer /Consumer Group	Instructions	Note
ALU	ADDI, SUBRI, ANDI, ORI, XORI, SLTI, SLTSI, MOVI, SETHI, ADD, SUB, AND, OR, NOR, XOR, SLT, SLTS, SVA, SVS, SEB, SHE, ZEB, ZEH, WSBH, CMOVZ, CMOVN, SLLI, SRLI, SRAI, ROTRI, SLL, SRL, SRA, ROTR, BMSKI33, FEXTI33, MOVPI45, NEG33, NOT33	Result in general register R; sources from general register R
MUL	MUL	Result in general register R
MOVD_E	MOVD44	Even Register of MOVD44

Producer /Consumer Group	Instructions	Note
MOVD_O	MOVD44	Odd Register of MOVD44
PUSH25_SP_Re0	PUSH25 (Re = 0)	SP of push25
PUSH25_SP_Re!0	PUSH25 (Re > 0)	SP of push25
POP25_SP	POP25	SP of pop25

- Unique Producer Instruction Group

Producer Group	Instructions	Note
LD_D	LWI[.bi], LHI[.bi], LHSI[.bi], LBI[.bi], LBSI[.bi], LW[.bi], LH[.bi], LHS[.bi], LB[.bi], LBS[.bi]	Load instructions which have the data register as the produced state
LMW_iHdL_nup	LMW.i with dependency on the highest-numbered register in register list, LMW.d with dependency on the lowest-numbered register in register list, and both have no base register update.	
PUSH25_R8	PUSH25 (Re >= 1)	R8 of push25
POP25_D	POP25	

- Unique Consumer Instruction Group

Consumer Group	Instructions	Note
ST_D	SWI, SHI, SBI, SW, SH, SB, SWI.bi, SHI.bi, SBI.bi, SW.bi, SH.bi, SB.bi	Store instructions which have the data register as the consumed state
MEM_A_!bi	LWI, LHI, LHSI, LBI, LBSI, LW, LH, LHS, LB, LBS, LMW, SWI, SHI, SBI, SW, SH, SB, SMW	Ra and Rb (for LW, LH, LHS, LB, LBS, SW, SH, and SB only) of LD/ST instructions (non-bi form)

Consumer Group	Instructions	Note
MEM_A.bi.Ra	LWI.bi, LHI.bi, LHSI.bi, LBI.bi, LBSI.bi, LW.bi, LH.bi, LHS.bi, LB.bi, LBS.bi, SWI.bi, SHI.bi, SBI.bi, SW.bi, SH.bi, SB.bi	Ra of the listed LD/ST instructions
MEM_A.bi.Rb	LW.bi, LH.bi, LHS.bi, LB.bi, LBS.bi, SW.bi, SH.bi, SB.bi	Rb of the listed LD/ST instructions
BR	JR, RET, JRAL, BEQ, BNE, BEQZ, BNEZ, BGEZ, BLTZ, BGTZ, BLEZ, BGEZAL, BLTZAL, BEQC, BNEC	
PUSH25_D	PUSH25	Registers to be pushed
MISC	ISYNC, MTSR	Consumes general register R

In AndesCore N8, a producer and its corresponding consumer normally have a latency of 1.

Table 64. N8 Producer–Consumer Latency

No	Producer	Consumer	Latency
1	ALU, MUL, MOVD_O, LD_D_!bi, LMW_iHdL_nup, PUSH25_R8, LD_A.bi, PUSH25_SP_Re0	ALU, MUL, MOVD_E, MOVD_O, ST_D, MEM_A.bi.Rb, BR, MISC, PUSH25_D	1
		MEM_A.bi.Ra, MEM_A_!bi, POP25_SP, PUSH25_SP_Re0, PUSH25_SP_Re!0	2
2	MOVD_E, POP25_SP, LD_D.bi, PUSH25_SP_Re!0, POP25_D	All consumers	1

9.3.2. Execution Latency

Table 65. N8 Instruction Execution Latency

Instruction Type	Latency	Descriptions
ALU Data Operation	1	32-bit: ADD, ADDI, AND, ANDI, CMOVN, CMOVZ, MOVI, NOR, OR, ORI, ROTR, ROTRI, SEB, SEH, SETHI, SLL, SLLI, SLT, SLTI, SLTS, SLTSI, SRA, SRAI, SRL, SRLI, SUB, SUBRI, SVA, SVS, WSBH, XOR, XORI, ZEB, ZEH. 16-bit: ADD333, ADD45, ADDI333, ADDI45, MOV55, MOVI55, SEB33, SEH33, SLLI333, SLT45, SLTI45, SLTS45, SLTSI45, SRAI45, SRLI45, SUB333, SUB45, SUBI333, SUBI45, X11B33, XLSB33, ZEB33, ZEH33, ADDRI36.SP, NEG33, NOT33, XOR33, AND33, OR33, BMSKI33, FEXTI33, MOVPI45.
Multiply (Fast)	1	32-bit: MUL 16-bit: MUL33
Multiply and Accumulation (Fast)	2	MADDR32, MSUBR32
Multiply (Small)	17	32-bit: MUL 16-bit: MUL33
Multiply and Accumulation (Small)	18	MADDR32, MSUBR32
Divide	12-37	DIVR, DIVSR (4 cycles if dividend==0)
Control Transfer	3	BGEZAL, BLTZAL, J, JAL, JR, JRAL, RET

