ANY-1 Instruction Set

© 2021 Robert Finch

Table of Contents

[Instruction Formats 5](#_Toc71525562)

[Example Instruction 6](#_Toc71525563)

[Instructions 7](#_Toc71525564)

[Arithmetic / Logical 7](#_Toc71525565)

[ABS – Absolute Value 7](#_Toc71525566)

[ADD - Addition 8](#_Toc71525567)

[ADDIS – Add Immediate Shifted **Error! Bookmark not defined.**](#_Toc71525568)

[AND – Bitwise And 9](#_Toc71525569)

[ANDIS – Bitwise And Immediate Shifted **Error! Bookmark not defined.**](#_Toc71525570)

[AISIP – Add Immediate Shifted to IP 10](#_Toc71525571)

[BMM – Bit Matrix Multiply 11](#_Toc71525572)

[BYTNDX – Byte Index 13](#_Toc71525573)

[CNTLZ – Count Leading Zeros 14](#_Toc71525574)

[CNTPOP – Count Population 15](#_Toc71525575)

[CSRx – Control and Status Access 92](#_Toc71525576)

[DEP – Deposit 17](#_Toc71525577)

[DIV – Division 18](#_Toc71525578)

[DIVR – Division 20](#_Toc71525579)

[DIVU – Division Unsigned 20](#_Toc71525580)

[EXT –Extract Bitfield 22](#_Toc71525581)

[EXTU –Extract Bitfield Unsigned 23](#_Toc71525582)

[FDP – Fused Dot Product 23](#_Toc71525583)

[FFO –Find First One 24](#_Toc71525584)

[MAX – Maximum Value 25](#_Toc71525585)

[MIN – Minimum Value 26](#_Toc71525586)

[MUL – Signed Multiply 27](#_Toc71525587)

[MULF – Fast Unsigned Multiply 30](#_Toc71525588)

[MUX – Multiplex 31](#_Toc71525589)

[NEG - Negate 33](#_Toc71525590)

[NOT – Logical Not 34](#_Toc71525591)

[OR – Bitwise Or 35](#_Toc71525592)

[ORIS – Bitwise Or Immediate Shifted **Error! Bookmark not defined.**](#_Toc71525593)

[PERM – Permute Bytes 36](#_Toc71525594)

[SEQ – Set if Equal 37](#_Toc71525595)

[SGE – Set if Greater Than or Equal 39](#_Toc71525596)

[SGEU – Set if Greater Than or Equal Unsigned 40](#_Toc71525597)

[SGT – Set if Greater Than 41](#_Toc71525598)

[SGTU – Set if Greater Than Unsigned 42](#_Toc71525599)

[SIGN – Sign 42](#_Toc71525600)

[SLL –Shift Left Logical Pair 43](#_Toc71525601)

[SLT – Set if Less Than 45](#_Toc71525602)

[SLE – Set if Less Than or Equal 46](#_Toc71525603)

[SLEU – Set if Less Than or Equal 46](#_Toc71525604)

[SLTU – Set if Less Than Unsigned 46](#_Toc71525605)

[SNE – Set if Not Equal 46](#_Toc71525606)

[SRA –Shift Right Arithmetic Pair 48](#_Toc71525607)

[SRL –Shift Right Logical Pair 49](#_Toc71525608)

[SUB - Subtract 50](#_Toc71525609)

[SUBF – Subtract From 51](#_Toc71525610)

[U21NDX – UTF21 Index 52](#_Toc71525611)

[WYDNDX – Wyde Index 53](#_Toc71525612)

[XOR – Bitwise Exclusive Or 54](#_Toc71525613)

[ZXB –Zero Extend Byte 55](#_Toc71525614)

[ZXW –Zero Extend Wyde 55](#_Toc71525615)

[ZXT –Zero Extend Tetra 56](#_Toc71525616)

[Memory Operations 57](#_Toc71525617)

[LDx – Load 57](#_Toc71525618)

[LDB – Load Byte (8 bits) 60](#_Toc71525619)

[LDBZ – Load Byte, Zero Extend (8 bits) 60](#_Toc71525620)

[LDO – Load Octa (64 bits) 61](#_Toc71525621)

[LDT – Load Tetra (32 bits) 62](#_Toc71525622)

[LDTZ – Load Tetra, Zero Extend (32 bits) 62](#_Toc71525623)

[LDW – Load Wyde (16 bits) 63](#_Toc71525624)

[LDWZ – Load Wyde, Zero Extend (16 bits) 63](#_Toc71525625)

[LEA – Load Effective Address 64](#_Toc71525626)

[LSM – Load or Store Multiple 66](#_Toc71525627)

[STx – Store 67](#_Toc71525628)

[STB – Store Byte (8 bits) 70](#_Toc71525629)

[STBZ – Store Byte and Zero (8 bits) 70](#_Toc71525630)

[STO – Store Octa (64 bits) 71](#_Toc71525631)

[STOZ – Store Octa and Zero (64 bits) 71](#_Toc71525632)

[STT – Store Tetra (32 bits) 73](#_Toc71525633)

[STTZ – Store Tetra and Zero (32 bits) 73](#_Toc71525634)

[STW – Store Wyde (16 bits) 73](#_Toc71525635)

[STWZ – Store Wyde and Zero (16 bits) 73](#_Toc71525636)

[Flow Control (Branch Unit) Operations 75](#_Toc71525637)

[BEQ – Branch if Equal 75](#_Toc71525638)

[BGE – Branch if Greater Than or Equal 77](#_Toc71525639)

[BGEU – Branch if Greater Than or Equal Unsigned 78](#_Toc71525640)

[BGT – Branch if Greater Than 79](#_Toc71525641)

[BGTU – Branch if Greater Than Unsigned 80](#_Toc71525642)

[BNE – Branch if Not Equal 81](#_Toc71525643)

[BLE – Branch if Less Than or Equal 82](#_Toc71525644)

[BLEU – Branch if Less Than or Equal Unsigned 82](#_Toc71525645)

[BLT – Branch if Less Than 83](#_Toc71525646)

[BLTU – Branch if Less Than Unsigned 83](#_Toc71525647)

[BRA – Unconditional Branch 85](#_Toc71525648)

[BRK – Break 92](#_Toc71525649)

[CHK – Check Register Against Bounds 86](#_Toc71525650)

[JAL – Jump and Link 88](#_Toc71525651)

[JMP – Jump 90](#_Toc71525652)

[PFI – Poll for Interrupt 90](#_Toc71525653)

[RET – Return from Subroutine 91](#_Toc71525654)

[REX – Redirect Exception 95](#_Toc71525655)

[SYNC -Synchronize 95](#_Toc71525656)

[Floating Point Instructions 92](#_Toc71525657)

[Vector Specific Instructions 102](#_Toc71525658)

[Arithmetic / Logical 102](#_Toc71525659)

[V2BITS 102](#_Toc71525660)

[VACC - Accumulate 103](#_Toc71525661)

[VBITS2V 104](#_Toc71525662)

[VCIDX – Compress Index 105](#_Toc71525663)

[VCMPRSS – Compress Vector 106](#_Toc71525664)

[VEINS / VMOVSV – Vector Element Insert 107](#_Toc71525665)

[VEX / VMOVS – Vector Element Extract 108](#_Toc71525666)

[VSCAN 109](#_Toc71525667)

[VSHLV – Shift Vector Left 110](#_Toc71525668)

[VSHRV – Shift Vector Right 111](#_Toc71525669)

[Memory Operations 112](#_Toc71525670)

