AndeStar™
Instruction Set Architecture
Manual

Document Number：MA0100-013
Issued：May 2009

Revision History

Revision Number	Revision Date	Author	Revised Chapters/Sections	Description
V1.3	5/5/2009	CH/Wilson		1. Added LBUP/SBUP instructions.
V1.2	12/31/2008	CH/Wilson		1. Added Baseline V2 instructions.
V1.1	6/13/2008	CH/Wilson		1. Added N10 latency information. 2. Added Reduced Register configuration description.
V1.0RB	2/25/2008	CH/Wilson		Added DIV/DIVS instructions
V 1.0	6/5/2007	CH/Wilson		First version

Table of Contents

List of Tables

List of Figures

Preface

This preface describe the contents of this manual

About this manual

This manual provide the information about AndeStar™ instruction set architecture and all information contains in this document is subject to change without notice

Version of AndeStar™ ISA manual

This manual is version 1.3.

Contact information

Please contact Andes Technology Corporation by email at support@andestech.com or on the Internet at www.andestech.com for support.

Copyright notice

Chapter 1
Introduction

This chapter describes overview of AndeStar instruction set and contains the following sections

1.1 32/16-Bit ISA

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

The Andes 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. The 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. The ISA is a RISC-style register-based instruction set.
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 Data Types

Andes ISA supports the following data types:

1. Integer
   - Bit (1-bit, b)
   - Byte (8-bit, B)
   - Halfword (16-bit, H)
   - Word (32-bit, W)

1.3 Registers

As a whole, the Andes instructions can access thirty-two 32-bit General Purpose Registers (GPR) and four 32-bit User Special Registers (USR). The four 32-bit USRs can be combined into two 64-bit registers and used to store 32-bit multiplication results. The

GPRs will be named from r0 to r31. And the USRs will be named d0.lo, d0.hi, d1.lo, and d1.hi.

For the Andes 16-bit instructions, a register index can be 5-bit, 4-bit, or 3-bit. So the 3-bit and 4-bit register operand field can only access a subset of the thirty-two GPRs. The 3-bit and 4-bit register subsets and its index-number to register-number mappings are listed in the following table (Table 1) along with the software usage convention used by Andes tool chains.

Table 1 Andes 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
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 Point / Saved by callee
r29	gp			Global Pointer
r30	lp			link pointer
r31	sp			Stack Pointer

The four USRs can only be accessed by the 32-bit instructions.

Table 2 Andes 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 Andes ISA assumes an implied Program Counter (PC) which records the address of the currently executing instruction. And the value of this implied PC can not be read but can be changed by using the control flow instructions. To provide hardware acceleration for accessing mixed-endian data in memory, the Andes ISA assumes an implied endian mode bit (PSW.BE) which affects the endian behavior of the load/store instructions. This implied endian mode bit can not be read but can be changed by using the SETEND instruction.

Table 3 Andes Status Registers

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

1.3.1 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 registers to 16. In this "Reduced Register" configuration, the general registers that can be used by instructions are defined in Table 4. Notice that in the table, r11-r14, and r16-r27 have been removed.

In this Reduced Register configuration, if any of the unimplemented register is used in an instruction, a Reserved Instruction exception will be generated.

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

1.4 Instruction Classes

The Andes ISA can be classified into the following classes:

1. Baseline Instructions
   - 32-Bit Baseline
   - 16-Bit Baseline
2. Extension Instructions
   - Performance Extension

1.5 Instruction Encoding

Bit [31] of a 32-bit instruction and Bit [15] of a 16-bit instruction is used to 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 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

The 32-bit instruction formats and the meaning of each filed are described below:

- Type-0 Instruction Format

0	opc_6	{sub_1, imm_24}

- Type-1 Instruction Format

0	opc_6	rt_5	imm_20	
0	opc_6	rt_5	sub_4	imm_16

➢ Type-2 Instruction Format

0	opc_6	rt_5	ra_5	imm_15
0	opc_6	rt_5	ra_5	{sub_1, imm_14}

➢ Type-3 Instruction Format

0	opc_6	rt_5	ra_5	rb_5	sub_10
0	opc_6	rt_5	ra_5	imm_5	sub_10

➢ Type-4 Instruction Format

0	opc_6	rt_5	ra_5	rb_5	rd_5	sub_5
0	opc_6	rt_5	ra_5	imm1_5	imm2_5	sub_5

- opc_6: 6-bit opcode
- rt_5: target register in 5-bit index register set
- ra_5: source register in 5-bit index register set
- rb_5: source register in 5-bit index register set
- rd_5: destination register in 5-bit index register set
- sub_10: 10-bit sub-opcode
- sub_5: 5-bit sub-opcode
- sub_4: 4-bit sub-opcode
- sub_2: 2-bit sub-opcode
- sub_1: 1-bit sub-opcode
- imm_24: 24-bit immediate value, for unconditional jump instructions (J, JAL). The immediate value is used as the lower 24-bit offset of same 32MB memory block (new PC[31:0] = {current PC[31:25], imm_24, 1'b0}
- imm_20: 20-bit immediate value. Sign-extended to 32-bit for MOVI operations.
- imm_16: signed PC relative address displacement for branch instructions.
- imm_15: 15-bit immediate value. Zero extended to 32-bit for unsigned operations, while sign extended to 32-bit for signed operations.
- imm_14: signed PC relative address displacement for branch instructions.
- imm_5, imm1_5, imm2_5: 5-bit unsigned count value or index value

For the detailed decoding information, please refer to the individual chapters.

1.5.2 16-Bit Instruction Format

Please see the “16-Bit Baseline Instructions” chapter for more information.

1.6 Miscellaneous

- Instruction addressing is Halfword aligned
- Integer branch instructions will NOT use conditional codes (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, imm(ra), addr=ra+(imm << 2)
  - lh rt, imm(ra), addr=ra+(imm << 1)
- Register Remapping for 16-bit instructions
  - 32 registers (32-bit instructions) -> 32/16/8 registers (16-bit instructions)

Chapter 2
32-Bit Baseline Instruction

This chapter describes 32bit baseline instructions, including Data-processing instructions, Load and Store instructions, Jump and Branch Instructions, Privileged Resource Access Instructions and Miscellaneous Instructions and contains the following sections

The 32-bit baseline instruction set can be cataloged as follows:

1. Data-processing: (with Register or Immediate)
   - Basic ALU : ADD, SUB, AND, NOR, OR, XOR…
   - Shifter : ROTR, SLL, SRL, SRA…
   - Compare : SLT, SLTS
   - Set constant : SETHI, MOVI
   - Multiply (& Add) : MUL, MULT64, MADD64…
   - Divide : DIV, DIVS

2. Load and Store:
   - 5 types of load element:
     - Word
     - Unsigned Halfword and Signed Halfword
     - Unsigned Byte and Signed Byte
   - 3 types of store element:
     - Word
     - Halfword
     - Byte
   - 2 types of address calculation:
     - Register + Immediate
     - Register + (Register << shift)
   - 2 types of addressing mode:
     - Regular (After-Increment) and no base register upate
     - Before-Increment and base register update
   - Additional features:
     - Load/Store multiple word : LMW, SMW
     - Load/Store word for atomic operation : LLW, SCW
     - Load/Store word with user mode privilege : LWUP, SWUP

3. Jump and Branch:
   - Unconditional jump : J, JR
   - Unconditional function call jump : JAL, JRAL
   - Unconditional function return : RET
   - Conditional branch : BEQ, BNE, BEQZ, BNEZ
   - Conditional function call branch : BGEZAL, BLTZAL

4. Privileged Resource Access:
   - System control : MFSR, MTSR
   - TLB management : TLBOP

5. Miscellaneous:
   - Conditional move : CMOVZ, CMOVN
   - Synchronization : MSYNC, ISYNC
   - Prefetch : DPREF, DPREFI
   - No operation : NOP
   - Serialization : ISB, DSB
   - Exception generation : SYSCALL, BREAK…
   - Special return : IRET, RET.TOFF, RET.ITOFF
   - Cache control : CCTL
   - Miscellaneous : SETEND, SETGIE, STANDBY

2.1 Data-processing Instructions

Table 5 ALU Instruction with Immediate (Baseline)

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 (unsigned)< SE(imm15s)) ? 1 : 0
SLTSI rt5, ra5, imm15s	Set on Less Than Signed Immediate	rt5 = (ra5 (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)

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 = ~(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 (unsigned)< rb5) ? 1 : 0
SLTS rt5, ra5, rb5	Set on Less Than Signed	rt5 = (ra5 (signed)< rb5) ? 1 : 0
SVA rt5, ra5, rb5	Set on Overflow Add	rt5 = ((ra5 + rb5) overflow)? 1 : 0
SVS rt5, ra5, rb5	Set on Overflow Subtract	rt5 = ((ra5 - rb5) overflow))? 1 : 0
SEB rt5, ra5	Sign Extend Byte	rt5 = SE(ra5[7:0])
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)

Mnemonic	Instruction	Operation
SLLI rt5, ra5, imm5u	Shift Left Logical Immediate	rt5 = ra5 << imm5u
SRLI rt5, ra5, imm5u	Shift Right Logical Immediate	rt5 = ra5 (logic)>> imm5u
SRAI rt5, ra5, imm5u	Shift Right Arithmetic Immediate	rt5 = ra5 (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 (logic)>> rb5(4,0)
SRA rt5, ra5, rb5	Shift Right Arithmetic	rt5 = ra5 (arith)>> rb5(4,0)
ROTR rt5, ra5, rb5	Rotate Right	rt5 = ra5 >>| rb5(4,0)

Table 8 Multiply Instruction (Baseline)

Mnemonic	Instruction	Operation
MUL rt5, ra5, rb5	Multiply Word to Register	rt5 = ra5 * rb5

MULTS64 d1, ra5, rb5	Multiply Word Signed	d1 = ra5 (signed)* rb5
MULT64 d1, ra5, rb5	Multiply Word	d1 = ra5 (unsigned)* rb5
MADDS64 d1, ra5, rb5	Multiply and Add Signed	d1 = d1 + ra5 (signed)* rb5
MADD64 d1, ra5, rb5	Multiply and Add	d1 = d1 + ra5 (unsigned)* rb5
MSUBS64 d1, ra5, rb5	Multiply and Subtract Signed	d1 = d1 - ra5 (signed)* rb5
MSUB64 d1, ra5, rb5	Multiply and Subtract	d1 = d1 - ra5 (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

Mnemonic	Instruction	Operation
DIV Dt, ra5, rb5	Unsigned Integer Divide	Dt.L = ra5 (unsigned) / rb5; Dt.H = ra5 (unsigned) mod / rb5;
DIVS Dt, ra5, rb5	Signed Integer Divide	Dt.L = ra5 (signed) / rb5; Dt.H = ra5 (signed) / 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 <<	Load Word Immediate	address = ra5 + SE(imm15s << 2)

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],	Store Byte Immediate with	Byte-memory(ra5) = rt5[7:0]

imm15s	Post Increment	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))
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 175 for details.

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

Mnemonic	Instruction	Operation
LLW rt5, [ra5 + (rb5 << sv)]	Load Locked Word	
SCW rt5, [ra5 + (rb5 << sv)]	Store Condition Word	

Table 17 Load / Store Instructions with User-mode Privilege

Mnemonic	Instruction	Operation
LWUP rt5, [ra5 + (rb5 << sv)]	Load Word with User-mode Privilege Translation	Equavalent to LW instruction but with the user mode privilege address translation. See page 119 for details.
SWUP rt5, [ra5 + (rb5 << sv)]	Store Word with User-mode Privilege Translation	Equavalent to SW instruction but with the user mode privilege address translation. See page 193 for details.

2.3 Jump and Branch Instructions

Table 18 Jump Instruction (Baseline)

Mnemonic	Instruction	Operation
J imm24s	Jump	PC = PC + SE(imm24s << 1)

JAL imm24s	Jump and Link	LP = next sequential PC (PC + 4); PC = PC + SE(imm24s << 1)
JR rb5	Jump Register	PC = rb5
RET rb5	Return from Register	PC = rb5
JRAL rb5 JRAL rt5, rb5	Jump Register and Link	jaddr = rb5; LP = PC + 4; 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(imm14s << 1)) : (PC + 4)
BNE rt5, ra5, imm14s	Branch on Not Equal (2 Register)	PC = (rt5 =! ra5)? (PC + SE(imm14s << 1)) : (PC + 4)
BEQZ rt5, imm16s	Branch on Equal Zero	PC = (rt5==0)? (PC + SE(imm16s << 1)) : (PC + 4)
BNEZ rt5, imm16s	Branch on Not Equal Zero	PC = (rt5 =! 0)? (PC + SE(imm16s << 1)) : (PC + 4)
BGEZ rt5, imm16s	Branch on Greater than or Equal to Zero	PC = (rt5 (signed)>= 0)? (PC + SE(imm16s << 1)) : (PC + 4)
BLTZ rt5, imm16s	Branch on Less than Zero	PC = (rt5 (signed)< 0)? (PC + sign-ext(imm16s << 1)) : (PC + 4)
BGTZ rt5, imm16s	Branch on Greater than Zero	PC = (rt5 (signed)> 0)? (PC + SE(imm16s << 1)) : (PC + 4)
BLEZ rt5, imm16s	Branch on Less than or Equal to Zero	PC = (rt5 (signed)<= 0)? (PC + SE(imm16s << 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;
JRAL.TON rb5 JRAL.TON rt5, rb5	Jump Register and Link and Translation ON	jaddr = rb5; LP = PC+4 or rt5 = PC+4; PC = jaddr; PSW.IT = 1, PSW.DT = 1;

Table 23 MMU Instruction (Baseline)

Mnemonic	Instruction	Operation
TLBOP Ra, TargetRead (TRD)	Read targeted TLB entry	
TLBOP Ra, TargetWrite (TWR)	Write targeted TLB entry	
TLBOP Ra,	Write PTE into a TLB entry	

RWrite (RWR)		
TLBOP Ra, RWriteLock (RWLK)	Write PTE into a TLB entry and lock	
TLBOP Ra, Unlock (UNLK)	Unlock a TLB entry	
TLBOP Rt, Ra, Probe (PB)	Probe TLB entry	
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	
LD_VLPT	Load VLPT page table (optional instruction)	Load VLPT page table which always goes through data TLB translation. On TLB miss, generate 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	
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	
ISB	Instruction Serialization Barrier	

Table 29 Exception Generation Instruction (Baseline)

Mnemonic	Instruction	Operation
BREAK	Breakpoint	
SYSCALL	System Call	
TRAP	Trap Always	
TEQZ	Trap on Equal Zero	
TNEZ	Trap on Not Equal Zero	

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 148)
RET.TOFF Rb5	Return and turn off address translation (instruction/data)	PC = Rb5, PSW.IT = 0, PSW.DT = 0 (page 148)

Table 31 Cache Control Instruction (Baseline)

Mnemonic	Instruction	Operation
CCTL	Cache Control	Read, write, and control cache states. Please see page 54.