Instruction Type	Latency	Descriptions
		BEQ, BNE, BEQZ, BNEZ, BGEZ, BLTZ, BGTZ, BLEZ, BEQC, BNEC
Load Single without Post-increment	2	LW, LWI, LH, LHI, LB, LBI, etc...
Load Single with Post-increment	3	LW.bi, LWI.bi, LH.bi, LHI.bi, LB.bi, LBI.bi, etc...
Store Single with Immediate Offset	1	SWI, SWI.bi, SHI, SHI.bi, SBI, SBI.bi, etc...
Store Single with Register Offset	2	SW, SW.bi, SH, SH.bi, Sb, SB.bi, etc...
Load Multiple	N+1	LMW
Load Multiple with base-register Update	N+2	LMW.m
Store Multiple	N	SMW
Store Multiple with base-register Update	N+1	SMW.m
Push with Re < 1	M+1	PUSH25
Push with Re >= 1	M+2	PUSH25
Pop	M+5	POP25
Move Double	2	MOVD
Resume Operations	4	ISB, IRET, Interruption

Instruction Type	Latency	Descriptions
Miscellaneous	1	MFSR, MTSR, DSB, ISYNC, MSYNC, STANDBY

1. An ideal AHB-Lite system bus is assumed here. All fetch/load/store operations through the system bus cost no extra delay cycles other than the two cycles of address and data phrases.
2. “N” represents the number of destination registers indicated by LMW and SMW instructions via the begin, end and enable fields.
3. “M” represents the number of registers to be pushed/popped by the push and pop instructions.

9.3.3. Data Hazard Penalty

Table 66. N8 Data Hazard Penalty

Event Type	Extra Penalty
The destination register of the current instruction is used as the source register of the address calculation by the following load, store or control transfer instruction	1

9.3.4. Miscellaneous Penalty

Table 67. N8 Miscellaneous Penalty

Event Type	Extra Penalty
Unaligned 32-bit instruction fetch after pipeline flushes (redirect or resume events)	1

Official
Release

10. AndesCore N12 Implementation ISA Feature List

This chapter describes CCTL and STANDBY instructions implemented in AndesCore N12 family in the following sections

- 
- 10.1 CCTL Instruction on page 405
 - 10.2 STANDBY Instruction on page 405

10.1. CCTL Instruction

CCTL subtype operations implemented in AndesCore N12 are shown in the following table with a LIGHT GREEN background. Note that four defined optional operations are not implemented and they are shown in the table with a CLEAR background.

Table 68. N12 Implementation of CCTL Instruction

SubType		bit 4-3			
		0	1	2	3
bit 2-0		00	01	10	11
		L1D_IX	L1D_VA	L1I_IX	L1I_VA
0	000	L1D_IX_INVAL	L1D_VA_INVAL	L1I_IX_INVAL	L1I_VA_INVAL
1	001	L1D_IX_WB	L1D_VA_WB	-	-
2	010	L1D_IX_WBINVAL	L1D_VA_WBINVAL	-	-
3	011	L1D_IX_RTAG	L1D_VA_FILLCK	L1I_IX_RTAG	L1I_VA_FILLCK
4	100	L1D_IX_RWD	L1D_VA_ULCK	L1I_IX_RWD	L1I_VA_ULCK
5	101	L1D_IX_WTAG	-	L1I_IX_WTAG	-
6	110	L1D_IX_WWD	-	L1I_IX_WWD	-
7	111	L1D_INVALALL	-	-	-

10.2. STANDBY Instruction

In AndesCore N12 implementation, *software interrupt* in IVIC mode will not wakeup a core in the STANDBY mode.

11. AndesCore N1213 Hardcore N1213_43U1HA0

Implementation Restriction

This chapter describes N1213 Hardcore (N1213_43U1H) implementation restriction in the following sections

11.1 Instruction Restriction on page 407

11.2 ISYNC Instruction Note on page 407

11.1. Instruction Restriction

The following instructions are not implemented in N1213 hardcore N1213_43U1HA0 (CPU_VER == 0x0C010003):



DIV/DIVS
STANDBY wait_done
MFUSR Rt, PC
MFUSR Rt, USR, Group 1 and Group 2
MTUSR Rt, USR, Group 1 and Group2

11.2. ISYNC Instruction Note

The correct instruction sequence to write or update any code data that will be executed afterwards for AndesCore N1213 hardcore is as follows:

```

UPD_LOOP:
  // preparing new code data in Ra
  .....
  // preparing new code address in Rb, Rc
  .....
  // writing new code data
  store Ra, [Rb, Rc]
  // looping control
  .....
  bne Rb, Re, UPD_LOOP

WB_LOOP:
  // preparing new code address in Rd
  isync Rd (or cctl Rd, L1D_VA_WB)
  // looping control
  bne Rd, Re, WB_LOOP
  msync
  isb

ICACHE_INVAL_LOOP:
  // preparing new code address in Rf
  cctl Rf, L1I_VA_INVAL
  // looping control
  .....

```

```
bne Rf, Re, I CACHE_I NV_LOOP  
i sb  
// execution of new code data can be started from here  
.....
```

Please see the ISYNC instruction note in Section 8.1.23 for other implementations.