[CVLDx – Compressed Vector Load 112](#_Toc71525671)

[CVSTx – Compressed Vector Store 114](#_Toc71525672)

[Root Opcode Map 116](#_Toc71525673)

[{SR3} Triadic Register Ops 117](#_Toc71525674)

[{SR2} Dyadic Register Ops 117](#_Toc71525675)

[{SR1} Monadic Register Ops 117](#_Toc71525676)

# Instruction Formats

Immediate Format:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 09h8 |

Extended Immediate

|  |  |
| --- | --- |
| 31 8 | 7 0 |
| Constant31..8 | 508 |

|  |  |
| --- | --- |
| 31 8 | 7 0 |
| Constant55..32 | 518 |

|  |  |
| --- | --- |
| 31 8 | 7 0 |
| Constant79..56 | 528 |

Register Format:

SR1 (one source register)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 0Ch6 | Func6 | Ra6 | Rt6 | 01h8 |

SR2 (two source register)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Func6 | Rb6 | Ra6 | Rt6 | 02h8 |

F8 Instruction Modifier

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | 58h8 |

F9 Bitfield Modifier

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | 59h8 |

Branch

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Constant6 | Rb6 | Ra6 | Const6 | 48h8 |

FA Branch Modifier

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 29 | 28 20 | 19 14 | 13 8 | 7 0 |
| DT3 | Constant9 | Rc6 | Rt6 | 5Ah8 |

Instruction Modifier

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 14 | 1312 | 11 9 | 8 | 7 0 |
| C3 | Rm3 | Constant12 | A | m3 | z | 5Bh8 |

z: 1 = zero vector element if mask bit clear, 0 = vector element unchanged (ignored for scalar ops)

m3: vector mask register (ignored for scalar operations).

Rm3: rounding mode

If any of Rt, Ra, Rb, Rc are vector registers, then the instruction is a vector instruction.

|  |  |
| --- | --- |
| Rn8 |  |
| 0 to 63 | scalar registers |
| 64 to 127 | vector registers |
| 128 to 255 | Rn is a seven-bit constant |

|  |  |  |
| --- | --- | --- |
| U2 | Execution Unit | Qualifier |
| 0 | Integer | .int |
| 1 | Floating-point | .fp |
| 2 | Decimal floating-point | .dfp |
| 3 | Posit | .pos |

|  |  |  |  |
| --- | --- | --- | --- |
| Sz4 | Size | Qualifier | Alt Qualifier |
| 0 | byte | .b |  |
| 1 | wyde | .w |  |
| 2 | tetra | .t | .s (single) |
| 3 | octa | .o | .d (double) |
| 4 | hexi | .h | .q (quad) |
| 8 | SIMD byte | .bp |  |
| 9 | SIMD wyde | .wp |  |
| 10 | SIMD tetra | .tp | .sp |
| 11 | SIMD octa | .op | .dp |
| 12 | SIMD hexi | .hp | .qp |

## Example Instruction

add.int.o x1,x2,x3,x0 ; scalar add of integers x2,x3

add.int.o v1,v2,v3,v0 ; vector add of integers v2,v3

add.int.o v1,v2,v0,x4 ; vector add scalar integers v2,x4

add.fp.o v1,v2,v3,v0 ; vector add float-point double v2,v3

# Instructions

## Arithmetic / Logical

### ABS – Absolute Value

**Description:**

This instruction takes the absolute value of a register and places the result in a target register.

**Instruction Format: R1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 66 | ~6 | Ra6 | Rt6 | 01h8 |

**Operation:**

If Ra < 0

Rt = -Ra

else

Rt = Ra

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Rt[x] = Ra[x] < 0 ? -Ra[x] : Ra[x]

**Execution Units:** I, F, D, P

**Exceptions:** none

**Notes:**

For sign-magnitude formats this instruction simply clears the MSB of the number. No rounding occurs.

### ADD - Addition

**Description:**

Add two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction.

**Operation:**

Rt = Ra + Imm

or

Rt = Ra + Rb

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] + Vb[x]

else if (z) Vt[x] = 0

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 04h8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 46 | Rb6 | Ra6 | Rt6 | 02h8 |

**Vector Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 84h8 |

**Vector Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 46 | Rb6 | Ra6 | Rt6 | 82h8 |

**Exceptions:** none

### AND – Bitwise And

**Description**:

Perform a bitwise ‘and’ operation between operands. The first operand must be in a register. The second operand may be in a register of may be an immediate value specified in the instruction. A third source operand must be in a register. The immediate constant is one extended before use.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 08h8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 86 | Rb6 | Ra6 | Rt6 | 03h8 |

**Operation:**

Rt = Ra & Imm

or

Rt = Ra & Rb & Rc

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] & Vb[x] & Vc[x]

else if (z) Vt[x] = 0

**Exceptions**: none

### AISIP – Add Immediate Shifted to IP

**Description**:

This instruction forms the sum of the instruction pointer and an immediate value shifted left a multiple of 32 times. The result is then placed in the target register. The low order 32 bits of the target register are zeroed out.

**Instruction Format**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 63 32 | 3128 | 2724 | 23 16 | 15 8 | 7 0 |
| Constant32 | F4 | Sh4 | 638 | Rt8 | Opcode8 |

**Exceptions**: none

### BLEND – Blend Colors

**Description**:

This instruction blends two colors whose values are in Ra and Rb according to an alpha value in Rc. The resulting color is placed in register Rt. The alpha value is an eight-bit value assumed to be a binary fraction less than one. The color values in Ra and Rb are assumed to be RGB888 format colors. The result is a RGB888 format color. The high order eight bits of the result register are set to the high order eight bits of Ra. Note that a close approximation to 1.0 – alpha is used. Each component of the color is blended.

**Instruction Format**: R3

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | 03 | 06 | Rc6 | 0 | m3 | z | F8h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 30h6 | Rb6 | Ra6 | Rt6 | 03h8 |

**Operation**:

Rt.R = (Ra.R \* alpha) + (Rb.R \* ~alpha)

Rt.G = (Ra.G \* alpha) + (Rb.G \* ~alpha)

Rt.B = (Ra.B \* alpha) + (Rb.B \* ~alpha)

**Clock Cycles**: 2

### BMM – Bit Matrix Multiply

BMM Rt, Ra, Rb

**Description**:

The BMM instruction treats the bits of register Ra and register Rb as an 8x8 matrix and performs a bit matrix multiply of the two registers and stores the result in the target register. An alternate mnemonic for this instruction is MOR.

**Instruction Format**: S2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 61 | 60 58 | 57 50 | 4948 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Fn3 | Rm3 | 03h8 | U2 | Sz4 | m3 | z | ~8 | Rb8 | Ra8 | Rt8 | 03h8 |

|  |  |
| --- | --- |
| Fn3 | Function |
| 0 | MOR |
| 1 | MXOR |
| 2 | MORT (MOR transpose) |
| 3 | MXORT (MXOR transpose) |
| 4 to 7 | reserved |

**Operation**:

for I = 0 to 7

for j = 0 to 7

Rt.bit[i][j] = (Ra[i][0]&Rb[0][j]) | (Ra[i][1]&Rb[1][j]) | … | (Ra[i][15]&Rb[15][j])

**Clock Cycles:** 1

**Execution Units: Integer** ALU

**Exceptions**: none

**Notes**:

The bits are numbered with bit 63 of a register representing I,j = 0,0 and bit 0 of the register representing I,j = 7,7.