Table 32 Miscellaneous Instructions (Baseline)

Mnemonic	Instruction	Operation

SETEND.B
SETEND.L
	Atomic set or clear of PSW.BE bit	PSW.BE = 1; // SETEND.B
		PSW.BE = 0; // SETEND.L

SETGIE.E
SETGIE.D
	Atomic set or clear of PSW.GIE bit	PSW.GIE = 1; // SETGIE.E
		PSW.GIE = 0; // SETGIE.D

STANDBY	Wait for External Event	Enter standby state and wait for external event. Please see page 183.

Chapter 3
16-Bit Baseline Instruction

This chapter describes 16bit baseline instructions and contains the following sections

The Andes 16-bit instruction set is a subset of the 32-bit instruction set. So every 16-bit instruction can be properly mapped onto a corresponding 32-bit instruction. The mappings are listed in the next section.

3.1 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	
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)]
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+	Store Byte Immediate	SBI rt5, [ra5+ZE(imm3u)]

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)
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

Chapter 4
16/32-Bit Baseline Version 2 Instruction

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

These new instructions are summarized in the following sections.

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	Operation
ADDI.gp rt5, imm19s	GP-implied Add Immediate	rt5 = gp + SE(imm19s)

Table 46 Multiply and Divide Instructions

Mnemonic	Instruction	Operation

MULR64 rt5, ra5, rb5	Multiply unsigned word to registers	res = ra5 (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 (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 = Floor(ra5 (unsigned) / rb5); rs5 = ra5 (unsigned) mod rb5;
DIVSR rt5, rs5, ra5, rb5	Signed integer divide to registers	rt5 = Floor(ra5 (signed) / rb5); rs5 = ra5 (signed) mod 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 304 for details.
SMWA.{b| a}{i | d}{m?} rb5, [ra5], re5, Enable4	Store multiple word with alignment check	Please see page 318 for details.
LBUP Rt, [Ra + (Rb << sv)]	Load Byte with User Privilege	Equivalent to LB instruction but with the user mode privilege address translation. See page 301 for details.
SBUP Rt, [Ra + (Rb << sv)]	Store Byte with User Privilege	Equivalent to SB instruction but with the user mode privilege address translation. See page 316 for details.

Chapter 5
32-bit ISA Extensions

This chapter describes instructions of various 32-bit ISA extensions and contains the following sections

5.1 Performance Extension V1 Instructions

The performance extension version 1 instructions are used to optimize high level language (C / C++) performance. The instructions in this extension are summarized in the following sections.

Table 48 ALU Instruction (Extension)

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) (arith) >> 1 (page 213)
MAX rt5, ra5, rb5	Return the Larger	rt5 = signed-max (ra5, rb5)
MIN rt5, ra5, rb5	Return the Smaller	rt5 = signed-min (ra5, rb5)
BSET rt5, ra5, imm_5	Bit Set	rt5 = ra5 || (1 << imm_5)
BCLR rt5, ra5, imm_5	Bit Clear	rt5 = (ra5 && ~(1 << imm_5))
BTGL rt5, ra5, imm_5	Bit Toggle	rt5 = ra5 ^ (1 << imm_5)
BTST rt5, ra5, imm_5	Bit Test	rt5 = (ra5 && (1 << imm_5))? 1 : 0
CLIPS rt5, ra5, imm_5	Clip Value Signed	rt5 = (ra5 > 2^{imm5}-1)? 2^{imm5}-1 : ((ra5 < -2^{imm5})? -2^{imm5} : ra5)
CLIP rt5, ra5, imm_5	Clip Value	rt5 = (ra5 > 2^{imm5}-1)? 2^{imm5}-1 : ((ra5 < 0)? 0 : ra5)
CLZ rt5, ra5	Counting Leading Zeros in Word	rt5 = COUNT_ZERO_FROM_MSB(ra5)
CLO rt5, ra5	Counting Leading Ones in Word	rt5 = COUNT_ONE_FROM_MSB(ra5)

5.2 Performance Extension V2 Instructions

The performance extension version 2 instructions are used to optimize multimedia encoding/decoding processing related applications. The instructions in this extension are summarized in the following sections.

Table 49 Performance Extension V2 Instructions.

Mnemonic	Instruction	Operation
BSE rt5, ra5, rb5	Bitstream Extraction	rt5 = Bitstream_Extract(ra5, rb5) Please see page 225 for details.
BSP rt5, ra5, rb5	Bitstream Packing	rt5 = Bitstream_Packing(ra5, rb5) Please see page 231 for details.
PBSAD rt5, ra5, rb5	Parallel Byte Sum of Absolute Difference	a = ABS(ra5(7,0) – rb5(7,0)); b = ABS(ra5(15,8) – rb5(15,8)); c = ABS(ra5(23,16) – rb5(23,16)); d = ABS(ra5(31,24) – rb5(31,24)); rt5 = a + b + c + d;
PBSADA rt5, ra5, rb5	Parallel Byte Sum of Absolute Difference Accumulate	a = ABS(ra5(7,0) – rb5(7,0)); b = ABS(ra5(15,8) – rb5(15,8)); c = ABS(ra5(23,16) – rb5(23,16)); d = ABS(ra5(31,24) – rb5(31,24)); rt5 = rt5 + a + b + c + d;

5.3 32-bit String Extension

The String Extension instructions are use to optimize text and string processing related algorithms. These instructions will be released in the future.

Chapter 6
Coprocessor Instructions

The Coprocessor ISA extension will be released in the future.

Coprocessors may perform the following instructions:

1. Move General Purpose to Coprocessor Register
2. Move Coprocessor to General Purpose Register
3. Load Coprocessor Register
4. Store Coprocessor Register
5. Coprocessor Data Operation

Current version does not support Coprocessor instructions

Chapter 7
Detail Instruction Description

This chapter describes the detail description of each AndeStar instruction and contains the following sections

7.1 32-bit Baseline instructions

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 content of two registers.

Description: The content of Ra is added with the content of Rb. And the result is written to Rt.

Operations:

```
Rt = Ra + Rb;
```

Exceptions: None

Privilege level: All

Note:

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 with a signed constant.

Description: The content of Ra is added with the sign-extended imm15s. And the result is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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

Description: The content of Ra is combined with the content of Rb using a bit-wise logical AND operation. And the result is written to Rt.

Operations:

```
Rt = Ra & Rb;
```

Exceptions: None

Privilege level: All

Note:

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

Description: The content of Ra is bit-wise ANDed with the zero-extended imm15u. And the result is written to Rt.

Operations:

```
Rt = Ra & ZE(imm15u);
```

Exceptions: None

Privilege level: All

Note:

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: It is used for conditional PC-relative branching based on the result of comparing the contents of two registers.

Description: If the content of Rt is equal to the content of Ra, then branch to the target address of adding the current instruction address with the sign-extended (imm14s << 1) value. The branch range is ± 16K bytes.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: It is used for 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, then branch to the target address of adding the current instruction address with the sign-extended (imm16s << 1) value. The branch range is ± 64K bytes.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: It is used for 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, then branch to the target address of adding the current instruction address with the sign-extended (imm16s << 1) value. The branch range is ± 64K bytes.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: It is used for 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, then branch to the target address of adding the current instruction address with the sign-extended (imm16s << 1) value. The branch range is ± 64K bytes. The program address of the next sequential instruction (PC+4) is written to R30 (Link Pointer register) unconditionally for function call return purpose.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: It is used for 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, then branch to the target address of adding the current instruction address with the sign-extended (imm16s << 1) value. The branch range is ± 64K bytes.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: It is used for 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, then branch to the target address of adding the current instruction address with the sign-extended (imm16s << 1) value. The branch range is ± 64K bytes.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: It is used for 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, then branch to the target address of adding the current instruction address with the sign-extended (imm16s << 1) value. The branch range is ± 64K bytes.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: It is used for 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, then branch to the target address of adding the current instruction address with the sign-extended (imm16s << 1) value. The branch range is ± 64K bytes. The program address of the next sequential instruction (PC+4) is written to R30 (Link Pointer register) unconditionally for function call return purpose.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: It is used for 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 the content of Ra, then branch to the target address of adding the current instruction address with the sign-extended (imm14s << 1) value. The branch range is ± 16K bytes.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: It is used for 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, then branch to the target address of adding the current instruction address with the sign-extended (imm16s << 1) value. The branch range is ± 64K bytes.

Operations:

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

Exceptions: None

Privilege level: All

Note:

BREAK (Breakpoint)

Format:

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

Syntax: BREAK SWID

Purpose: It is used to generate a Breakpoint exception.

Description:

BREAK instruction will unconditionally generate a Breakpoint exception and transfer control to the Breakpoint exception handler. The 15-bits SWID is used by software as a parameter to distinguish different breakpoint features and usages.

Operations:

```
Generate_Exception(Breakpoint);
```

Exceptions: Breakpoint

Privilege level: All

Note:

CCTL (Cache Control)

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. This instruction is typically used by software to maintain cache coherence for shared memory.

Description:

This instruction is used to perform cache control operations based on the SubType field. The definition and encoding for the SubType field are listed in the following tables. Some CCTL operations can be used with user privilege to assist in coherence and synchronization management. Some CCTL cache read operations has an additional destination register Rt. And some CCTL write operation has an additional source register Rb. Although defined, certain SubType encodings of the cache control instruction are optional to implement. And execution of an unimplemented optional cache control operation will cause a Reserved Instruction exception. Similarly, execution of an undefined SubType encoding of this instruction will cause a Reserved Instruction exception as well. For the "level" field, please see "multi-level cache management operation" in page 60.

Table 50 CCTL SubType Encoding

SubType		bit 4-3			
bit 2-0		0	1	2	3
		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 51 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 (C)	Index	-/-	-	Optional
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	VA	-/-	-	Optional

	cache (E)				
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
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	Index/w	Yes/-		Optional

	data L1I cache (G)				
L1I_IX_WTAG	Write tag L1I cache (H)	Index	-/Yes	-	Optional
L1I_IX_WWD	Write word data L1I cache (I)	Index/w	-/Yes	-	Optional

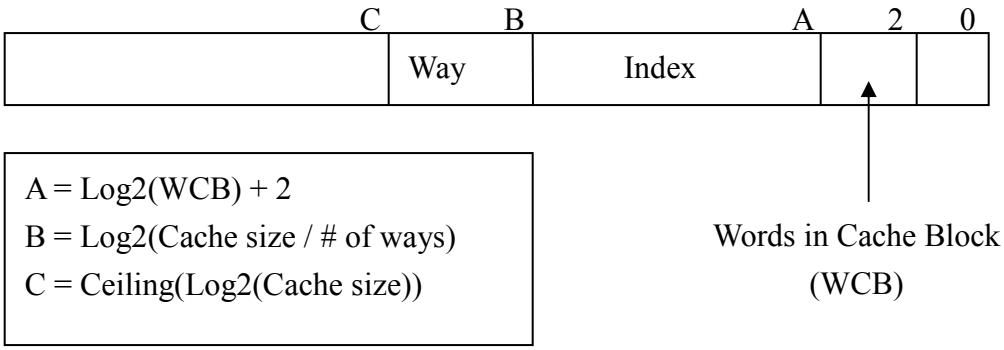
*Note: The “Write word data L1D cache” operation may be omitted by an implementation since its effect can be achieved using store instructions after the CCTL “Write tag L1D cache” operation and a “Data Serialization Barrier” instruction.

The content of the register Ra is used to find the correct cache location for the cache operation. It is used in the following two ways:

Ra Type	Usage
Index	The content of Ra is used 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 per cache implementation-dependent. However, it should have a general form shown in Figure 1. Software could discover the real format by consulting the ICM CFG (cr1) and DCM_CFG (cr2) Configuration Registers in the Andes processor core.
Index/w	The content of Ra is used directly as a (Index, Way, Word) triple to access the cache location without going through any translation. The “Word” means a 4-bytes word in the cache line pointed to by the (Index, Way) pair. The remaining description is similar to the “Index” type above.
VA	The content of Ra is used 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. And the specified operation is only performed 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 can be grouped and described in the following categories:

A. Invalidate cache block

This operation unlocks the cache line and sets the state of the cache line 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, only the lock state will be set. A locked cache line will not be replaced when a cache miss/fill event happens. A locked cache line 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. And if this instruction is supported for a multi-way set-associative cache, only up to

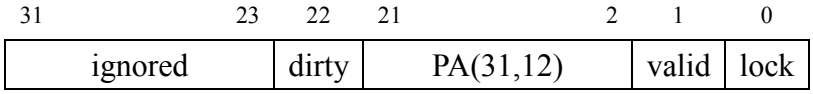
"way-1" cache lines in a cache set can be locked using this instruction. If software wants to get a predictable behavior, care must be taken to avoid locking more than "way-1" cache lines in a cache set. If "way" cache lines in a cache set are all locked by software, and if ICALCK/DCALCK of Cache Control system register is 0, then 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.



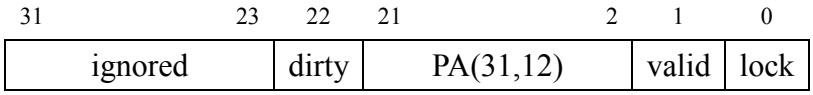
The content of Ra specifies the index and way of the target cache line.

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.



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 the state of all of the cache lines to invalid. It is implementation-dependent on how this instruction affects other states of the cache line 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 the software 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 has the benefit that software can use just one CCTL instruction to affect all levels of caches without using separate instructions to manage each level of cache individually.

  To enable this, all of the 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 all other CCTL SubTypes, the “level” encoding will be a “Don’t care” field.

  For software to make sure the completeness of the “all level” operations, a “MSYNC all” instruction needs to be used after the “all level” operations to guarantee that any instructions after the “MSYNC all” will see the effects completed by the “all level” operations.

  For a system with one level of caches, the “all level” flavor behaves the same as the “one level” operation.

  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 will invalidate cache lines of different size in different levels of a cache hierarchy. So to use this instruction, a programmer needs to first make sure writing back all data in a cache line unit aligned to the largest cache line size in the cache hierarchy unless the programmer is sure that portion of the cache unit covering the largest cache line size, when not written back, contain 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 *user mode*	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 and access bit 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.

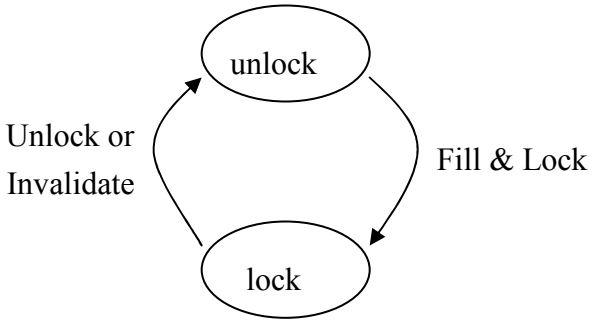
Note:

(1) A CCTL instruction should operate on the specified cache regardless of the cache enable/disable control bits in the Cache Control register provided the specified cache is implemented.

(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.



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 a condition stored in a register.

Description: If the content of Rb is not equal to zero, then move the content of Ra into Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 a condition stored in a register.

Description: If the content of Rb is equal to zero, then move the content of Ra into Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 contents of two 32-bit registers.

Description: Divide the 32-bit content of Ra with the 32-bit content of Rb. The 32-bit quotient result is written to Dt.LO register and the 32-bit remainder result is written to Dt.HI register. The contents of Ra and Rb are treated as unsigned integers.