### BYTNDX – Byte Index

**Description:**

This instruction searches Ra, which is treated as an array of eight bytes, for a byte value specified by Rb or an immediate value and places the index of the byte into the target register Rt. If the byte is not found -1 is placed in the target register. A common use would be to search for a null byte. The index result may vary from -1 to +7. The index of the first found byte is returned (closest to zero).

The result tag type is set to integer.

**Instruction Format: S**R2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 06 | Rb6 | Ra6 | Rt6 | 1Ah8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 28 | 27 20 | 19 14 | 13 8 | 7 0 |
| 1 | Imm8 | Ra6 | Rt6 | 1Ah8 |

**R2 Supported Formats**: .o

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rt = Index of (Rb in Ra)

**Exceptions:** none

### CMP – Compare

**Description**

Compare two registers or a register and an immediate value and return the relationship between them. Both values are treated as signed numbers.

**Instruction Format: R1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 2Ah6 | Rb6 | Ra6 | Rt6 | 02h8 |

**Operation:**

Rt = Ra < Rb ? –1 : Ra = Rb ? 0 : 1

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] < Vb[x] ? –1 : Va[x]=Vb[x] ? 0 : 1

### CNTPOP – Count Population

**Description:**

Count the number of ones and place the count in the target register.

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = popcnt(Va[x])

**Instruction Format: R1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 2h6 | ~6 | Ra6 | Rt6 | 01h8 |

**Execution Units: integer** ALU

**Exceptions:** none

### CNTLZ – Count Leading Zeros

**Description**:

Count the number of leading zeros (starting at the MSB) in Ra and place the count in the target register.

**Instruction Format**: R1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 0h6 | ~6 | Ra6 | Rt6 | 01h8 |

**R1 Supported Formats**: .o

**Clock Cycles**: 1

**Execution Units:** Integer ALU

**Exceptions:** none

### COM – Ones Complement

**Description:**

Complement all the bits in the register. 1’s become 0’s and 0’s become 1’s.

**Instruction Format: RI**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 36 | ~6 | Ra6 | Rt6 | 01h8 |

**Operation**

**Rt = ~Ra**

**OR**

**Rt = ~Ra**

**Vector Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = ~Va[x]

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

**Exceptions**: none

### DEP – Deposit

**Description**:

Insert to a bitfield. Rc specifies the bitfield offset, Rd specifies the width of the bitfield. Rb specifies the data to insert. Ra contains the original source data. The least significant Rd minus one bits of Rb are inserted into Ra at the position specified by Rc. The final result is placed into Rt.

This instruction may also be used to perform a left shift of a single register by specifying x0 for Ra.

**Formats Supported**: R4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rc6 | Rd6 | A | m3 | z | F9h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 36 | Rb6 | Ra6 | Rt6 | 1Ch8 |

|  |  |
| --- | --- |
| DT3 | Meaning |
| 00 | Rc,Rd are both regs |
| 01 | Rc is a six bit immediate, Rd is a reg |
| 10 | Rd is a six bit immediate, Rc is a reg |
| 11 | Both Rc, Rd are six bit immediates |

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### DIF – Difference

**Description:**

This instruction computes the difference between two signed values in registers Ra and Rb and places the result in a target Rt register. The difference is calculated as the absolute value of Ra minus Rb.

**Instruction Format:** R2, R2S

**Supported Formats**: .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 1

**Execution Units:** Integer

**Operation:**

Rt = Abs(Ra - Rb)

**Exceptions**: none

### DIV – Division

**Description**:

Divide two operand values and place the result in the target register. The first operand must be in a register specified by the Ra field of the instruction. The second operand may be a register specified by the Rb field of the instruction or an immediate value. Both operands are treated as signed values.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 10h8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 10h6 | Rb6 | Ra6 | Rt6 | 02h8 |

**Execution Units**: ALU

**Clock Cycles**: 67

**Exceptions**: none

### DIVR – Division

**Description**:

This instruction is supplied as division is not commutative. Divide two operand values and place the result in the target register. The first operand must be an immediate value. The second operand must be a register specified by the Ra field of the instruction. Both operands are treated as signed values. This instruction allows a constant to be divided by a register value “reverse” to how the DIV instruction works.

**Formats Supported**: RI

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 63 32 | 3130 | 2928 | 27 24 | 23 16 | 15 8 | 7 0 |
| Constant32 | T2 | U2 | Sz4 | Ra8 | Rt8 | 21h8 |

**Execution Units**: ALU

**Clock Cycles**: 67

**Exceptions**: none

### DIVU – Division Unsigned

**Description**:

Divide two operand values and place the result in the target register. The first operand must be in a register specified by the Ra field of the instruction. The second operand may be either a register specified by the Rb field of the instruction, an immediate value. Both operands are treated as unsigned values.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 11h8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 11h6 | Rb6 | Ra6 | Rt6 | 02h8 |

**Execution Units**: ALU

**Clock Cycles**: 67

**Exceptions**: none

### EXI0,EXI1,EXI2 – Extended Immediate

**Description:**

These instructions are used to extend the constant field of the following instruction. The constant is extended from bit eight. Multiple constant extensions may be present to extend a constant up to 64 bits. When multiple extensions are present they should be placed in order least significant to most significant. (EXI0 first, EXI1 second, EXI2 third). The constant extensions sign extend to the width of the machine.

Constant extensions may be applied for most instructions with a constant field.

Interrupts are locked out between the modifier and the following instruction.

**Instruction Format: EXI**

|  |  |
| --- | --- |
| 31 8 | 7 0 |
| Constant31..8 | 508 |

|  |  |
| --- | --- |
| 31 8 | 7 0 |
| Constant55..32 | 518 |

|  |  |
| --- | --- |
| 31 8 | 7 0 |
| Constant79..56 | 528 |

### EXT –Extract Bitfield

**Description**:

A bitfield is extracted from the source by shifting the source to the right and ‘and’ masking. The result is sign extended to the width of the machine. This instruction may be used to sign extend a value from an arbitrary bit position. The width specified should be one less than the desired width. The source is value is contained in the register pair Ra, Rb. The field width is specified by Rc and field offset by Rd.

**Instruction Format**: R4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | F9h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 46 | Rb6 | Ra6 | Rt6 | 1Ch8 |

|  |  |
| --- | --- |
| DT3 | Meaning |
| 00 | Rc,Rd are both regs |
| 01 | Rc is a six bit immediate, Rd is a reg |
| 10 | Rd is a six bit immediate, Rc is a reg |
| 11 | Both Rc, Rd are six bit immediates |

**Execution Units:** Integer ALU

**Exceptions**: none

**Notes**:

### EXTU –Extract Bitfield Unsigned

**Description**:

A bitfield is extracted from the source by shifting the source to the right and ‘and’ masking. The result is zero extended to the width of the machine. This instruction may be used to zero extend a value from an arbitrary bit position. The width specified should be one less than the desired width. The source is a 128-bit value which is the concatenation of Rb and Ra. Rc contains the field offset, Rd the width.

**Instruction Format**: R4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | F9h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 56 | Rb6 | Ra6 | Rt6 | 1Ch8 |

|  |  |
| --- | --- |
| DT3 | Meaning |
| 00 | Rc,Rd are both regs |
| 01 | Rc is a six bit immediate, Rd is a reg |
| 10 | Rd is a six bit immediate, Rc is a reg |
| 11 | Both Rc, Rd are six bit immediates |

**Execution Units:** Integer ALU

**Exceptions**: none

**Notes**:

### FDP – Fused Dot Product

**Description**:

Calculate the dot product x = (a \* b) + (c \* d). The operations are fused together meaning no rounding occurs until the final product is produced.

**Instruction Format**: R4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | F8h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Func6 | Rb6 | Ra6 | Rt6 | 37h8 |

### FFO –Find First One

**Description**:

A bitfield contained in Ra is searched beginning at the most significant bit to the least significant bit for a bit that is set. The index into the bitfield of the bit that is set is stored in Rt. If no bits are set, then Rt is set equal to -1. The field offset is specified by Rc, the field width by Rd.

**Instruction Format**: R4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | F9h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 66 | Rb6 | Ra6 | Rt6 | 1Ch8 |

|  |  |
| --- | --- |
| DT3 | Meaning |
| 00 | Rc,Rd are both regs |
| 01 | Rc is a six bit immediate, Rd is a reg |
| 10 | Rd is a six bit immediate, Rc is a reg |
| 11 | Both Rc, Rd are six bit immediates |

**Clock Cycles**:

**Execution Units:** Integer

**Exceptions**: none

### MAX – Maximum Value

**Description:**

Determines the maximum of two values in registers Ra, Rb and places the result in the target register Rt.

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 29h6 | Rb6 | Ra6 | Rt6 | 02h8 |

**Operation:**

IF Ra > Rb

Rt = Ra

else

Rt = Rb

### MIN – Minimum Value

**Description:**

Determines the minimum of two values in registers Ra, Rb and places the result in the target register Rt.

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 28h6 | Rb6 | Ra6 | Rt6 | 02h8 |

**Operation:**

IF Ra < Rb

Rt = Ra

else

Rt = Rb

### MOD – Instruction Modifier

**Description:**

Used to modify the operation of the following instruction.

Interrupts are locked out between the modifier and the following instruction.

**Instruction Format: F8**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | 58h8 |

A[0]: 1 = apply vector mask, 0=ignore mask spec

A[1]: 1 = apply rounding mode. 0 = ignored rounding mode spec

There are four basic additional elements supplied for the following instruction.

1. A vector mask specification, used only by vector instructions.
2. Two additional source registers
3. A rounding mode specification, useful only to applicable instructions
4. A data type to help identify the operation required.

Two additional register fields allow up to four source operands for the following instruction. If these registers are not required they should be specified as x0.

Application of the vector mask and rounding mode are optional. Two bits in the ‘A’ field indicate which of these modifiers is applied.

|  |  |
| --- | --- |
| DT3 | Interpretation: Instruction is operating on: |
| 0 | Integer |
| 1 | Floating point (double precision) |
| 2 | Posit (64-bit) |
| 3-7 | reserved |

**Instruction Format: F9**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | 59h8 |

A[0]: 1 = apply vector mask, 0=ignore mask spec

A[1]: 1 = apply rounding mode. 0 = ignored rounding mode spec

The F9 modifier is almost the same as the F8 modifier, except that the data type DT field is interpreted differently. The data type field uses two bits to indicate whether Rc and Rd are register specs or six-bit unsigned integer values. This modifier is used primarily for the bitfield extract / deposit instructions although it is also applicable to the SLLP instruction.

**Instruction Format: FA**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 29 | 28 20 | 19 14 | 13 8 | 7 0 |
| DT3 | Constant9 | Rc6 | Rt6 | 5Ah8 |

The FA modifier applies to branch instructions to both extend the range of a branch and allow branch-to-register, and branch-and-link capability. When the FA modifier is present, the Rc register overrides the use of the IP in calculating the branch target address. The target address is then the sum of register Rc and a constant supplied in the instruction.

The constant field of the FA modifier adds an additional nine bits to the branch displacement. This allows branching extended to ±4MB.

The Rt field may be set to the address of the instruction following the branch, to allow conditional branch to subroutine capability.

Finally, the data type field identifies the type of data being compared by the branch instruction.

### MUL – Signed Multiply

**Description**:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as signed values, the result is a signed result.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 06h8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 66 | Rb6 | Ra6 | Rt6 | 02h8 |

**Execution Units**: ALU

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] \* Vb[x]

**Exceptions**: none

### MULF – Fast Unsigned Multiply

**Description**:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as unsigned values. The result is an unsigned result. The fast multiply multiplies only the low order 24 bits of the first operand times the low order 16 bits of the second. The result is a 40-bit unsigned product.

**Instruction Format**: R2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 1Ch6 | Rb6 | Ra6 | Rt6 | 02h8 |

**Instruction Format:** RI

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 15h8 |

**Execution Units**: ALU

**Clock Cycles:** 1

**Exceptions**: none

### MULU – Unsigned Multiply

**Description**:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as unsigned values, the result is a unsigned result.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 0Eh8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Eh6 | Rb6 | Ra6 | Rt6 | 02h8 |

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] \* Vb[x]

**Exceptions**: none

### MUX – Multiplex

**Description**:

The MUX instruction performs a bit-by-bit copy of a bit of Rb to the target register if the corresponding bit in Ra is set, or a copy of a bit from Rc if the corresponding bit in Ra is clear.

**Instruction Format**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| ~3 | ~3 | ~6 | Rc6 | A | m3 | z | F8h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 04h6 | Rb6 | Ra6 | Rt6 | 03h8 |

**Exceptions**: none

**Execution Units: integer** ALU

### NEG - Negate

**Description:**

This is an alternate mnemonic for the SUBF instruction where the constant is zero.

**Instruction Format**: R2

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| 012 | Ra6 | Rt6 | 05h8 |

**Scalar Operation**

Rt = 0 - Rb

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = 0 - Vb[x]

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

**Notes**

For sign-magnitude operations the sign bit is inverted, no subtract occurs. The result is not rounded.

### NOT – Logical Not

**Description:**

This instruction takes the logical ‘not’ value of a register and places the result in a target register. If the source register contains a non-zero value, then a zero is loaded into the target. Otherwise, if the source register contains a zero a one is loaded into the target register.

NOT reduces the value to a single bit Boolean.

**Instruction Format**: R1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 46 | ~6 | Ra6 | Rt6 | 01h8 |

**Operation:**

Rt = !Ra

**Exceptions**: none

### OR – Bitwise Or

**Description**:

Perform a bitwise or operation between operands. The immediate constant is zero extended before use.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 09h8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 96 | Rb6 | Ra6 | Rt6 | 03h8 |

**Operation**

**Rt = Ra | Immediate**

**OR**

**Rt = Ra | Rb**

**Vector Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] | Vb[x] | Vc[x]

**Exceptions**: none

### PERM – Permute Bytes

**Description**:

This instruction allows any combination of bytes in a source register to be copied to a target register. The low order twenty-four bits of register Rb or a 12-bit immediate constant are used to identify which source bytes are copied to the destination. The twenty-four-bit value is composed of eight three-bit fields. Field S0 indicates the source byte for target byte position 0. S1 indicates the source byte for target byte position 1. S2 to S7 work similarly for the remaining target bytes. There are many interesting possibilities with this instruction. A single source byte could be copied to all target byte positions for instance. Or the order of bytes in a word could be reversed.

**Instruction Format: S**R2, PERM

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 61 | 60 58 | 57 50 | 4948 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| 03 | Rm3 | ~8 | 02 | Sz4 | m3 | z | ~8 | Rb8 | Ra8 | Rt8 | 17h8 |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 61 | 60 58 | 57 50 | 4948 | 47 44 | 4341 | 40 | 39 24 | 23 16 | 15 8 | 7 0 |
| 13 | Rm3 | Imm23..16 | 02 | Sz4 | m3 | z | Imm15..0 | Ra8 | Rt8 | 17h8 |

**Execution Units**: integer ALU

**Clock Cycles**: 1

**Exceptions**: none

### PTRDIF – Difference Between Pointers

**Description**:

Subtract two values then shift the result right. Both operands must be in a register. The right shift is provided to accommodate common object sizes. It may still be necessary to perform a divide operation after the PTRDIF to obtain an index into odd sized or large objects. Rc may vary from zero to thirty-one.