If the content of Rb is zero, an Arithmetic exception will be generated if the IDIVZE bit of the INT_MASK register is 1, which 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.LO = quotient;
   Dt.HI = remainder;
} else if (INT_MASK.IDIVZE == 0) {
   Dt.LO = 0;
   Dt.HI = 0;
} else {
   Generate_Exception(Arithmetic);
}
```

Exceptions: Arithmetic

Privilege level: All

Note:

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	000	Dt	0	Ra	Rb	0000	DIVS 101110

Syntax: DIVS Dt, Ra, Rb

Purpose: Divide the signed integer contents of two 32-bit registers.

Description: Divide the 32-bit content of Ra with the 32-bit content of Rb. The 32-bit quotient result is written to Dt.LO register and the 32-bit remainder result is written to Dt.HI register. The contents of Ra and Rb are treated as signed integers.

If the content of Rb is zero, an Arithmetic exception will be generated if the IDIVZE bit of the INT_MASK register is 1, which enables exception generation for the "Divide-By-Zero" condition. If the quotient overflows, an Arithmetic exception will always be generated. 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(Arithmetic);
   }
   remainder = Ra mod Rb
   Dt.LO = quotient;
   Dt.HI = remainder;
} else if (INT_MASK.IDIVZE == 0) {
   Dt.LO = 0;
   Dt.HI = 0;
} else {
   Generate_Exception(Arithmetic);
```

```
}
```

Exceptions: Arithmetic
Privilege level: All
Note:

DPREF/DPREFI (Data Prefetch)

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 in advance before the actual load or store operations to reduce memory access latency.

Description:

The effective byte address calculated from the data prefetch instruction (DPREF/DPREFI) is used by an implementation to take an implementation-dependent action which is expected to increase performance by moving the memory line containing the address from memory into data caches in advance. However, an implementation may decide to do nothing at any stage by treating this instruction as a NOP. As a hint, this instruction should not generate any observable results or any exceptions which alter the behavior of a program (e.g. an address exception which leads to OS aborting 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.

For this instruction to be effective in increasing performance, the data prefetch instruction should be implemented non-blocking to overlap with other instructions. And the actual number of cache lines and memory hierarchies affected are implementation-dependent.

The effective byte address for DPREF is Ra+(Rb<<shift2) where shift2 is a 2-bits shift amount for Rb.

The effective byte address for DPREFI is defined in two flavors, one for word and one for double word. The double word flavor has a byte address of Ra+SignExtend(imm15s.000) where imm15s represents an 8-byte-double-word offset. This coarser-grain immediate offset will not sacrifice the data prefetching performance since most of the cache block moves between the memory and data cache is greater than 8 bytes. The word flavor has a byte address of Ra+SignExtend(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. These two flavors are distinguished by the "d" bit (bit 24) in the instruction encoding as follows:

"d" bit	Meaning
0	DPREFI.w
1	DPREFI.d

The SubType field of this instruction is used as a hint to tell hardware the intended use of the prefetched data, so that the hardware implementation may use different prefetch schemes to optimize the performance. The definition of the SubType field is listed in the following table.

Sub Type	Mnemonics	Hint	Descripton
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)
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 is permitted for writing, the processor will do nothing. If the cache line

			is in the data cache but is 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, the cache line 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. And the final memory states will be UNPREDICTABLE in this case depending on cache hit or miss when this instruction is issued. This memory states altering characteristic is implementation-dependent.
6-15	-	Imp-dep	Implementation dependent

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

Operations:

```
VA = Ra+(Rb << shift2); // DPREF
VA = Ra+(imm15s.000);   // DPREFI.d
VA = Ra+(imm15s.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 the address is used to indicate a cache line rather than a specific data type.

(2) This data prefetch instruction does not generate any exception. And if a hardware page table walker is implemented, it is suggested that this instruction should 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 unguarded inside a loop. So it is very likely that a data prefetch instruction may generate an address which is outside of legal data range. Generating hardware page table walk memory access on a data prefetch instruction with such data address will cause unnecessary execution penalty which deter the usability of this instruction.

DSB (Data Serialization Barrier)

Format:

31	30 25	24 20	19 5	4 0
0	MISC 110010	00000	000000000000000	DSB 01000

Syntax: DSB
Purpose: It is used to serialize a read-after-write data dependency for certain architecture/hardware states updates which affect data processing related operations. It guarantees a modified architecture or hardware state can be seen by any 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. This instruction is not needed to serialize general register states which 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 any following load/store instructions. Other than this case, all uses of DSB are in privileged programs for managing system states.

Operations:

```
Serialize_Data_States()
```

Exceptions: None
Privilege level: All
Note:

1. The following table lists some of the state writers and readers where DSB is needed between them in order 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-H0IM	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
DMA channel selection (DMA_CHNSEL)	MTSR DMA_CHNSEL	MTSR/MFSR DMA channel registers	New DMA channel number

2. PSW.GIE is defined to control if an asynchronous interrupt can be inserted in between two instructions or not. So when we say that an instruction will observe an updated PSW.GIE value, we mean that the updated PSW.GIE value will affect 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 will not guaranteed itself to

observe the updated PSW.GIE value and may observe the old PSW.GIE value. So if the PSW.GIE is being updated to 0 to disable interrupt, an interrupt can still be inserted between the DSB and the immediate following instruction (inst1). The DSB instruction only guarantees that the following instructions after it (inst1, inst2, etc.) will observe the updated PSW.GIE value to be 0, thus preventing any interrupt inserted between the immediate following instruction (inst1) and its next instruction (inst2), and onward.

```
setgie.d
dsb
inst1
inst2
```

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

INT_MASK.SIM
INT_MASK.H5IM-H0IM

IRET (Interruption Return)

Format:

31	30 25	24 20	19 5	4 0
0	MISC 110010	00000	000000000000000	IRET 00100

Syntax: IRET

Purpose: It is used to return from interruption to the instruction and states when the processor was being interrupted.

Description: The main function of the IRET instruction is to conditionally update (pop) the system register and program counter stack such that the processor behavior after the IRET instruction will return to a state when the processor was being interrupted. To be more specific, the following states will be updated based on interruption level (INTL) conditions:

- For INTL=0, 1
  PC ← IPC
  PSW ← IPSW

- For INTL=2
  PC ← IPC
  IPC ← P_IPC
  PSW ← IPSW
  IPSW ← P_IPSW
  EVA ← P_EVA
  P0 ← P_P0
  P1 ← P_P1
  ITYPE ← P_ITYPE
  EXCI ← P_EXCI

- For INTL=3
  PC ← OIPC
  PSW.INTL ← 2

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

Operations:

```
If (PSW.INTL==0 or PSW.INTL ==1 or PSW.INTL==2) {
  PC = IPC;
  PSW = IPSW;
  If (PSW.INTL==2) {
    IPC = P_IPC;
    IPSW = P_IPSW;
    EVA = P_EVA;
    P0 = P_P0;
    P1 = P_P1;
    ITYPE = P_ITYPE;
    EXCI = P_EXCI;
  }
} else {
  PC = OIPC;
  PSW.INTL = 2;
}
Serialize_Data_States()
Serialize_Instruction_States()
```

Exceptions: Privileged Instruction
Privilege level: Superuser and above
Note:

ISB (Instruction Serialization Barrier)

Format:

31	30 25	24 20	19 5	4 0
0	MISC 110010	00000	000000000000000	ISB 01001

Syntax: ISB
Purpose: It is used to serialize a read-after-write data dependency for certain architecture/hardware states updates which affect instruction fetching related operations. It also serializes all architecture/hardware states updates that are serialized by a DSB instruction. It guarantees a modified architecture or hardware state can be seen by any 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. This instruction is not needed to serialize general register states which are serialized by hardware.

The interruption return instruction (IRET) includes implicit ISB operation (including data serialization). So using either IRET or ISB instruction will achieve the expected instruction and data serialization behavior.

Operations:

```
Serialize_Data_States()
Serialize_Instruction_States()
```

Exceptions: None
Privilege level: All
Note:

The following table lists some of the state writers and readers where ISB is needed between them in order 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
I$ line valid/data	ISYNC	Instruction fetch	New instruction data
I$ line valid	CCTL L1I (sub=inval)	Instruction fetch	I$ miss
I$ line valid/tag	CCTL L1I (sub=WR tag)	CCTL L1I (sub=Rd tag)	I$ hit
I$ 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

ISYNC (Instruction Data Coherence Synchronization)

Format:

31	30 25	24 20	19 5	4 0
0	MISC 110010	Ra	000000000000000	ISYNC 01101

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

Description:

This instruction is used to guarantee that any instruction data updated before this instruction can be properly observed by an instruction fetch event after an instruction serialization barrier instruction. This instruction alleviates the Andes architecture implementation from implementing coherence logic between instruction and data caches in order to handle self-modifying code.

Software is required to use this instruction and the correct instruction sequence, after generating new instruction data, to synchronize the updated data between I and D caches in order to correctly execute the newly modified instruction later defined after an instruction serialization barrier instruction. If this instruction and the correct instruction sequence are not used in the above situation, the Andes implementation may still fetch and execute the old instruction data after any period of time.

After updating instruction data, this instruction initiates a data cache write-back operation and an instruction cache invalidating operation. And particularly, this instruction 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 could happen for this instruction.

Operation:

```
VA = (Ra)
PA = AddressTranslation(VA)
SynchronizeAllCaches(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 the address is used to indicate a cache line rather than a specific data type. Thus, no Data Alignment Check exception is generated for this instruction.
4. The correct instruction sequence for writing or updating any code data that will be executed afterwards is as follows, except for AndesCore N1213 hardcore: N1213_43U1H

```
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
ISYNC_LOOP:
    // preparing new code address in Rd
    isync Rd
    // looping control
    bne ISYNC_LOOP
    isb
    // execution of new code data can be started from here
    ......
```

For AndesCore N12 hardcore (N1213_43U1H), 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
......
```

J (Jump)

Type: 32-Bit Baseline

Format:

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

Syntax: J imm24s

Purpose: Unconditional branch relative to current instruction.

Description: Branch 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 ± 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 than the encoding syntax shown here, please consult the assembly programming guide or disassembler manual to get the correct meaning of the displayed syntax.

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: Unconditional function call relative to current instruction.

Description: Branch 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 ± 16M bytes. The program address of the next sequential instruction (PC+4) is written to R30 (Link Pointer register) for function call return purpose.

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 than the encoding syntax shown here, please consult the assembly programming guide or disassembler manual to get the correct meaning of the displayed syntax.

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: Unconditional branch to an instruction address stored in a register.

Description: Branch unconditionally to an instruction address stored in Rb. The JR hint field is used to distinguish this instruction from the RET instruction which has the same architecture behavior but different software usages.

Operations:

```
PC = Rb;
```

Exceptions: None

Privilege level: All

Note:

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

Description: Branch unconditionally to an instruction address stored in Rb and 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 will see PSW.IT as 0 (and PSW.DT as 0 if included), thus will not go through the address translation process. The JR hint field is used to distinguish this instruction from the RET.xTOFF instruction which has the same architecture behavior but different software usages.

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 which is in a non-translated address space. Please see

JRAL.xTON instruction for the reverse process of coming from a non-translated address space to a translated address space.

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: Unconditional function call to an instruction address stored in a register.

Description: Branch unconditionally to an instruction address stored in Rb. The program address of the next sequential instruction (PC+4) is written to Rt for function call return purpose.

Operations:

```
jaddr = Rb;
Rt = PC + 4;
PC = jaddr;
```

Exceptions: None

Privilege level: All

Note:

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

Description: Branch unconditionally to an instruction address stored in Rb and 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. The program address of the next sequential instruction (PC+4) is written to Rt for function call return purpose. This instruction guarantees that fetching of the target instruction will see PSW.IT as 1 (and PSW.DT as 1 if included), thus will go through the address translation process.

Operations:

```
jaddr = Rb
Rt = PC + 4;
PC = jaddr;
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 which is in a translated address space. Please see JR.xTOFF/RET.xTOFF instruction for the reverse process of returning from a translated address space to a non-translated address space.

LB (Load Byte)

Type: 32-Bit Baseline

Format:

LB

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: To load a zero-extended 8-bit byte from memory into a general register.

Description: This instruction loads a zero-extended byte from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + (Rb << sv) value after the memory load operation. And 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:

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: To load a zero-extended 8-bit byte from memory into a general register.

Description: This instruction loads a zero-extended byte from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + SE(imm15s) value after the memory load operation. And 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:

1. "LBI R0, [R0+0]" baseline version 2 special behavior:
   This instruction will become 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 this special behavior is used, compiler and assembler should avoid generating this instruction.

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: To load a sign-extended 8-bit byte from memory into a general register.

Description: This instruction loads a sign-extended byte from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + (Rb << sv) value after the memory load operation. And 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:

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: To load a sign-extended 8-bit byte from memory into a general register.

Description: This instruction loads a sign-extended byte from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + SE(imm15s) value after the memory load operation. And 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:

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: To load a zero-extended 16-bit halfword from memory into a general register.

Description: This instruction loads a zero-extended halfword from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + (Rb << sv) value after the memory load operation. And 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, a Data Alignment Check exception will be generated.

Operations:

```
Addr = Ra + (Rb << sv);
If (.bi form) {
   Vaddr = Ra;
} else {
   Vaddr = Addr;
}
```

```
if (!Halfword_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) {
   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:

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: To load a zero-extended 16-bit halfword from memory into a general register.

Description: This instruction loads a zero-extended halfword from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + SE(imm15s << 1) value after the memory load operation. And 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, a Data Alignment Check exception will be generated.

Operations:

```
Addr = Ra + Sign_Extend(imm15s << 1);
If (.bi form) {
   Vaddr = Ra;
} else {
```

```
   Vaddr = Addr;
}
if (!Halfword_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) {
   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:

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: To load a sign-extended 16-bit halfword from memory into a general register.

Description: This instruction loads a sign-extended halfword from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + (Rb << sv) value after the memory load operation. And 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, a Data Alignment Check exception will be generated.

Operations:

```
Addr = Ra + (Rb << sv);
If (.bi form) {
   Vaddr = Ra;
} else {
   Vaddr = Addr;
}
```

```
if (!Halfword_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) {
   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:

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: To load a sign-extended 16-bit halfword from memory into a general register.

Description: This instruction loads a sign-extended halfword from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + SE(imm15s << 1) value after the memory load operation. And 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, a Data Alignment Check exception will be generated.

Operations:

```
Addr = Ra + Sign_Extend(imm15s << 1);
If (.bi form) {
   Vaddr = Ra;
} else {
```

```
   Vaddr = Addr;
}
if (!Halfword_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) {
   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:

LLW (Load Locked Word)

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: It is used as a primitive to perform atomic read-modify-write operations.
Description: The 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, then 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 exists per processor at any one time. And if a new LLW instruction is issued before an active sequence is completed by a SCW instruction, the new LLW instruction will start a new sequence which replaces the previous sequence.

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. And the success or failure status is stored back into the source register (Please see SCW instruction description 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 bytes memory region and its exact size is implementation-dependent, but 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 all success or fail conditions.