The result tag is forced to integer.

**Instruction Format**: R3

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 59 | 5856 | 55 48 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| ~ | Rm3 | 30h8 | Sz4 | m3 | z | Rc8 | Rb8 | Ra8 | Rt8 | 03h8 |

**Operation**:

Rt = Abs(Ra – Rb) >> Rc

**Clock Cycles**: 1

**Execution Units: Integer**

**Exceptions**:

None

### SEQ – Set if Equal

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is equal to a second operand in register (Rb) or an immediate constant then the target register is set to a one, otherwise the target register is set to a zero.

For floating-point operations positive and negative zero are considered equal.

If a vector operation is taking place then the target register is one of the vector mask registers.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 26h8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 26h6 | Rb6 | Ra6 | Rt6 | 02h8 |

### SGE – Set if Greater Than or Equal

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than or equal to a second operand in register (Rb) then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as signed values.

There is no immediate form to this instruction. An immediate equivalent may be achieved using the SGT instruction and adjusting the constant by one.

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 2Dh6 | Rb6 | Ra6 | Rt6 | 02h8 |

### SGEU – Set if Greater Than or Equal Unsigned

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than or equal to a second operand in register (Rb) then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as signed values.

There is no immediate form to this instruction. An immediate equivalent may be achieved using the SGTU instruction and adjusting the constant by one.

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 2Fh6 | Rb6 | Ra6 | Rt6 | 02h8 |

### SGT – Set if Greater Than

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than a second operand which is a constant supplied in the instruction, then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as signed values.

There is no register form of this instruction. The register equivalent operation may be performed using the SLT instruction and swapping the registers.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 29h8 |

### SGTU – Set if Greater Than Unsigned

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than a second operand which is a constant supplied in the instruction, then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as signed values.

There is no register form of this instruction. The register equivalent operation may be performed using the SLTU instruction and swapping the registers.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 2Bh8 |

### SIGN – Sign

**Synopsis**

Take sign of value. This is an extended Mnemonic for the [CMP](#_CMP_–_Compare) instruction.

**Description**

The sign of a register is placed in the target register Rt.

**Instruction Format: R1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 2Ah6 | 06 | Ra6 | Rt6 | 02h8 |

**Operation:**

Rt = Ra < 0 ? –1 : Ra = 0 ? 0 : 1

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] < 0 ? –1 : Va[x]=0 ? 0 : 1

### SLL –Shift Left Logical

**Description**:

Left shift an operand value by an operand value and place the result in the target register. Zeros are shifted into the least significant bits. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rb field of the instruction, or an immediate value.

**Instruction Formats**: R2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 19h6 | Rb6 | Ra6 | Rt6 | 02h8 |

**Instruction Formats**: R2I

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 1Ah6 | Imm6 | Ra6 | Rt6 | 02h8 |

**Operation Size:** .o, .t, .w, .b

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### SLLP –Shift Left Logical Pair

**Description**:

Left shift a pair of operand values by an operand value and place the result in the target register. The upper 64 bits of the result are placed in the target register. Zeros are shifted into the least significant bits. The operand pair must be in registers specified by the Ra and Rc field of the instruction. The third operand may be either a register specified by the Rb field of the instruction, or an immediate value.

This instruction may also be used to perform a left rotate of a single register by specifying the same register for Ra and Rc.

**Instruction Formats**: R3

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | F9h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 10h6 | Rb6 | Ra6 | Rt6 | 03h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 11h6 | Imm6 | Ra6 | Rt6 | 03h8 |

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### SLT – Set if Less Than

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than a second operand in either a register (Rb) or a constant supplied in the instruction, then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as signed values.

The register form of the instruction may also be used to test for greater than by swapping the operands around.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 59 | 58 56 | 55 48 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| ~ | Rm3 | 0Ch8 | Sz4 | m3 | z | 2Ch8 | Rb8 | Ra8 | Rt8 | 03h8 |

### SLE – Set if Less Than or Equal

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than or equal to a second operand in register (Rb) then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as signed values.

There is no immediate form to this instruction. An immediate equivalent may be achieved using the SLT instruction and adjusting the constant by one.

### SLEU – Set if Less Than or Equal

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than or equal to a second operand in register (Rb) then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as unsigned values.

There is no immediate form to this instruction. An immediate equivalent may be achieved using the SLTU instruction and adjusting the constant by one.

### SLTU – Set if Less Than Unsigned

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than a second operand in either a register (Rb) or a constant supplied in the instruction, then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as unsigned values.

The register form of the instruction may also be used to test for greater than by swapping the operands around.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 59 | 58 56 | 55 48 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| ~ | Rm3 | 0Ch8 | Sz4 | m3 | z | 2Eh8 | Rb8 | Ra8 | Rt8 | 03h8 |

### SNE – Set if Not Equal

**Description:**

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is not equal to a second operand in register (Rb) or an immediate constant then the target register is set to a one, otherwise the target register is set to a zero.

For floating-point operations positive and negative zero are considered equal.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 27h8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 27h6 | Rb6 | Ra6 | Rt6 | 02h8 |

### SRA –Shift Right Arithmetic Pair

**Description**:

This is an alternate mnemonic for the signed field extract [EXT](#_EXT_–Extract_Bitfield) instruction.

Right shift a pair of operand values by an operand value and place the result in the target register. The lower 64 bits of the result are placed in the target register. The sign bit is shifted into the most significant bits. The operand pair must be in registers specified by the Ra and Rb field of the instruction. The third operand may be either a register specified by the Rc field of the instruction, or an immediate value.

**Instruction Format**: R4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | 636 | Rc6 | A | m3 | z | F9h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 46 | Rb6 | Ra6 | Rt6 | 1Ch8 |

|  |  |
| --- | --- |
| DT3 | Meaning |
| 10 | Rd is a six bit immediate, Rc is a reg |
| 11 | Both Rc, Rd are six bit immediates |

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### SRL –Shift Right Logical Pair

**Description**:

This is an alternate mnemonic for the unsigned field extract [EXTU](#_EXTU_–Extract_Bitfield) instruction.

Right shift a pair of operand values by an operand value and place the result in the target register. The lower 64 bits of the result are placed in the target register. Zeros are shifted into the most significant bits. The operand pair must be in registers specified by the Ra and Rb field of the instruction. The third operand may be either a register specified by the Rc field of the instruction, or an immediate value.

This instruction may also be used to perform a right rotate of a single register by specifying the same register for Ra and Rb.

**Instruction Format**: R4

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | 636 | Rc6 | A | m3 | z | F9h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 56 | Rb6 | Ra6 | Rt6 | 1Ch8 |

|  |  |
| --- | --- |
| DT3 | Meaning |
| 10 | Rd is a six bit immediate, Rc is a reg |
| 11 | Both Rc, Rd are six bit immediates |

**Instruction Format**: SR4

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 59 | 58 56 | 55 48 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| ~ | Rm3 | BFh8 | 54 | m3 | z | Rc8 | Rb8 | Ra8 | Rt8 | 2Ch8 |

**Operation Size:** .w

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### SUB - Subtract

**Description:**

Subtract two values. Both operands must be in a register.

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 56 | Rb6 | Ra6 | Rt6 | 02h8 |

**Scalar Operation**

Rt = Ra - Rb

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] - Vb[x]

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

### SUBF – Subtract From

**Description:**

Subtract two values. The first operand must be in a register. The second operand must be an immediate value specified in the instruction. There is no register form for this instruction.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 05h8 |

**Operation:**

Rt = Imm - Ra

**Exceptions:** none

### U21NDX – UTF21 Index

**Description:**

This instruction searches Ra, which is treated as an array of three UTF21 values, for a value specified by Rb and places the index of the value into the target register Rt. If the UTF21 value is not found -1 is placed in the target register. A common use would be to search for a null. The index result may vary from -1 to +2. The index of the first found value is returned (closest to zero).

The result is tagged as an integer.

**Instruction Format:** R2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 06 | Rb6 | Ra6 | Rt6 | 23h8 |

**Supported Formats**: .o

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rt = Index of (Rb in Ra)

**Exceptions:** none

### WYDNDX – Wyde Index

**Description:**

This instruction searches Ra, which is treated as an array of four wydes, for a wyde value specified by Rb and places the index of the wyde into the target register Rt. If the wyde is not found -1 is placed in the target register. A common use would be to search for a null wyde. The index result may vary from -1 to +3. The index of the first found wyde is returned (closest to zero).

**Instruction Format:** R2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 06 | Rb6 | Ra6 | Rt6 | 1Bh8 |

**R2 Supported Formats**: .o

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rt = Index of (Rb in Ra)

**Exceptions:** none

### XOR – Bitwise Exclusive Or

**Description:**

Perform a bitwise exclusive or operation between operands. The first operand must be in a register. The second operand may be a register or immediate value. A third operand must be in a register. The immediate constant is zero extended before use.

**Instruction Format: RI**

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 0Ah8 |

**Instruction Format: R2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Ah6 | Rb6 | Ra6 | Rt6 | 03h8 |

**Operation**

**Rt = Ra ^ Immediate**

**OR**

**Rt = Ra ^ Rb**

**Vector Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] ^ Vb[x] ^ Vc[x]

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

**Exceptions**: none

### ZXB –Zero Extend Byte

**Description**:

This is an alternate mnemonic for the bitfield extract (EXTU) operation.

**Instruction Format**: EXT

A bitfield in the source specified by Ra is extracted, the result is copied to the target register. Rc specifies the bit offset. Rd specifies the bit width.

**Clock Cycles**: 1

**Execution Units:** Integer ALU

**Exceptions**: none

**Notes**:

### ZXW –Zero Extend Wyde

**Description**:

This is an alternate mnemonic for the bitfield extract (EXTU) operation.

**Instruction Format**: BFI

A bitfield in the source specified by Ra is extracted, the result is copied to the target register. Rc specifies the bit offset. Rd specifies the bit width.

**Clock Cycles**: 1

**Execution Units:** Integer ALU

**Exceptions**: none

**Notes**:

### ZXT –Zero Extend Tetra

**Description**:

This is an alternate mnemonic for the bitfield extract (EXTU) operation.

**Instruction Format**: EXT

A bitfield in the source specified by Ra is extracted, the result is copied to the target register. Rc specifies the bit offset. Rd specifies the bit width.

**Clock Cycles**: 1

**Execution Units:** Integer ALU

**Exceptions**: none

**Notes**:

## Memory Operations

### CACHE – Cache Command

CACHE Cmd, d[Rn]

**Description:**

This instruction commands the cache controller to perform an operation. Commands are summarized in the command table below. Commands may be issued to both the instruction and data cache at the same time. The address of the cache line to be invalidated is passed in Ra if needed.

**Instruction Formats**: CACHE

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 27 20 | 19 14 | 13 11 | 10 8 | 7 0 |
| 153..0 | Const8 | Ra6 | DC3 | IC3 | 60h8 |

**Commands:**

|  |  |  |
| --- | --- | --- |
| IC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 1 | invline | invalidate line associated with given address |
| 2 | invall | invalidate the entire cache (address is ignored) |
| 3 to 7 |  | reserved |

|  |  |  |
| --- | --- | --- |
| DC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 1 | enable | enable cache (instruction cache is always enabled) |
| 2 | disable | not valid for the instruction cache |
| 3 | invline | invalidate line associated with given address |
| 4 | invall | invalidate the entire cache (address is ignored) |
| 5 to 7 |  | reserved |

Notes:

### LDx – Load

**Description**:

Load a value from memory into a register.

**Formats Supported**:

**Register Indirect with Displacement**

This mode may make use of immediate prefixes to extend the range.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 28 | 27 20 | 19 14 | 13 8 | 7 0 |
| Func3..0 | Const8 | Ra6 | Rt6 | 60h8 |

**Scalar Indexed Form (LD)**

The effective address (EA) is calculated as the sum of Ra plus Rb multiplied by a scale.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 28 | 27 | 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Func3..0 | 0 | S | Rb6 | Ra6 | Rt6 | 61h8 |

z: 1= zero extend, 0 = sign extend

|  |  |
| --- | --- |
| Sc3 | Multiplier |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |

***Operation:***

Rt = Memory[d + Ra + Rb \* Sc]

**Vector forms**

**Stridden Form (LDS)**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 40 | 39 36 | 3533 | 32 | 31 27 | 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Const12..5 | Sz4 | m3 | z | Const4..0 | N | Rb6 | Ra6 | Rt6 | 62h8 |

Data is loaded from memory addresses separated by the stride amount specified by register field Rb, beginning with the sum of Ra and an immediate value. If the vector mask bit is clear and the ‘z’ bit is set in the instruction then the corresponding element of the vector register is loaded with zero. If the vector mask bit is clear and the ‘z’ bit is clear in the instruction then the corresponding element of the vector register is left unchanged (no value is loaded from memory).

Elements are loaded only up to the length specified in the vector length register.

|  |  |  |
| --- | --- | --- |
| Vm[x] | z | Result |
| 0 | 0 | Vt[x] = Vt[x] (unchanged) |
| 0 | 1 | Vt[x] = 0 (set to zero) |
| 1 | 0 | Vt[x] = memory, sign extended |
| 1 | 1 | Vt[x] = memory, zero extended |

|  |  |
| --- | --- |
| U2 | Unit |
| 0 | integer |
| 1 | floating-point |
| 2 | decimal-float |
| 3 | posit |

|  |  |
| --- | --- |
| Sz4 | Operation Size |
| 0 | byte |
| 1 | wyde |
| 2 | tetra |
| 3 | octa |
| 4 | hexi (double octa) |
| 5 | quad octa |
| 6 | reserved |
| 7 | pointer |
| … | reserved |
| 15 | cache cmd |

***Operation:***

for x = 0 to vector length

if (Vm[x])

Vt[x] = Memory[d+Ra + Rb \* x]

else

Vt[x] = z ? 0 : Vt[x]

**Indexed Form**

Data is loaded from memory addresses beginning with the sum of Ra and a vector element from Vb.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 50 | 4948 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Const21..8 | U2 | Sz4 | m3 | z | Const7..0 | Vb8 | Ra8 | Rt8 | 63h8 |

***Operation:***

n = 0

for x = 0 to vector length

if (Vm[x])

Vt[x] = Memory[d + Ra + Vb[x]]

else

Vt[x] = z ? 0 : Vt[x]

**Exceptions**: none

### LDB – Load Byte (8 bits)

**Description**:

Data is loaded from the memory address which is the sum of an immediate value and the sum of Ra and Rb times a scale. The value loaded is sign extended from bit 7 to the machine width.