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

Operations:

```
VA = Ra + Rb;
If (VA(1,0) != 0) {
   Generate_Exception(Data_alignment_check);
}
(PA, Attributes) = Address_Translation(VA, PSW.DT);
Excep_status = Page_Exception(Attributes, UserMode, 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 LLW and SCW instruction sequences.

Additional Software Constraints:

For N1213 hardcore N1213_43U1HA0 (CPU_VER==0x0C010003), additional software constraints must be followed to ensure correct LLW/SCW operations.

- If LLW/SCW are used in an interruption handler, it must be followed that
  - Execution of a LLW instruction must lead to execution of a SCW instruction. UPREDICTABLE result may happen if execution of a LLW instruction eventually leads to execution of an IRET instruction without going through a SCW instruction.

LMW (Load Multiple Word)

Format:

31	30 25	24 20	19 15	14 10	9 6	5	4	3	2	1 0
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: load 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 specified by a registers list formed by Rb, Re, and the four-bit Enable4 field as follows.

<Registers 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} use the contents of Ra as the first memory load address. {a} use 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 this register range. 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 <Registers 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 result 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. And the rest of the loaded registers should have 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, UserMode, 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 loaded its own base register and the memory access to load the base register occurred 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. And they do not guarantee single access to a memory location during the execution either. 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 implemented by an implementation is:

- For LMW/SMW.i : increasing memory addresses from base address.
- For LMW/SMW.d: decreasing memory addresses from 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 implemented by 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 or .
- 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 will more likely 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”.

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: To load a 32-bit word from memory into a general register.

Description: This instruction loads a word from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + (Rb << sv) value after the memory load operation. And 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, a Data Alignment Check exception will be generated.

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:

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: To load a 32-bit word from memory into a general register.

Description: This instruction loads a word from the memory into the general register Rt. Two different forms are used 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 will be updated with the Ra + SE(imm15s << 2) value after the memory load operation. And 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, a Data Alignment Check exception will be generated.

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:

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: To 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, a Data Alignment Check exception will be generated.

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, UserMode, 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:

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 with the lower 32-bit content of a 64-bit data register. The final result is written back to the lower 32-bit of the 64-bit data register.

Description: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The lower 32-bit multiplication result is added with the content of Dt.LO 32-bit data register. And the final result is written back to Dt.LO data register. The contents of Ra and Rb can be either signed or unsigned integers.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 with the content of a 64-bit data register. The final result is written back to the 64-bit data register.

Description: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The 64-bit multiplication result is added with the content of Dt data register. And the final result is written back to Dt data register. The contents of Ra and Rb are treated as unsigned integers.

Operations:

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

Exceptions: None

Privilege level: All

Note:

MADDS64 (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	000	Dt	0	Ra	Rb	0000	MADDS64 101010

Syntax: MADDS64 Dt, Ra, Rb

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

Description: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The 64-bit multiplication result is added with the content of Dt data register. And the final result is written 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:

MFSR (Move From System Register)

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: It is used to move the content of a system register into a general register.

Description:

The content of the system register specified by the SRIDX will be moved into the general register Rt.

Operations:

```
GR[Rt] = SR[SRIDX];
```

Exceptions: Privileged Instruction

Privilege level: Superuser and above

Note:

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: The content of a User Special Register specified by USR and Group is moved into a general register Rt. The USR definition is defined in the following tables.

Table 52 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 of group 1 registers 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 Privileged Instruction exception. Privileged software should provide means for a user mode program to request such access permission. Reading of reserved registers will cause Reserved Instruction exception. Please check Andes Privileged Architecture specification for detailed definitions of Group 1 USR registers.

Table 53 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 of group 2 registers 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 Privileged Instruction exception. Privileged software should provide means for a user mode program to request such access permission. Reading of reserved registers will cause Reserved Instruction exception. Please check Andes Privileged Architecture specification for detailed definitions of Group 2 USR registers.

Table 54 Group 2 MFUSR 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:

```
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)]
```

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: To initialize a register with a constant.

Description: Move the sign-extended imm20s into general register Rt.

Operations:

```
Rt = SE(imm20s);
```

Exceptions: None

Privilege level: All

Note:

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. The final result is written back to the lower 32-bit of the 64-bit data register.

Description: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The lower 32-bit multiplication result is subtracted from the content of Dt.LO 32-bit data register. And the final result is written back to Dt.LO data register. The contents of Ra and Rb can be either signed or unsigned integers.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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. The final result is written back to the 64-bit data register.

Description: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The 64-bit multiplication result is subtracted from the content of Dt data register. And the final result is written back to Dt data register. The contents of Ra and Rb are treated as unsigned integers.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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. The final result is written back to the 64-bit data register.

Description: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The 64-bit multiplication result is subtracted from the content of Dt data register. And the final result is written 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:

MSYNC (Memory Data Coherence Synchronization)

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 certain phase of memory load/store operations. This instruction is used to order loads and stores for synchronizing memory accesses between two and more Andes cores or an Andes core and the other 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 from these operations assuming that the hardware implementation will not automatically ensure the required ordering behavior. Hardware implementation is free to enforce more ordering behavior. And in such a case, this instruction becomes a NOP.

The following table lists the MSYNC SubType definitions:

Table 55 MSYNC SubType definitions

SubType	Name
0	All
1	Store
2 - 7	Reserved

If a Reserved SubType is used in this instruction, a Reserved Instruction Exception will be generated.

1. SubType 0, All:

For an implementation which does not support coherent caches, this operation ensures that all loads and *non-dirty* stores before this instruction completes before all loads and stores

after this instruction can start. *Non-dirty* store includes non-cacheable store, and cacheable store which has been written back implicitly or explicitly.

For an implementation which support coherent caches, this operation ensures that all loads and stores before this instruction completes before all loads and stores after 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 before this instruction completes before all loads and stores after this instruction can start. *Non-dirty* store includes non-cacheable store, and cacheable store which has been written back implicitly or explicitly.

For an implementation which support coherent caches, this operation ensures that all stores before this instruction completes before all loads and stores after 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

Notes:

MTSR (Move To System Register)

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: It is used to move the content of a general register into a system register.

Description:

The content of the general register Ra will be moved into the system register specified by the SRIDX.

Operations:

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

Exceptions: Privileged Instruction

Privilege level: Superuser and above

Note:

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	MFUSR 100001

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

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

Description: The content of a general register Rt is moved into a User Special Register specified by USR and Group. The USR definition is defined in the following tables.

Table 56 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 of group 1 registers 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 Privileged Instruction exception. Privileged software should provide means for a user mode program to request such access permission. Writing of reserved registers will cause Reserved Instruction exception. Please check Andes Privileged Architecture specification for detailed definitions of Group 1 USR registers. The data dependency serializations of group 1 registers between MTUSR DMA_CHNSEL and MTUSR <Channel register> or between MTUSR and MFUSR requires DSB instruction inserted in the middle. Please see Andes Privileged Architecture specification for more details.

Table 57 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 of group 2 registers 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 Privileged Instruction exception. Privileged software should provide means for a user mode program to request such access permission. Writing of reserved registers will cause Reserved Instruction exception. Please check Andes Privileged Architecture specification for detailed definitions of Group 2 USR registers. The data dependency serializations of group 2 registers between MTUSR PFM_CTL and MTUSR <PFMCx register> or between MTUSR and MFUSR requires DSB instruction inserted in the middle. Please see Andes Privileged Architecture specification for more details.

Table 58 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:

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: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The lower 32-bit of the multiplication result is written 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:

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	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: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The lower 32-bit multiplication result is written 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:

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	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: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The 64-bit multiplication result is written to Dt data register. The contents of Ra and Rb are treated as unsigned integers.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The 64-bit multiplication result is written 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:

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: Do nothing. This instruction is aliased to "`SRLI R0,R0,0`", but will be treated by an implementation as a true NOP.

Operations:

```
;
```

Exceptions: None

Privilege level: All

Note:

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

Description: The content of Ra is combined with the content of Rb using a bit-wise logical NOR operation. And the result is written to Rt.

Operations:

```
Rt = ~(Ra | Rb);
```

Exceptions: None

Privilege level: All

Note:

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

Description: The content of Ra is combined with the content of Rb using a bit-wise logical OR operation. And the result is written to Rt.

Operations:

```
Rt = Ra | Rb;
```

Exceptions: None

Privilege level: All

Note:

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: The content of Ra is bit-wise ORed with the zero-extended imm15u. And the result is written to Rt.

Operations:

```
Rt = Ra | ZE(imm15u);
```

Exceptions: None

Privilege level: All

Note:

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: Unconditional function call return to an instruction address stored in a register.

Description: Branch unconditionally to an instruction address stored in Rb. Note that the architecture behavior of this instruction is the same as the JR instruction. But software will use this instruction instead of JR for function call return purpose. This facilitates software's need to distinguish the two different usages which is helpful in call stack backtracing applications. Distinguishing a function return jump from a regular jump will also help on implementation performance (e.g. return address prediction).

Operations:

```
PC = Rb;
```

Exceptions: None

Privilege level: All

Note:

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

Description: Branch unconditionally to an instruction address stored in Rb and 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 will see PSW.IT as 0 (and PSW.DT as 0 if included), thus will not go through the address translation process. Note that the architecture behavior of this instruction is the same as the JR.xTOFF instruction. But software will use this instruction instead of JR.xTOFF for function call return purpose. This facilitates software's need to distinguish the two different usages which is helpful in call stack backtracing applications. Distinguishing a function return jump from a regular jump will also help on 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 which is in a non-translated address space. Please see JRAL.xTON instruction for the reverse process of coming from a non-translated address space to a translated address space.

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 right rotation operation on the content of a register.

Description: The content of Ra is right-rotated. The rotation amount is specified by the low-order 5-bits of the Rb register. And the result is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 right rotation operation on the content of a register.

Description: The content of Ra is right-rotated. The rotation amount is specified by the imm5u constant. And the result is written to Rt.

Operations:

```
Rt = CONCAT(Ra(imm5u-1,0), Ra(31,imm5u));
```

Exceptions: None

Privilege level: All

Note:

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: To store an 8-bit byte from a general register into memory.

Description: The least-significant 8-bit byte in the general register Rt is stored to the memory. Two different forms are used 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 will be 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, UserMode, 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:

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: To store an 8-bit byte from a general register into a memory location.

Description: The least-significant 8-bit byte in the general register Rt is stored to the memory location. Two different forms are used 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 will be 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, UserMode, 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:

SCW (Store Conditional Word)

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: It is used as a primitive to perform atomic read-modify-write operations.
Description: The 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, then 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 exists per processor at any one time. And if a new LLW instruction is issued before an active sequence is completed by a SCW instruction, the new LLW instruction will start a new sequence which replaces the previous sequence.

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, then 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, then the store operation will not be performed. And a result of 0 is written to the general register Rt to indicate a fail 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, then whether the store operation will be performed or not is implementation-dependent (i.e. UNPREDICTABLE). However, the final success or fail status 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 bytes memory region and its exact size is implementation-dependent, but 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 all success or fail conditions.

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

Operations:

```
VA = Ra + Rb;
If (VA(1,0) != 0) {
   Generate_Exception(Data_alignment_check);
}
(PA, Attributes) = Address_Translation(VA, PSW.DT);
Excep_status = Page_Exception(Attributes, UserMode, 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: Since SCW instruction is defined to conditionally execute in the memory system, depending on the cache coherence design in the memory system, an implementation may need to prevent SCW from being invalidated by any interruption during the period after the SCW request has entered the memory system until the success/fail status has returned from the memory system to complete the SCW instruction.

For an implementation with a non-coherent cache system, an internal and an external flags may be needed to implement the “lock” state. For such a system, the success or fail of 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

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

Additional Software Constraints:

For N1213 hardcore N1213_43U1HA0 (CPU_VER==0x0C010003), additional software constraints must be followed to ensure correct LLW/SCW operations.

- If LLW/SCW are used in an interruption handler, it must be followed that
  - Execution of a LLW instruction must lead to execution of a SCW instruction. UPREDICTABLE result may happen if execution of a LLW instruction eventually leads to execution of an IRET instruction without going through a SCW instruction.

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: The least-significant-byte of Ra is sign-extended. And the result is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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: The least-significant-halfword of Ra is sign-extended. And the result is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

SETEND (Set data endian)

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: It is used to control the data endian mode in the PSW register.

Description:

This instruction has two flavors. The SETEND.B will set the data endian mode to big-endian while the SETEND.L will set 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 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.

SETGIE (Set global interrupt enable)

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: It is used to control the global interrupt enable bit in the PSW register.

Description:

This instruction has two flavors. The SETGIE.E will enable the global interrupt while the SETGIE.D will disable the global interrupt. The EN bit in this instruction encoding distinguishes 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 behavior.

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: To initialize the high portion of a register with a constant.

Description: Move the imm20u into the upper 20-bits of general register Rt. The lower 12-bits of Rt will be filled with 0.

Operations:

```
Rt = imm20u << 12;
```

Exceptions: None

Privilege level: All

Note:

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: To store a 16-bit halfword from a general register into memory.

Description: The least-significant 16-bit halfword in the general register Rt is stored to the memory. Two different forms are used 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 will be updated with the Ra + (Rb << sv) value after the memory store operation.

The memory address has to be halfword-aligned. Otherwise, a Data Alignment Check exception will be generated.

Operations:

```
Addr = Ra + (Rb << sv);
If (.bi form) {
   Vaddr = Ra;
} else {
   Vaddr = Addr;
}
if (!Halfword_Aligned(Vaddr)) {
   Generate_Exception(Data_alignment_check);
```

```
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, UserMode, 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:

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: To store a 16-bit halfword from a general register into memory.

Description: The least-significant 16-bit halfword in the general register Rt is stored to the memory. Two different forms are used 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 will be 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, a Data Alignment Check exception will be generated.

Operations:

```
Addr = Ra + Sign_Extend(imm15s << 1);
If (.bi form) {
   Vaddr = Ra;
} else {
   Vaddr = Addr;
}
if (!Halfword_Aligned(Vaddr)) {
```

```
   Generate_Exception(Data_alignment_check);
}
(PAddr, Attributes) = Address_Translation(Vaddr, PSW.DT);
Excep_status = Page_Exception(Attributes, UserMode, STORE);
If (Excep_status == NO_EXCEPTION) {
   Store_Memory(PAddr, WORD, 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:

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 logical left shift operation on the content of a register.

Description: The content of Ra is left-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 5-bits of the Rb register. And the result is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 logical left shift operation on the content of a register.

Description: The content of Ra is left-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm5u constant. And the result is written to Rt.

Operations:

```
Rt = CONCAT(Ra(31-imm5u,0), imm5u`b0);
```

Exceptions: None

Privilege level: All

Note:

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 comparison between the contents of two registers.