**Formats Supported**: LD

**Operation:**

Rd = Memory8[d + Ra + Rb\*Sc]

**Exceptions**: none

### LDBZ – Load Byte, Zero Extend (8 bits)

**Description**:

Data is loaded from the memory address which is the sum of an immediate value and the sum of Ra and Rb times a scale. The value loaded is zero extended from bit 8 to the machine width.

**Formats Supported**: LD

**Operation:**

Rd = Memory8[d + Ra + Rb\*Sc]

**Exceptions**: none

### LDO – Load Octa (64 bits)

**Description**:

Data is loaded into Rt from the memory address which is the sum of an immediate value and the sum of Ra and Rb scaled.

**Formats Supported**: RR,RI

**Operation:**

Rt = Memory64[d + Ra + Rb\*Sc]

**Execution Units**: Mem

**Exceptions**: none

### LDT – Load Tetra (32 bits)

**Description**:

Data is loaded from the memory address which is the sum of Ra and an immediate value or the sum of Ra and Rb scaled. The value loaded is sign extended from bit 31 to the machine width.

**Formats Supported**: RR,RI

**Operation:**

Rt = Memory32[d + Ra + Rb\*Sc]

**Execution Units**: Mem

**Exceptions**: none

### LDTZ – Load Tetra, Zero Extend (32 bits)

**Description**:

Data is loaded from the memory address which is the sum of Ra and an immediate value or the sum of Ra and Rb scaled. The value loaded is zero extended from bit 8 to the machine width.

**Formats Supported**: RR,RI

**Operation:**

Rt = Memory32[d + Ra + Rb\*Sc]

**Execution Units**: Mem

**Exceptions**: none

### LDW – Load Wyde (16 bits)

**Description**:

Data is loaded from the memory address which is the sum of Ra and an immediate value or the sum of Ra and Rb scaled. The value loaded is sign extended from bit 15 to the machine width.

**Formats Supported**: LD

**Operation:**

Rt = Memory16[d + Ra + Rb\*Sc]

**Execution Units**: Mem

**Exceptions**: none

### LDWZ – Load Wyde, Zero Extend (16 bits)

**Description**:

Data is loaded from the memory address which is the sum of Ra and an immediate value or the sum of Ra and Rb scaled. The value loaded is zero extended from bit 16 to the machine width.

**Formats Supported**: LD

**Operation:**

Rt = Memory16[d + Ra + Rb\*Sc]

**Execution Units**: Mem

**Exceptions**: none

### LEA – Load Effective Address

**Description**:

This instruction computes the effective address for a load/store operation. The data type tag for the target register is set to indicate it contains a pointer.

**Formats Supported**:

**Scalar Indexed Form (LD)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 28 | 27 | 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Func3..0 | 1 | S | Rb6 | Ra6 | Rt6 | 61h8 |

***Operation:***

Rt = d + Ra + Rb \* Sc

**Vector forms**

**Stridden Form (LDS)**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 50 | 4948 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Const21..8 | U2 | Sz4 | m3 | z | Const7..0 | Rb8 | Ra8 | Rt8 | 69h8 |

|  |  |  |
| --- | --- | --- |
| Vm[x] | z | Result |
| 0 | 0 | Vt[x] = Vt[x] (unchanged) |
| 0 | 1 | Vt[x] = 0 (set to zero) |
| 1 | 0 | Vt[x] = memory address |
| 1 | 1 | Vt[x] = memory address |

|  |  |
| --- | --- |
| U2 | Unit |
| 0 | integer |
| 1 | floating-point |
| 2 | decimal-float |
| 3 | posit |

|  |  |
| --- | --- |
| Sz4 | Operation Size |
| 0 | byte |
| 1 | wyde |
| 2 | tetra |
| 3 | octa |
| 4 | hexi |
|  |  |

***Operation:***

for x = 0 to vector length

if (Vm[x])

Vt[x] = d + Ra + Rb \* x

else

Vt[x] = z ? 0 : Vt[x]

**Indexed Form**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 48 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Const23..8 | Sz4 | m3 | z | Const7..0 | Vb8 | Ra8 | Rt8 | 6Ah8 |

***Operation:***

n = 0

for x = 0 to vector length

if (Vm[x])

Vt[x] = d + Ra + Vb[x]

else

Vt[x] = z ? 0 : Vt[x]

**Exceptions**: none

### LSM – Load or Store Multiple

**Description:**

The LSM prefix instruction allows multiple registers or values to be loaded or stored using the following load / store instruction. Register x0 cannot be stored using this prefix. If the register spec field is zero then no load or store takes place at that position. Up to seven registers may be specified.

**Formats Supported:** LSM

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 56 | 55 48 | | 47 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Rg8 | | Rf8 | Re8 | Rd8 | Rc8 | Rb8 | Ra8 | 6Fh8 |

**Execution Units**: Mem

**Exceptions**: none

### STx – Store

**Description**:

Store values to memory. Either the contents of a scalar or vector register or a six-bit immediate constant may be stored. Both scalar and vector store operations are possible.

**Formats Supported**:

**Register Indirect with Displacement**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 28 | 2726 | 25 20 | 19 14 | 13 8 | 7 0 |
| Func3..0 | C2 | Rb6 | Ra6 | Const6 | 60h8 |

**Scalar Indexed Form (ST)**

The effective address (EA) is calculated as the sum of Ra plus Rc multiplied by a scale.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | F8h8 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 28 | 27 | 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Func3..0 | ~ | S | Rb6 | Ra6 | ~6 | 61h8 |

**Scalar Indexed Form (ST)**

The effective address (EA) is calculated as the sum of Ra plus Rb multiplied by a scale and a constant.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 59 48 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Const19..8 | Sz4 | Sc3 | z | Const7..0 | Rb8 | Ra8 | Rs8 | 70h8 |

z: 1= zero extend, 0 = sign extend

|  |  |
| --- | --- |
| Sc3 | Multiplier |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |
| 4 | 16 |

***Operation:***

Memory[d+Ra + Rb \* Sc] = Rs

**Vector forms**

**Stridden Form (STS)**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 50 | 4948 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Const21..8 | U2 | Sz4 | m3 | z | Const7..0 | Rb8 | Ra8 | Rs8 | 72h8 |

Data is stored to memory addresses separated by the stride amount specified by register field Rb, beginning with the sum of Ra and an immediate value. If the vector mask bit is clear and the ‘z’ bit is set in the instruction then memory for the corresponding element of the vector register is stored with zero. If the vector mask bit is clear and the ‘z’ bit is clear in the instruction then memory corresponding to the element of the vector register is left unchanged (no value is stored to memory).

Elements are loaded only up to the length specified in the vector length register.

|  |  |  |
| --- | --- | --- |
| Vm[x] | z | Result |
| 0 | 0 | Memory = Memory (unchanged) |
| 0 | 1 | Memory = 0 (set to zero) |
| 1 | 0 | memory = Vt[x] |
| 1 | 1 | memory = Vt[x] |

|  |  |
| --- | --- |
| U2 | Unit |
| 0 | integer |
| 1 | floating-point |
| 2 | decimal-float |
| 3 | posit |

|  |  |
| --- | --- |
| Sz4 | Operation Size |
| 0 | byte |
| 1 | wyde |
| 2 | tetra |
| 3 | octa |
| 4 | hexi |
| 5,6 | reserved |
| 7 | pointer |

***Operation:***

for x = 0 to vector length

if (Vm[x])

Memory[d+Ra + Rb \* x] = Vt[x]

else

Memory[d+Ra + Rb \* x] = z ? 0 : Memory[d+Ra + Rb \* x]

**Indexed Form**

Data is stored to memory addresses beginning with the sum of Ra and a vector element from Vb.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 48 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Const23..8 | Sz4 | m3 | z | Const7..0 | Vb8 | Ra8 | Rt8 | 73h8 |

***Operation:***

n = 0

for x = 0 to vector length

if (Vm[x])

Memory[d + Ra + Vb[x]] = Vt[x]

else

Memory = z ? 0 : Memory

**Exceptions**: none

### STB – Store Byte (8 bits)

**Description:**

This instruction stores a byte (8 bit) value to memory. The memory address is calculated as the sum of an immediate constant and the sum of Ra and Rb scaled.