Description: If the content of Ra is less than (unsigned comparison) the content of Rb, a result of 1 is written to Rt; otherwise, a result of 0 is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 comparison between the content of a register and 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, the result of 1 will be written to Rt; otherwise, the result of 0 will be written 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 (unsigned)< SE(imm15s)) ? 1 : 0;
```

Exceptions: None

Privilege level: All

Note:

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 comparison between the contents of two registers.

Description: If the content of Ra is less than (signed comparison) the content of Rb, a result of 1 is written to Rt; otherwise, a result of 0 is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 comparison between the content of a register and 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, the result of 1 will be written to Rt; otherwise, the result of 0 will be written to Rt. The sign-extended imm15s will generate a signed constant in the following range:

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

Operations:

```
Rt = (Ra (signed)< SE(imm15s)) ? 1 : 0;
```

Exceptions: None

Privilege level: All

Note:

SMW (Store Multiple Word)

Format:

31	30 25	24 20	19 15	14 10	9 6	5	4	3	2	1 0
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: Store 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 registers list formed by Rb, Re, and the four-bit Enable4 field as follows.

<Registers 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} use the contents of Ra as the first memory store address. {a} use 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 this register range. 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 <Registers 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 result will happen to the contents of the memory range pointed to by the base register and the base register itself if the {m?} option is specified after this instruction.
- The register is 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 <Registers Specification>, the value stored to the 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, UserMode, 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. And they do not guarantee single access to a memory location during the execution either. 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 implemented by an implementation is:

- For LMW/SMW.i : increasing memory addresses from base address.
- For LMW/SMW.d: decreasing memory addresses from 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 implemented by 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 or .
- 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 will more likely 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".

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 arithmetic right shift operation on the content of a register.

Description: The content of Ra is right-shifted arithmetically, that is, the shifted out bits are filled with the sign-bit Ra(31). The shift amount is specified by the low-order 5-bits of the Rb register. And the result is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 arithmetic right shift operation on the content of a register.

Description: The content of Ra is right-shifted arithmetically, that is, the shifted out bits are filled with sign-bit Ra(31). The shift amount is specified by the imm5u constant. And the result is written to Rt.

Operations:

```
Rt = CONCAT(imm5u`bRa(31), Ra(31,imm5u));
```

Exceptions: None

Privilege level: All

Note:

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 logical right shift operation on the content of a register.

Description: The content of Ra is right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the low-order 5-bits of the Rb register. And the result is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 logical right shift operation on the content of a register.

Description: The content of Ra is right-shifted logically, that is, the shifted out bits are filled with zero. The shift amount is specified by the imm5u constant. And the result is written to Rt.

Operations:

```
Rt = CONCAT(imm5u`b0, Ra(31,imm5u));
```

Exceptions: None

Privilege level: All

Note: "`SRLI R0,R0,0`" will be aliased to NOP and treated by an implementation as NOP.

STANDBY (Wait For External Event)

Format:

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

* N12 implementation for this instruction, please refer to chapter 9.2

Syntax: STANDBY SubType (= no_wake_grant, wake_grant)

Purpose: It is used for a core to enter a 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 that 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 may need the external power manager's consent. Thus two flavors of STANDBY instruction are defined to distinguish the different usages. The SubType field definitions are listed as follows.

Table 59 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) to 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 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. And the instruction execution restarts either from the instruction following the STANDBY instruction or from the enabled interrupt handler which cause the core to leave the standby mode. When entering an interrupt handler, the IPC system register will have 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 any such clock or voltage regulation causes any core/memory state loss, software is responsible to preserve the needed states before the core enters standby mode. And 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:

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: The content of Rb is subtracted from the content of Ra. And the result is written to Rt.

Operations:

```
Rt = Ra - Rb;
```

Exceptions: None

Privilege level: All

Note:

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: The content of Ra is subtracted from the sign-extended imm15s. And the result is written to Rt.

Operations:

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

Exceptions: None

Privilege level: All

Note:

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 contents of two registers.

Description: If adding the contents of Ra and Rb results in 32-bit 2's complement arithmetic overflow condition, a result of 1 is written to Rt; otherwise, a result of 0 is written 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:

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 contents of two registers.

Description: If subtracting the contents of Ra and Rb results in 32-bit 2's complement arithmetic overflow condition, a result of 1 is written to Rt; otherwise, a result of 0 is written 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:

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: To store a 32-bit word from a general register into memory.

Description: This instruction stores a word from the the general register Rt into the memory. Two different forms are used 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 will be updated with the Ra + (Rb << sv) value after the memory store operation.

The memory address has to be word-aligned. Otherwise, a Data Alignment Check exception will be generated.

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, UserMode, 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:

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: To store a 32-bit word from a general register into memory.

Description: This instruction stores a word from the general register Rt into the memory. Two different forms are used 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 will be 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, a Data Alignment Check exception will be generated.

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, UserMode, 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:

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: To 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, a Data Alignment Check exception will be generated.

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, UserMode, 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:

SYSCALL (System Call)

Format:

31	30 25	24 20	19 5	4 0
0	MISC 110010	00000	SWID	SYSCALL 01011

Syntax: SYSCALL SWID

Purpose: It is used to generate a System Call exception.

Description:

SYSCALL instruction will unconditionally generate a System Call exception and transfer control to the System Call exception handler. The 15-bits SWID is used by software as a parameter to distinguish different system call services.

Operations:

```
Generate_Exception(System_Call);
```

Exceptions: System Call

Privilege level: All

Note:

TEQZ (Trap if equal 0)

Format:

31	30 25	24 20	19 5	4 0
0	MISC 110010	Ra	SWID	TEQZ 00110

Syntax: TEQZ Ra, SWID

Purpose: It is used to generate a conditional Trap exception.

Description:

TEQZ instruction will generate a Trap exception and transfer control to the Trap exception handler if the content of Ra is equal to 0. The 15-bits SWID is used by software as a parameter to distinguish different trap features and usages.

Operations:

```
If (GR[Ra] == 0)
    Generate_Exception(Trap);
```

Exceptions: Trap

Privilege level: All

Note:

TNEZ (Trap if not equal 0)

Format:

31	30 25	24 20	19 5	4 0
0	MISC 110010	Ra	SWID	TNEZ 00111

Syntax: TNEZ Ra, SWID

Purpose: It is used to generate a conditional Trap exception.

Description:

TNEZ instruction will generate a Trap exception and transfer control to the Trap exception handler if the content of Ra is not equal to 0. The 15-bits SWID is used by software as a parameter to distinguish different trap features and usages.

Operations:

```
If (GR[Ra] != 0)
    Generate_Exception(Trap);
```

Exceptions: Trap

Privilege level: All

Note:

TLBOP (TLB Operation)

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 is used to perform TLB control operations based on the SubType field. The definition and encoding for the SubType field is listed in the following table. Depending on the SubType, different TLBOP instructions have different number of operands from 0 to 2.

Table 60 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

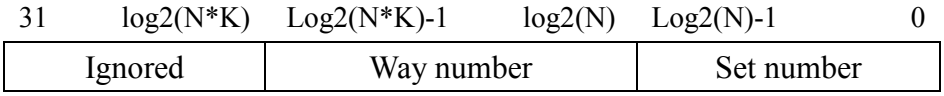
following categories:

a. TLB Target Read

```
Syntax: TLBOP  Ra,TargetRead
```

This operation reads a specified entry in the software-visible portion of the TLB structure. The specified entry is indicated by the Ra register. The read result is placed in the TLB_VPN, TLB_DATA, and TLB_MISC registers.

The TLB entry number for a non-fully-associative N sets K ways TLB cache is as follows:



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
```

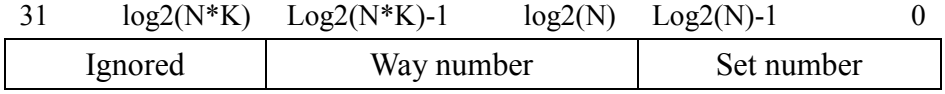
Important: Since the TLB_MISC register contains the current process's Context ID and Access Page Size information, any use of this instruction is required to save/restore the TLB_MISC register if you want the current process to run correctly immediately after this operation.

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:



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 is going to write into the TLB_DATA portion of the TLB structure. The other write operands are in the TLB_VPN and TLB_MISC registers.

If all the ways in the specified set is all locked during the write operation of this instruction, depending on the setting in the TBALCK field of the MMU Control system register (MMU_CTL), this operation may generate a precise or an imprecise “Data Machine Error” exception. Note that the default value of the TBALCK is to generate the 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.
iret/isb          // inst serialization barrier
```

d. **TLB Random Write and Lock**

`Syntax: TLBOP  Ra,RWriteLock`

This operation is similar to TLB Random Write operation to write 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 all the ways in the specified set is all locked during the write operation of this instruction, depending on the setting in the TBALCK field of the MMU Control system register (MMU_CTL), this operation may generate a precise or an imprecise "Data Machine Error" exception. Note that the default value of the TBALCK is to generate the exception.

e. TLB Unlock

`Syntax: TLBOP  Ra,Unlock`

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 and generates an entry number where the VA matches the VPN in that entry. The search result is written into the general register Rt. The search VA is specified in the general register Ra. The result that 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, the “sw” bit will be set. If the VA can be found in the software-invisible part of the TLB, the “hw” bit will be set. And if the VA cannot be found in either the software-visible or software-invisible part of the TLB, the “nf” bit will be set.

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,Invalidate`

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 flushing the entire TLB structure. And the exact behavior is implementation-dependent.

If this instruction encounters a multiple match condition when searching the TLB, all matched entries should be invalidated and no "Data Machine Error" exception will be generated.

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             // Inst serialization 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:

(1) All non-instruction-fetch-related exceptions generated by a TLBOP instruction should have the INST field of the ITYPE register set to 0.

TRAP (Trap exception)

Format:

31	30 25	24 20	19 5	4 0
0	MISC 110010	00000	SWID	TRAP 00101

Syntax: TRAP SWID

Purpose: It is used to generate an unconditional Trap exception.

Description:

TRAP instruction will unconditionally generate a Trap exception and transfer control to the Trap exception handler. The 15-bits SWID is used by software as a parameter to distinguish different trap features and usages.

Operations:

```
Generate_Exception(Trap);
```

Exceptions: Trap

Privilege level: All

Note:

WSBH (Word Swap Byte within Halfword)

Type: 32-Bit Baseline

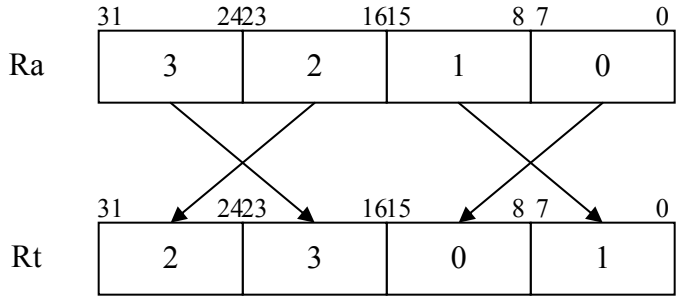
Format:

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: The bytes within each halfword of Ra is swapped. And the result is written to Rt.



Operations:

```
Rt = CONCAT(Ra(23,16),Ra(31,24),Ra(7,0),Ra(15,8));
```

Exceptions: None

Privilege level: All

Note:

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: Doing a bit-wise logical Exclusive OR operation on the content of two registers.

Description: The content of Ra is combined with the content of Rb using a bit-wise logical exclusive OR operation. And the result is written to Rt.

Operations:

```
Rt = Ra ^ Rb;
```

Exceptions: None

Privilege level: All

Note:

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: The content of Ra is bit-wise exclusive ORed with the zero-extended imm15u. And the result is written to Rt.

Operations:

```
Rt = Ra ^ ZE(imm15u);
```

Exceptions: None

Privilege level: All

Note:

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: The least-significant-byte of Ra is zero-extended. And the result is written to Rt.

Operations:

```
Rt = ZE(Ra(7,0));
```

Exceptions: None

Privilege level: All

Note:

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: The least-significant-halfword of Ra is zero-extended. And the result is written to Rt.

Operations:

```
Rt = ZE(Ra(15,0));
```

Exceptions: None

Privilege level: All

Note:

7.2 32-bit Performance Extension instructions

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. The result is written to Rt.

Operations:

```
if (Ra >= 0) {
   Rt = Ra;
} else {
   Rt = -Ra;
}
```

Exceptions: None

Privilege level: All

Note:

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 and rounds up a half-integer result to the nearest integer. The result is written 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:

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. The bit position is specified by the imm5u value. The cleared result is written to Rt.

Operations:

```
onehot = 1 << imm5u;
Rt = Ra & ~onehot;
```

Exceptions: None

Privilege level: All

Note:

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. The bit position is specified by the imm5u value. The modified result is written to Rt.

Operations:

```
onehot = 1 << imm5u;
Rt = Ra | onehot;
```

Exceptions: None

Privilege level: All

Note:

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. The bit position is specified by the imm5u value. The modified result is written to Rt.

Operations:

```
onehot = 1 << imm5u;
Rt = Ra ^ onehot;
```

Exceptions: None

Privilege level: All

Note:

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, the result of one is written to Rt. If the bit is cleared, the result of zero is written to Rt.

Operations:

```
onehot = 1 << imm5u;
Rt = Ra & onehot;
if (Rt == 0) {
   Rt = 0;
} else {
   Rt = 1;
}
```

Exceptions: None

Privilege level: All

Note:

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^{imm5u}-1 and 0. The limited result is written to Rt. For example, if imm5u is 0, the result should be always 0. If Ra is negative, then the result is 0 as well.

Operations:

```
if (Ra > 2^imm5u-1) {
   Rt = 2^imm5u-1;
} else if (Ra < 0) {
   Rt = 0;
} else {
   Rt = Ra;
}
```

Exceptions: None

Privilege level: All

Note:

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^{imm5u}-1 and -2^{imm5u}. The limited result is written to Rt. For example, if imm5u is 3, the result should be between 7 and -8.

Operations:

```
if (Ra > 2^imm5u-1) {
   Rt = 2^imm5u-1;
} else if (Ra < -2^imm5u) {
   Rt = -2^imm5u;
} else {
   Rt = Ra;
}
```

Exceptions: None

Privilege level: All

Note:

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 leading from the most significant bit of a general register..

Description: Starting from the most significant bit (bit 31)of Ra, count the number of successive ones. The result is written to Rt. For example, if bit 31 of Ra is 0, the result is 0. If Ra has a value of 0xFFFFFFFF, then 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:

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 leading from the most significant bit of a general register..

Description: Starting from the most significant bit (bit 31)of Ra, count the number of successive zeros. The result is written to Rt. For example, if bit 31 of Ra is 1, the result is 0. If Ra has a value of 0, then 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:

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 of the contents of two general registers.

Description: This instruction compares two signed integers stored in Ra and Rb and picks the larger value as the result. The result is written to Rt.

Operations:

```
if (Ra >= Rb) {
   Rt = Ra;
} else {
   Rt = Rb;
}
```

Exceptions: None

Privilege level: All

Note:

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 of the contents of two general registers.

Description: This instruction compares two signed integers stored in Ra and Rb and picks the smaller value as the result. The result is written to Rt.

Operations:

```
if (Ra >= Rb) {
   Rt = Rb;
} else {
   Rt = Ra;
}
```

Exceptions: None

Privilege level: All

Note:

7.3 32-bit Performance Extension Version 2 instructions

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: It is used to extract a number of bits from a register for bit stream extraction.
Description:
BSE 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 non-occupied bits in Rt will be filled with 0, otherwise the non-occupied bits in Rt will be untouched. 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 non-occupied bits in Rt is untouched if Rb(30) is equal to 1, the Rb(12,8) will be updated 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) eaual to 1 is illustrated in Figure 4.

If the sum of Rb(4,0) (=N) and Rb(12,8) (=M) is equal to 31, then all remaining M+1 bits in Ra will be extracted, and Rb(5,0) will be set 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 is used for re-adjusting the Rb(4,0) if some bits that has been extracted will be re-extracted again.

If the sum of Rb(4,0) (=N) and Rb(12,8) (=M) is greater than 31, then all remaining 32-N bits in Ra will be extracted to Rt(M,M+N-31), and Rt(M+N-32,0) will be filled with 0. In this case, Rb(5,0) will be set to 0x20 and Rb(12,8) will be set 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, Rb(30), acted as an "underflow" flag, will be set to 1 indicating that not all required bits are extracted, and 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, since both registers will be updated to different data, UNPREDICTABLE result will be written to the register.

Figure 3. Basic BSE operation with Rb(30) == 0

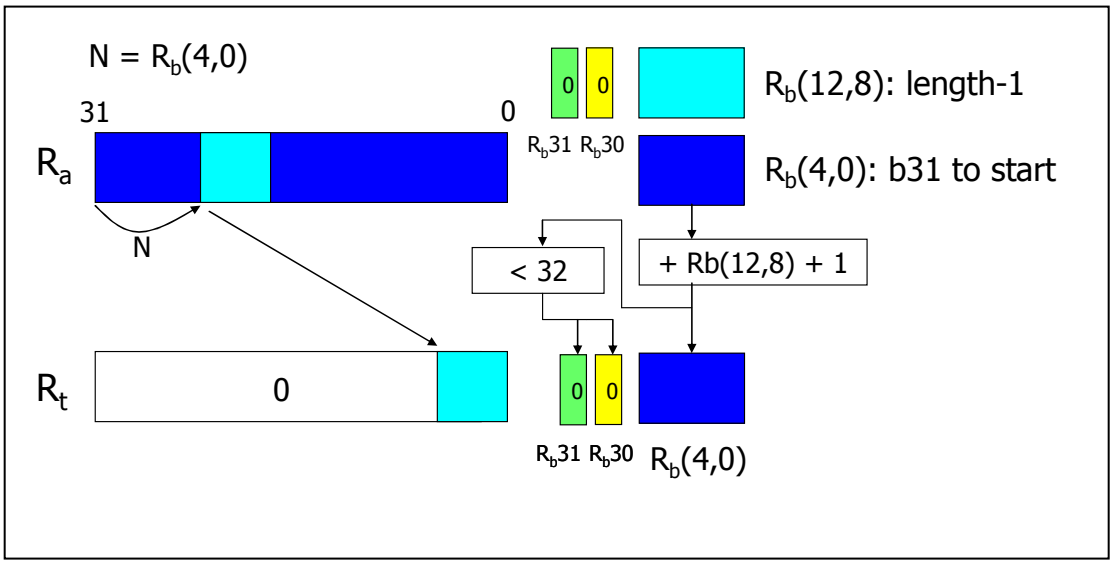


Figure 4. Basic BSE operation with Rb(30) == 1

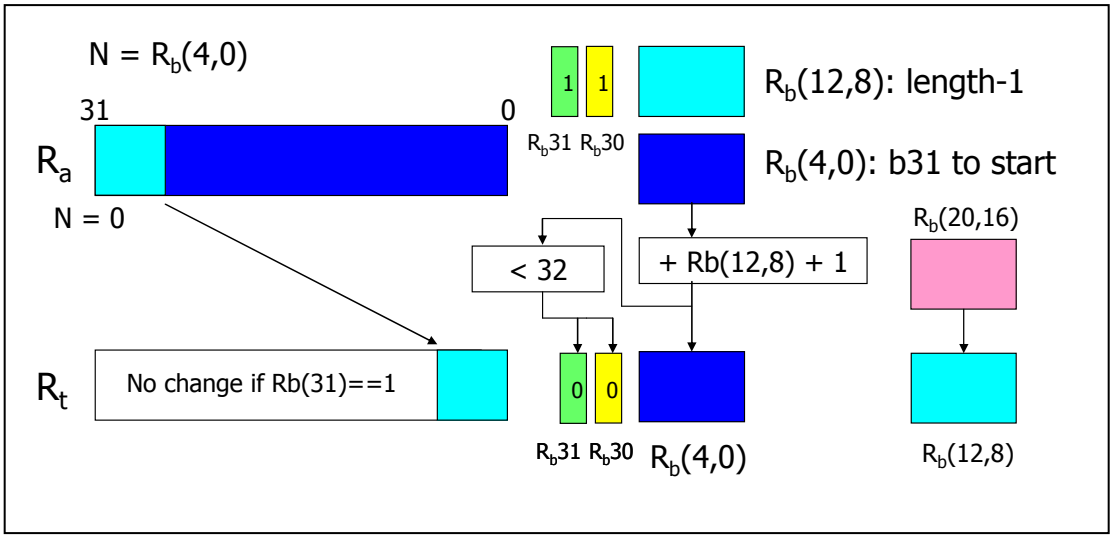


Figure 5. BSE operation extracting all remaining bits with Rb(30) == 0.

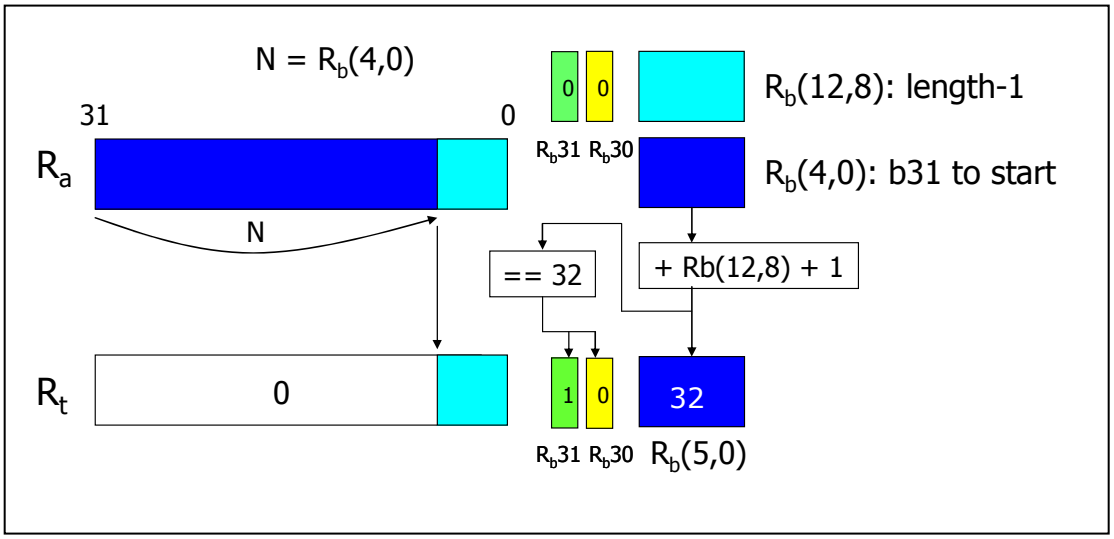
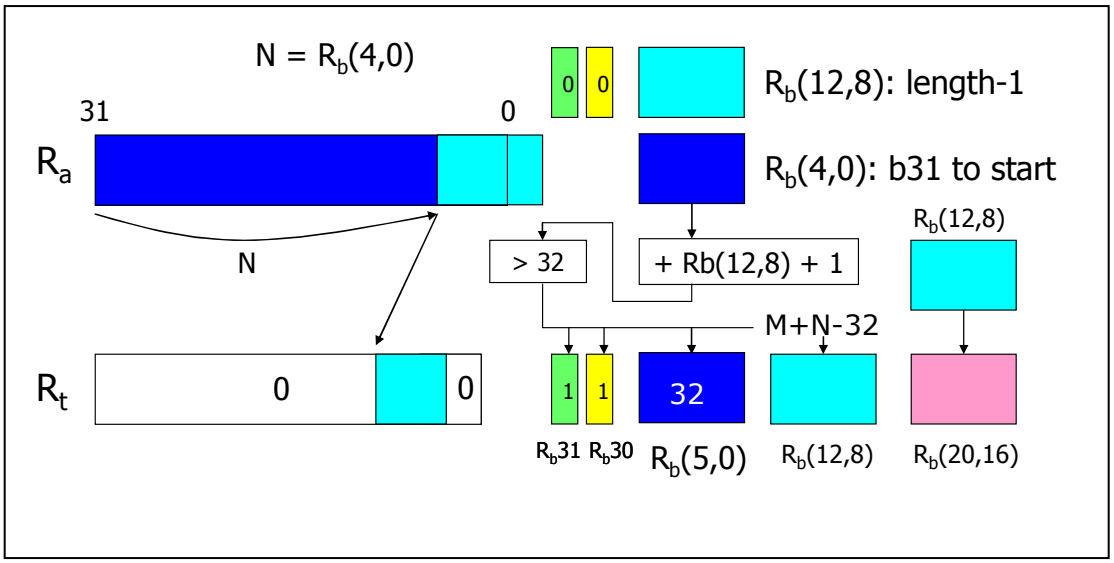


Figure 6. BSE operation with the "underflow" condition with Rb(30) == 0.



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];
} 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. A normal multiple-bits extraction that follows can be performed using several baseline instructions:
   - 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 will save at least 3 instructions when compared to a normal extraction flow.

Following is an example 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
```

2. BSP instructions can be used to update the Rb(12,8) and Rb(4,0) values.
3. Adjustment of Rb(4,0) value can be performed using basic addition and subtraction.

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: It is used to insert a number of bits to a register for bit stream packing.
Description:
BSP 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, indicating that a previous BSP operation is “overflowed”, in addition to the usual packing operation, Rb(12,8) will be updated 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, then all M+1 bits in Ra will be inserted to Rt and fill up all bits in Rt, and Rb(5,0) will be set 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 packing in Ra. In this case, only 32-N bits will be inserted from Ra to Rt and the remaining M+N-31 bits in Ra will need to be inserted in the next BSP instruction. In this case, Rb(5,0) will be set to

0x20 and Rb(12,8) will be set 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, R(30), acted as an "overflow" flag, will be set to 1 indicating that not all required bits are packed; and 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, since both registers will be updated to different data, UNPREDICTABLE result will be written to the register.

Figure 7. Basic BSP operation

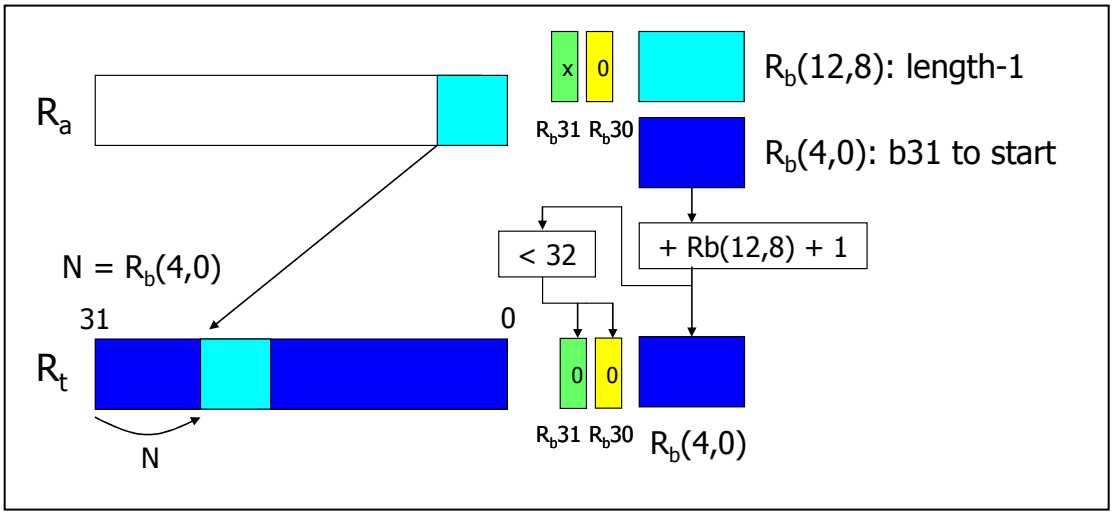


Figure 8. BSP operation with Rb(30) == 1

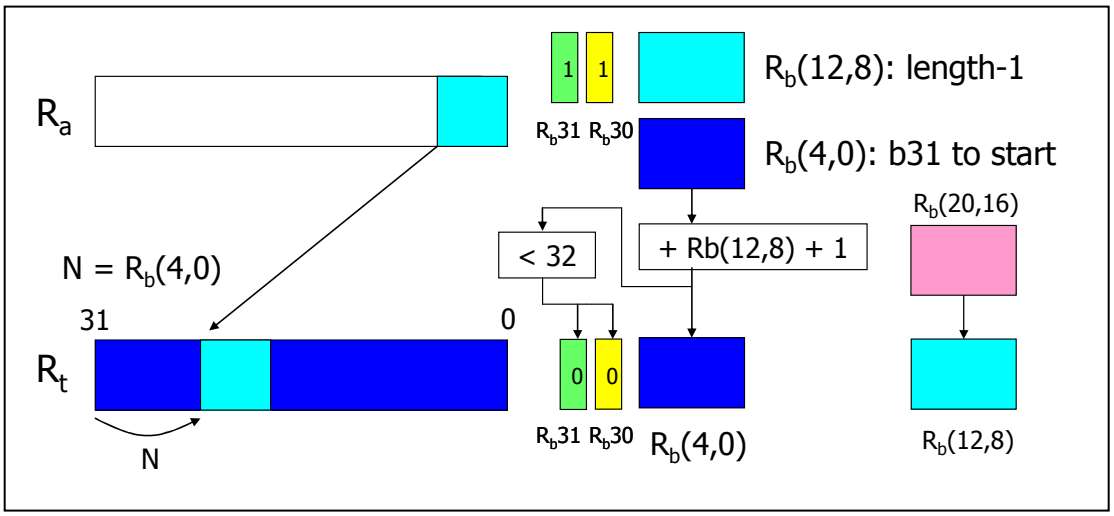


Figure 9. BSP operation filling up Rt.

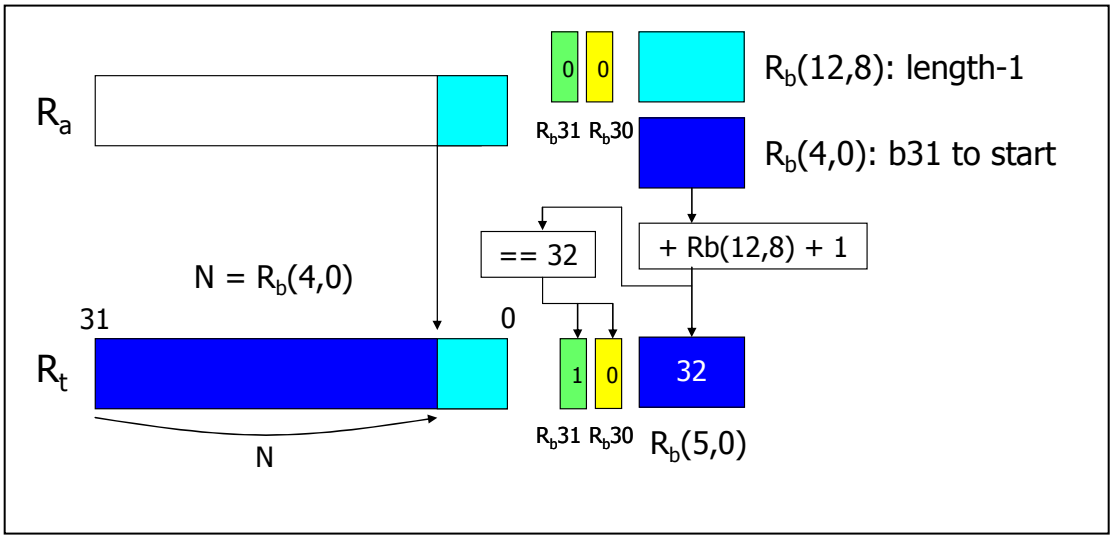
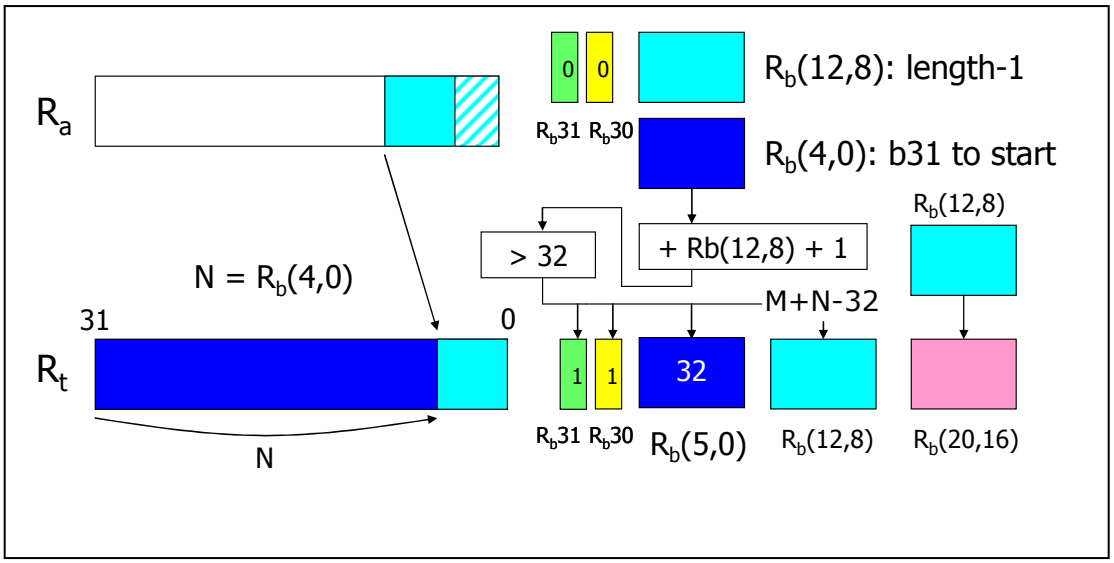


Figure 10. BSP operation with the "overflow" condition.



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. A normal multiple-bit packing operation that follows can be performed using several baseline instructions:
   - 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 will save at least 3 instructions when compared to a normal bit-packing flow.

Following is an example 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. BSP instructions can be used to update the Rb(12,8) and Rb(4,0) values.
3. Adjustment of Rb(4,0) value can be performed using basic addition and subtraction.

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: The four un-signed 8-bit elements of Ra are subtracted from the four unsigned 8-bit elements of Rb. The absolute value of each difference is added together and the result is written 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:

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: The four un-signed 8-bit elements of Ra are subtracted from the four unsigned 8-bit elements of Rb. The absolute value of each difference is added together along with the content of Rt. The accumulated result is written 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:

7.4 32-bit STRING Extension instructions

This STRING extension is currently reserved for Andes Technology internal use. It will be released in the future.

7.5 16-bit Baseline instructions

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: It is used to add the contents of two registers.

Description: For ADD333, the contents of Ra3 and Rb3 are added. And the result is written to Rt3. For ADD45, the contents of Rt4 and Rb5 are added. And the result is written to the source register Rt4.

Operations:

```
Rt3 = Ra3 + Rb3;  // ADD333
Rt4 = Rt4 + Rb5;  // ADD45
```

Exceptions: None

Privilege level: All

Note:

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: It is used to add a zero-extended immediate into the content of a register.

Description: For ADDI333, the zero-extended 3-bit immediate “imm3u” is added to the content of Ra3. And the result is written to Rt3. For ADDI45, the zero-extended 5-bit immediate is added to the content of Rt4. And the result is written to the source register Rt4.

Operations:

```
Rt3 = Ra3 + ZE(imm3u);     // ADDI333
Rt4 = Rt4 + ZE(imm5u);     // ADDI45
```

Exceptions: None

Privilege level: All

Note:

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: It is used for conditional PC-relative branching based on the result of comparing the contents of a register with the content of the implied R5.

Description: If the content of the implied register R5 is equal to the content of Rt3 (#Rt3 != 5), then branch to the target address of adding the current instruction address with 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:

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: It is used for 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, then branch to the target address of adding the current instruction address with 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:

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: It is used for 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, then branch to the target address of adding the current instruction address with 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:

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: It is used for conditional PC-relative branching based on the result of comparing the content of a register with the content of the implied R5.

Description: If the content of the implied register R5 is not equal to the content of Rt3 (#Rt3 != 5), then branch to the target address of adding the current instruction address with 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:

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: It is used for 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, then branch to the target address of adding the current instruction address with 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:

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: It is used for 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, then branch to the target address of adding the current instruction address with 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:

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: It is used to generate a Breakpoint exception.

Description:

BREAK16 instruction will unconditionally generate a Breakpoint exception and transfer control to the Breakpoint exception handler. The 9-bits SWID is used by software as a parameter to distinguish different breakpoint features and usages.

Operations:

```
Generate_Exception(Breakpoint);
```

Exceptions: Breakpoint

Privilege level: All

Note:

J8 (Jump Immediate)

Type: 16-Bit Baseline

Format:

15	14 8	7 0
1	J8 1010101	imm8s

Syntax: J8 imm8s

32-bit Equivalent: J SE(imm8s)

Purpose: It is used for unconditional PC-relative branching.

Description: Jump unconditionally to the target address of adding the current instruction address with 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:

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: It is used for unconditional branching to an address stored in a general register.

Description: Jump unconditionally to the target address stored in the register Rb5.

Operations:

```
TAddr = Rb5;
PC = TAddr;
```

Exceptions: None

Privilege level: All

Note:

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: It is used for unconditional branching to a function call.

Description: Jump unconditionally to the target address stored in the register Rb5. And the next sequential address (PC + 2) of the current instruction is written to R30.

Operations:

```
TAddr = Rb5;
R30 = PC + 2;
PC = TAddr;
```

Exceptions: None

Privilege level: All

Note:

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: It is used to 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 with the zero-extended imm3u value. The loaded byte is zero-extended to the width of the general register and then written into 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:

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: It is used to 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 with the zero-extended (imm3u << 1) value. The loaded halfword is zero-extended to the width of the general register and then written into Rt3. Notice that imm3u is a halfword-aligned offset.

The memory address has to be halfword-aligned. Otherwise, a Data Alignment Check exception will be generated.

Operations:

```
VAddr = Ra3 + Zero_Extend((imm3u << 1));
if (!Halfword_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) {
   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:

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: It is used to load a 32-bit word from memory into a general register.

Description: This instruction loads a word from the memory into the general register Rt3. Two different forms are used 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 will be updated with the Ra3 + ZE(imm3u << 2) value after the memory load operation.

The memory address has to be word-aligned. Otherwise, a Data Alignment Check exception will be generated.

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:

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: It is used to load a 32-bit word from memory into a general register.

Description: This instruction loads a word from the memory into the general register Rt3. The memory address is specified by adding the implied FP (i.e. R28) register with the zero-extended (imm7u << 2) value. Notice that the imm7u is a word-aligned offset. The memory address has to be word-aligned. Otherwise, a Data Alignment Check exception will be generated.

Operations:

```
VAddr = FP (i.e. R28) + Zero_Extend(imm7u << 2);
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);
} 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:

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: It is used to load a 32-bit word from memory into a general register.

Description: This instruction loads a word from the memory into the general register Rt4. The memory address is specified in Ra5.

The memory address has to be word-aligned. Otherwise, a Data Alignment Check exception will be generated.

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, 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:

MOV55 (Move Register)

Type: 16-Bit Baseline

Format:

15	14 10	9 5	4 0
1	MOVI55 00000	Rt5	Ra5

Syntax: MOVI55 Rt5, Ra5

32-bit Equivalent: ADDI/ORI Rt5, Ra5, 0

Purpose: It is used to move contents between general-purpose registers.

Description: The content of Ra5 is moved into Rt5.

Operations:

```
Rt5 = Ra5;
```

Exceptions: None

Privilege level: All

Note:

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: It is used to move a sign-extended immediate into a general-purpose register.

Description: The sign-extended 5-bit immediate “imm5s” is moved into Rt5.

Operations:

```
Rt5 = Sign_Extend(imm5s);
```

Exceptions: None

Privilege level: All

Note:

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: It is used to do no operation. It may be used to 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:

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: It is used for unconditional function call return to an address stored in a general register.

Description: Jump unconditionally to the target address stored in the register Rb5. Note that the architecture behavior of this instruction is the same as the JR5 instruction. But software will use this instruction instead of JR5 for function call return purpose. This facilitates software's need to distinguish the two different usages which is helpful in call stack backtracing applications. Distinguishing a function return jump from a regular jump will also help on implementation performance (e.g. return address prediction).

Operations:

```
TAddr = Rb5;
PC = TAddr;
```

Exceptions: None

Privilege level: All

Note:

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: It is used to store an 8-bit byte from a general register into a memory location.

Description: The least-significant 8-bit byte in the general register Rt3 is stored to the memory location whose address is specified by adding the content of Ra3 with 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:

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: It is used to sign-extend the least-significant byte of Ra3 and the result is written to Rt3.

Description: The least-significant byte of Ra3 is sign-extended to the width of a general register. And the result is written to the register Rt3.

Operations:

```
Rt3 = Sign_Extend(Ra3(7,0));
```

Exceptions: None

Privilege level: All

Note:

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: It is used to sign-extend the least-significant halfword of Ra3 and the result is written to Rt3.

Description: The least-significant halfword of Ra3 is sign-extended to the width of a general register. And the result is written to the register Rt3.

Operations:

```
Rt3 = Sign_Extend(Ra3(15,0));
```

Exceptions: None

Privilege level: All

Note:

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: It is used to store a 16-bit halfword from a general register into a memory location.

Description: The least-significant 16-bit halfword in the general register Rt3 is stored to the memory location whose address is specified by adding the content of Ra3 with the zero-extended (imm3u << 1) value. Notice that imm3u is a halfword-aligned offset. The memory address has to be halfword-aligned. Otherwise, a Data Alignment Check exception will be generated.

Operations:

```
VAddr = Ra3 + Zero_Extend(imm3u << 1);
if (!Halfword_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, 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:

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: It is used to left-shift a register content to a fixed number of bits in the range of 0 to 7.

Description: The content of Ra3 is shifted left by a fixed number of bits between 0 and 7 specified by the “imm3u”. And zero will be shifted in to fill the shifted-out bits.
The shifted result is written to Rt3.

Operations:

```
Rt3 = Ra3 << imm3u;   // or
Rt3 = Ra3(31-imm3u,0).Dupliate(0, imm3u);
```

Exceptions: None

Privilege level: All

Note:

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: It is used to 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, then R15 will be written with 1; otherwise, R15 will be written with 0.

Operations:

```
If (Ra4 (unsigned)< Rb5) {
   R15 = 1;
} else {
   R15 = 0;
}
```

Exceptions: None

Privilege level: All

Note:

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: It is used to 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, then R15 will be written with 1; otherwise, R15 will be written with 0.

Operations:

```
If (Ra4 (unsigned)< Zero_Extend(imm5u)) {
   R15 = 1;
} else {
   R15 = 0;
}
```

Exceptions: None

Privilege level: All

Note:

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: It is used to 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, then R15 will be written with 1; otherwise, R15 will be written with 0.

Operations:

```
If (Ra4 (signed)< Rb5) {
   R15 = 1;
} else {
   R15 = 0;
}
```

Exceptions: None

Privilege level: All

Note:

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: It is used to 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, then R15 will be written with 1; otherwise, R15 will be written with 0.

Operations:

```
If (Ra4 (signed)< Zero_Extend(imm5u)) {
   R15 = 1;
} else {
   R15 = 0;
}
```

Exceptions: None

Privilege level: All

Note:

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: It is used to 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: The content of Rt4 is shifted right by a fixed number of bits between 0 and 31 specified by the "imm5u". And the sign bit of Rt4, i.e. Rt4(31), will be duplicated to fill the shifted-out bits. The shifted result is written 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:

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: It is used to 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: The content of Rt4 is shifted right by a fixed number of bits between 0 and 31 specified by the "imm5u". And the shifted-out bits will be filled with zeros. The shifted result is written to the source register Rt4.

Operations:

```
Rt4 = Rt4 (0)>> imm5u;    // or
Rt4 = Duplicate(0, imm5u).Rt4(31,imm5u);
```

Exceptions: None

Privilege level: All

Note:

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: It is used to subtract the contents of two registers.

Description: For SUB333, the content of Rb3 is subtracted from Ra3. And the result is written to Rt3. For SUB45, the content of Rb5 is subtracted from Rt4. And the result is written to the source register Rt4.

Operations:

```
Rt3 = Ra3 - Rb3;  // SUB333
Rt4 = Rt4 - Rb5;  // SUB45
```

Exceptions: None

Privilege level: All

Note:

SUBI (Subtract Immediate)

Type: 16-Bit Baseline

Format:

SUBI333

15	14 9	8 6	5 3	2 0
1	SUBI333 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: It is used to subtract a zero-extended immediate from the content of a register.

Description: For SUBI333, the zero-extended 3-bit immediate "imm3u" is subtracted from the content of Ra3. And the result is written to Rt3. For SUBI45, the zero-extended 5-bit immediate is subtracted from the content of Rt4. And the result is written to the source register Rt4.

Operations:

```
Rt3 = Ra3 - ZE(imm3u);     // SUBI333
Rt4 = Rt4 - ZE(imm5u);     // SUBI45
```

Exceptions: None

Privilege level: All

Note:

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: It is used to store a 32-bit word from a general register into memory.

Description: This instruction stores a word from the general register Rt3 into the memory. Two different forms are used 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 will be updated with the Ra3 + ZE(imm3u << 2) value after the memory store operation.

The memory address has to be word-aligned. Otherwise, a Data Alignment Check exception will be generated.

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:

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: It is used to store a 32-bit word from a general register into memory.

Description: This instruction stores a word from the general register Rt3 into the memory. The memory address is specified by adding the implied FP (i.e. R28) register with the zero-extended (imm7u << 2) value. Notice that the imm7u is a word-aligned offset. The memory address has to be word-aligned. Otherwise, a Data Alignment Check exception will be generated.

Operations:

```
VAddr = FP (i.e. R28) + Zero_Extend(imm7u << 2);
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);
} 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:

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: It is used to store a 32-bit word from a general register into memory.

Description: This instruction stores a word from the general register Rt4 into the memory. The memory address is specified in Ra5.

The memory address has to be word-aligned. Otherwise, a Data Alignment Check exception will be generated.

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:

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: It is used to extract the least-significant 11 bits of Ra3 and the result is written to Rt3.

Description: The least-significant 11 bits of Ra3 is extracted. And the result is written to the register Rt3.

Operations:

```
Rt3 = Ra3 & 0x7FF;
```

Exceptions: None

Privilege level: All

Note:

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: It is used to extract the least-significant bit of Ra3 and the result is written to Rt3.

Description: The least-significant bit of Ra3 is extracted. And the result is written to the register Rt3.

Operations:

```
Rt3 = Ra3 & 0x1;
```

Exceptions: None

Privilege level: All

Note:

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: It is used to zero-extend the least-significant byte of Ra3 and the result is written to Rt3.

Description: The least-significant byte of Ra3 is zero-extended to the width of a general register. And the result is written to the register Rt3.

Operations:

```
Rt3 = Zero_Extend(Ra3(7,0));
```

Exceptions: None

Privilege level: All

Note:

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: It is used to zero-extend the least-significant halfword of Ra3 and the result is written to Rt3.

Description: The least-significant halfword of Ra3 is zero-extended to the width of a general register. And the result is written to the register Rt3.

Operations:

```
Rt3 = Zero_Extend(Ra3(15,0));
```

Exceptions: None

Privilege level: All

Note:

7.6 16-bit and 32-bit Baseline Version 2 instructions

These Baseline Version 2 instructions are added to improve code density.

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: It is used to add a sign-extended immediate into the content of the stack pointer register (R31).

Description: The sign-extended 10-bit immediate “imm10s” is added to the content of R31 (stack pointer). And the result is written back to R31.

Operations:

```
R31 = R31 + SE(imm10s);
```

Exceptions: None

Privilege level: All

Note:

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: It is used to load a 32-bit word from memory into a general register.

Description: This instruction loads a word from the memory into the general register Rt3. The memory address is specified by adding the implied SP (i.e. R31) register with the zero-extended (imm7u << 2) value. Notice that the imm7u is a word-aligned offset. The memory address has to be word-aligned. Otherwise, a Data Alignment Check exception will be generated.

Operations:

```
VAddr = SP (i.e. R31) + Zero_Extend(imm7u << 2);
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);
} 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:

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: It is used to store a 32-bit word from a general register into memory.

Description: This instruction stores a word from the general register Rt3 into the memory. The memory address is specified by adding the implied SP (i.e. R31) register with the zero-extended (imm7u << 2) value. Notice that the imm7u is a word-aligned offset. The memory address has to be word-aligned. Otherwise, a Data Alignment Check exception will be generated.

Operations:

```
VAddr = SP (i.e. R31) + Zero_Extend(imm7u << 2);
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);
} 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:

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 with a signed constant.

Description: The content of Gp (R29) is added with the sign-extended imm19s. And the result is written to Rt. The imm19s value will cover a range of 512K byte region relative to the location pointed to by the GP register.

Operations:

```
Rt = R29 + SE(imm19s);
```

Exceptions: None

Privilege level: All

Note:

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 the unsigned integer content of another register.

Description: Divide the 32-bit content of Ra with the 32-bit content of Rb. The 32-bit quotient result is written to Rt. And the 32-bit remainder result is written to Rs. The contents of Ra and Rb are treated as unsigned integers.

If the content of Rb is zero, an Arithmetic exception will be generated if the IDIVZE bit of the INT_MASK register is 1, which 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;
   Rs = remainder;
} else if (INT_MASK.IDIVZE == 0) {
   Rt = 0;
   Rs = 0;
} else {
   Generate_Exception(Arithmetic);
}
```

Exceptions: Arithmetic

Privilege level: All

Note:

DIVSR (Signed Integer Divide to Registers)

Type: 32-Bit Baseline Optional

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 the signed integer content of another register.

Description: Divide the 32-bit content of Ra with the 32-bit content of Rb. The 32-bit quotient result is written to Rt. And the 32-bit remainder result is written to Rs. The contents of Ra and Rb are treated as signed integers.

If the content of Rb is zero, an Arithmetic exception will be generated if the IDIVZE bit of the INT_MASK register is 1, which enables exception generation for the "Divide-By-Zero" condition. If the quotient overflows, an Arithmetic exception will always be generated. 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(Arithmetic);
   }
   remainder = Ra mod Rb
   Rt = quotient;
   Rs = remainder;
} else if (INT_MASK.IDIVZE == 0) {
   Rt = 0;
   Rs = 0;
} else {
```

```
        Generate_Exception(Arithmetic);
    }
```

Exceptions: Arithmetic

Privilege level: All

Note:

LBI.gp (GP-implied Load Byte Immediate)

Type: 32-Bit Baseline version 2

Format:

31	30 25	24 20	19	18 0
0	LBGP 010111	Rt	0 LBI	imm19s

Syntax: LBI.gp Rt, [+ imm19s]

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

Description: This instruction loads a zero-extended byte from the 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 will cover 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:

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: To load a sign-extended 8-bit byte from memory into a general register.

Description: This instruction loads a sign-extended byte from the 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 will cover 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:

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: To 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 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).

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:

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: To load a zero-extended 16-bit halfword from memory into a general register.

Description: This instruction loads a zero-extended halfword from the 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 will cover 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:

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: To load a sign-extended 16-bit halfword from memory into a general register.

Description: This instruction loads a sign-extended halfword from the 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 will cover 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:

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	1 0
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: load 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 specified by a registers list formed by Rb, Re, and the four-bit Enable4 field as follows.

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

- {b | a} option specifies the way how the first address is generated. {b} use the contents of Ra as the first memory load address. {a} use 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 this register range. 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 <Registers 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 result 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. And the rest of the loaded registers should have 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 loaded its own base register and the memory access to load the base register occurred 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:

(5) LMW and SMW instructions do not guarantee atomicity among individual memory access operations. And they do not guarantee single access to a memory location during the execution either. Any I/O access that has side-effects other than simple stable memory-like access behavior should not use these two instructions.

(6) 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 implemented by an implementation is:

- For LMW/SMW.i : increasing memory addresses from base address.
- For LMW/SMW.d: decreasing memory addresses from base address.

(7) 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 implemented by 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 or .

- 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.

(8) Based on the more likely access order of (2) and (3), upon interruption, the EVA register for un-aligned LMW/SMW will more likely 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".

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: To load a 32-bit word from memory into a general register.

Description: This instruction loads a 32-bit word from the 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 will cover 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:

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 with the 32-bit content of a destination register. The final result is written back to the destination register.

Description: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The lower 32-bit multiplication result is added with the content of Rt. And the final result is written 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:

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. The final result is written back to the destination register.

Description: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The lower 32-bit multiplication result is subtracted from the content of Rt. And the final result is written 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:

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: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The 64-bit multiplication result is written 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. 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:

```
Mresult = CONCAT(1`b0,Ra) * CONCAT(1`b0,Rb);
If (PSW.BE == 1) {
   R[Rt(4,1).0(0)](31,0) = Mresult(63,32);
   R[Rt(4,1).1(0)](31,0) = Mresult(31,0);
} else {
   R[Rt(4,1).1(0)](31,0) = Mresult(63,32);
   R[Rt(4,1).0(0)](31,0) = Mresult(31,0);
}
```

Exceptions: None

Privilege level: All

Note:

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: Multiply the 32-bit content of Ra with the 32-bit content of Rb. The 64-bit multiplication result is written 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. 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:

```
Mresult = Ra * Rb;
If (PSW.BE == 1) {
   R[Rt(4,1).0(0)](31,0) = Mresult(63,32);
   R[Rt(4,1).1(0)](31,0) = Mresult(31,0);
} else {
   R[Rt(4,1).1(0)](31,0) = Mresult(63,32);
   R[Rt(4,1).0(0)](31,0) = Mresult(31,0);
}
```

Exceptions: None

Privilege level: All

Note:

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: To store an 8-bit byte from a general register into a memory location.

Description: The least-significant 8-bit byte in the general register Rt is stored to the memory location. The memory address is specified by the implied GP register (R29) plus a sign-extended imm19s value. The imm19s value will cover 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:

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: To store an 8-bit byte from a general register into memory with the user mode privilege address translation.

Description: The least-significant 8-bit byte in the general register Rt is stored 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:

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: To store a 16-bit halfword from a general register into a memory location.

Description: The least-significant 16-bit halfword in the general register Rt is stored 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 will cover 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:

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	1 0
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: Store 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 registers list formed by Rb, Re, and the four-bit Enable4 field as follows.

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

- {b | a} option specifies the way how the first address is generated. {b} use the contents of Ra as the first memory store address. {a} use 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 this register range. 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 <Registers 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 result will happen to the contents of the memory range pointed to by the base register and the base register itself if the {m?} option is specified after this instruction.
- The register is 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 <Registers Specification>, the value stored to the 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:

(5) LMW and SMW instructions do not guarantee atomicity among individual memory access operations. And they do not guarantee single access to a memory location during the execution either. Any I/O access that has side-effects other than simple stable memory-like access behavior should not use these two instructions.

(6) 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 implemented by an implementation is:

- For LMW/SMW.i : increasing memory addresses from base address.
- For LMW/SMW.d: decreasing memory addresses from base address.

(7) 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 implemented by 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 or .
- 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.

(8) Based on the more likely access order of (2) and (3), upon interruption, the EVA register for un-aligned LMW/SMW will more likely 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".

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: To store a 32-bit word from a general register into a memory location.

Description: The 32-bit word in the general register Rt is stored 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 will cover 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:

Chapter 8
Instruction Latency for AndesCore Implementations

This chapter lists the instruction latency information for AndesCore families and contains the following sections

8.1 N12 Implementation

8.1.1 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 a 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 (latency – 1).

Producer/Consumer Instruction Group

- Producer/Consumer Instruction Group

Producer /Consumer Group	Instructions	Note
ALU	ADDI, SUBRI, ANDI, ORI, XORI, SLTI, SLTSI, MOVI, STEHI, 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, MADDS64, MADD64, MSUBS64, MSUB64, MADD32,	result in accumulator register D

	MSUB32	

- 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, most of a producer and corresponding consumer have a latency of 1. Only cases need special attention, or deviate from this general rule of thumb will be 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
5	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, LMW.i or LMW.d base is the	ALU, ST_D, BR	N+1
11		MUL, M2D, MEM_A, MISC_E1	N+2

	dependency reg, LMW.im or LWM.dm		
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
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

* Rule 5b exists when the configuration flag “NDS_POSTWRITE_E1_BYPASS” is turned off. Turning on configuration flag “NDS_POSTWRITE_E1_BYPASS” will make all LD_A consumers to 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
Instruction	Latency
RET (correct target prediction)	3
DSB	3
ISB, IRET	10
TLBOP Invalidate VA	10
TLBOP Invalidate All	98
DPREF/DPREFI (miss dcache)	4

8.1.2 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

8.2 N10 Implementation

8.2.1 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 a 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 (latency – 1).

Producer/Consumer Instruction Groups

- Producer/Consumer Instruction Group

Producer /Consumer Group	Instructions	Note
ALU	ADDI, SUBRI, ANDI, ORI, XORI, SLTI, SLTSI, MOVI, STEHI, 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, MADDS64,	result in accumulator

	MADD64, MSUBS64, MSUB64, MADD32, MSUB32	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
MFUSR_GR	MFUSR GR ← D	Result in general register R
SCW	SCW	Generate success/fail status in general register R
DIV	DIV, DIVS	
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_!bi	SWI, SHI, SBI, SW, SH, SB, SWUP, SCW	Store instructions which have the data register as the consumed state
ST_D_bi	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,	LD/ST instructions

	LB, LBS, LWUP, LLW, LMW, SWI, SHI, SBI, SW, SH, SB, SWUP, SCW, SMW, DPREF, DPREFI	(non-bi form) which have the base address registers (R1 or R2) as the consumed state
MEM_A_bi_R1	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 R1 as the consumed state
MEM_A_bi_R2	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 R2 as the consumed state
BR	JR, RET, JRAL, BEQ, BNE, BEQZ, BNEZ, BGEZ, BLTZ, BGTZ, BLEZ, BGEZAL, BLTZAL	
MISC	TLBOP TRD, TLBOP TWR, TLBOP RWR, TLBOP RWLK, TLBOP UNLK, TLBOP PB, TLBOP INV, CCTL, ISYNC, MTSR, MTUSR	Consumes general register R

In AndesCore N10, most of a producer and corresponding consumer have a latency of 1. Only cases need special attention, or deviate from this general rule of thumb will be described below.

No	Producer	Consumer	Latency	
			2R1W	3R2W
1	LD_D, MUL, MFUSR_GR,	MEM_A_!bi, MEM_A_bi_R1, ALU, MUL, BR, MISC	2	2

2	LMW_iHdL_nup*	MEM_A_bi_R2	1	2
3		ST_D_!bi	1	1
4		ST_D_bi	2	1
5	SCW	MEM_A_!bi, MEM_A_bi_R1, ALU, MUL, BR, MISC	3	3
6		MEM_A_bi_R2	2	3
7		ST_D_!bi	2	2
8		ST_D_bi	3	2

* Note that this dependency latency does not include the fixed self-stalling latency described in the next section for the LMW instructions.

8.2.2 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
DIV, DIVS	Variable (4 – 35) [i.e. 4+floor(log2(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" has 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

8.2.3 Cycle Penalty due to N10 Pipeline Control Mishap Recover

This section describes the AndesCore 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

8.2.4 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 missed cache. The following instructions will 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 the 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.

Chapter 9
AndesCore N12 implementation

This chapter describes CCTL and STANDBY instruction implementated in AndesCore N12 family and contains the following sections

9.1 CCTL Instruction

All CCTL subtype operations implemented by AndesCore N12 implementation are shown in the following table with a LIGHT GREEN background. And four defined optional operations are not implemented (shown in the table as a CLEAR background).

Table 61 N12 Implementation of CCTL instruction

SubType		bit 4-3			
bit 2-0		0	1	2	3
		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	-	-	-

9.2 STANDBY Instruction

In AndesCore N12 implementation, *software interrupt* in IVIC mode will not wakeup a core which has entered the STANDBY mode.

Chapter 10
AndesCore N1213 Hardcore Implementation Restriction

This chapter describes N1213 Hardcore (N1213_43U1H) implementation restriction and contains the following sections

10.1 Instruction Restriction

The following instructions are not implemented by N1213 hardcore (CPU_VER == 0x0C010003):, Part number N1213_43U1H

DIV/DIVS
STANDBY wait_done
MFUSR Rt, PC
MFUSR Rt, USR, Group 1 and Group 2
MTUSR Rt, USR, Group 1 and Group2

10.2 ISYNC Instruction Note

The correct instruction sequence for writing or updating any code data that will be executed afterwards for AndesCore N1213 hardcore (N1213_43U1H) 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
......
```

Please see the ISYNC instruction note for other implementations.

Proprietary Notice

Confidentiality status

This document is Preliminary. This document has no restriction on distribution.

Feedback on this Manual

If you have any problems with this Manual, Please contact Andes Technology Corporation by email at support@andestech.com
or on the Internet at www.andestech.com 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 also welcome.