**Instruction Format**: ST

**Operation:**

Memory8[d + Ra + Rb\*Sc] = Rs

### STBZ – Store Byte and Zero (8 bits)

**Description:**

This instruction stores a byte (8 bit) value to memory. The memory address is calculated as the sum of an immediate constant and the sum of Ra and Rb scaled. After the byte is stored to memory the register is zeroed out.

**Instruction Format**: ST

**Operation:**

Memory8[d + Ra + Rb\*Sc] = Rs

Rs = 0

### STO – Store Octa (64 bits)

**Description:**

This instruction stores an octa-byte (64 bit) value to memory. The memory address is calculated as the sum of an immediate constant and the sum of Ra and Rb scaled.

**Instruction Format:** ST

**Operation:**

Memory64[d + Ra + Rb\*Sc] = Rs

### STOZ – Store Octa and Zero (64 bits)

**Description:**

This instruction stores an octa-byte (64 bit) value to memory. The memory address is calculated as the sum of an immediate constant and the sum of Ra and Rb scaled. After the tetra is stored to memory the register is zeroed out.

**Instruction Format:** ST

**Operation:**

Memory64[d + Ra + Rb\*Sc] = Rs

Rs = 0

### STPTR – Store Pointer (64 bits)

**Description:**

This instruction stores an octa-byte (64 bit) value to memory. The memory address is calculated as the sum of an immediate constant and the sum of Ra and Rb scaled. STPTR begins a series of stores to memory addresses scaled by eight bits, until the address zero is reached. The first store proceeds normally, for the second and subsequent stores a byte store operation takes place with the value zero being to memory.

The purpose of the STPTR instruction is to allow a code dense implementation of a write barrier that indicates where in memory a pointer is stored with increasing resolution.

This instruction assumes that card memory used to record pointer locations is located at the low end of the memory system.

**Instruction Format:** ST

**Operation:**

ea = d + Ra + Rb\*Sc

Memory64[ea] = Rs

while ea <> 0

ea = ea >> 8

Memory8[ea] = 0

### STT – Store Tetra (32 bits)

**Description:**

This instruction stores a tetra-byte (32 bit) value to memory. The memory address is calculated as the sum of an immediate constant and the sum of Ra and Rb scaled.

**Instruction Format:** ST

**Operation:**

Memory32[d + Ra + Rb\*Sc] = Rs

### STTZ – Store Tetra and Zero (32 bits)

**Description:**

This instruction stores a tetra-byte (32 bit) value to memory. The memory address is calculated as the sum of an immediate constant and the sum of Ra and Rb scaled. After the tetra is stored to memory the register is zeroed out.

**Instruction Format:** ST

**Operation:**

Memory32[d + Ra + Rb\*Sc] = Rs

Rs = 0

### STW – Store Wyde (16 bits)

**Description:**

This instruction stores a byte (16 bit) value to memory. The memory address is calculated as the sum of an immediate constant and the sum of Ra and Rb scaled.

**Instruction Format:** ST

**Operation:**

Memory16[d + Ra + Rb\*Sc] = Rs

### STWZ – Store Wyde and Zero (16 bits)

**Description:**

This instruction stores a byte (16 bit) value to memory. The memory address is calculated as the sum of an immediate constant and the sum of Ra and Rb scaled. After the wyde is stored to memory the register is zeroed out.

**Instruction Format:** ST

**Operation:**

Memory16[d + Ra + Rb\*Sc] = Rs

Rs = 0

## Flow Control (Branch Unit) Operations

### Branches

The branch modifier may be used to make it possible to branch to a target address contained in a register, and to store the return address in a register. Simultaneously the branch displacement is increased to 24 bits allowing a ±32MB branch range.

### BAL – Branch and Link

**Description**:

This instruction may be used to call a subroutine. The address of the instruction after the BAL is stored in the specified return address register (Rt) then a jump to the address specified in the instruction is made. The address range is 24 bits or ±8MB. The constant is shifted left twice before use.

The return address register is assumed to be x1 if not otherwise specified. The BAL instruction does not require space in branch predictor tables.

**Formats Supported**: BAL

|  |  |  |
| --- | --- | --- |
| 31 10 | 9 8 | 7 0 |
| Constant22 | Rt2 | 41h8 |

**Flags Affected**: none

**Operation:**

Rt = IP + 4

IP = IP + displacement

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

### BEQ – Branch if Equal

**Description**:

This instruction branches to the target address if the contents of Ra and Rb are equal, otherwise program execution continues with the next instruction. With a branch modifier instruction, the target address is formed as the sum of Rc and a displacement. If Rc is r63 then the instruction pointer value is used. Otherwise, the target address is the sum of the instruction pointer value and the displacement specified in the instruction.

**Formats Supported**: BR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Constant6 | Rb6 | Ra6 | Const6 | 4Eh8 |

**Operation:**

If (Ra = Rb)

IP = IP + Displacement14

With Modifier

Rt = IP + 4

If (Ra = Rb)

IP = Rc + Displacement23

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

For a floating-point comparison positive and negative zero are considered equal.

### BGE – Branch if Greater Than or Equal

**Description**:

This instruction branches to the target address if the contents of Ra is greater than or equal to Rb, otherwise program execution continues with the next instruction. The values in Ra and Rb are treated as signed values. The target address is formed as the sum of Rc and a displacement. If Rc is x63 then the instruction pointer value is used.

**Formats Supported**: BR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Constant6 | Rb6 | Ra6 | Const6 | 49h8 |

**Operation:**

If (Ra >= Rb)

IP = IP + Displacement

**Execution Units**: Branch

**Exceptions:** none

### BGEU – Branch if Greater Than or Equal Unsigned

**Description**:

This instruction branches to the target address if the contents of Ra is greater than or equal to Rb, otherwise program execution continues with the next instruction. The values in Ra and Rb are treated as unsigned values. The target address is formed as the sum of Rc and a displacement. If Rc is r63 then the program counter value is used.

**Formats Supported**: BR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Constant6 | Rb6 | Ra6 | Const6 | 4Bh8 |

**Operation:**

Rt = IP + 8

If (Ra >= Rb)

PC = Rc + Displacement

**Execution Units**: Branch

**Exceptions:** none

### BGT – Branch if Greater Than

**Description**:

This instruction is an alternate mnemonic for the [BLT](#_BLT_–_Branch) instruction where the register operands have been swapped.

This instruction branches to the target address if the contents of Ra is less than Rb, otherwise program execution continues with the next instruction. The values in Ra and Rb are treated as signed values. The target address is formed as the sum of Rc and a displacement. If Rc is x63 then the program counter value is used.

**Formats Supported**: BR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Constant6 | Rb6 | Ra6 | Const6 | 48h8 |

**Operation:**

Rt = IP + 8

If (Ra < Rb)

PC = Rc + Displacement

**Execution Units**: Branch

**Exceptions**: none

### BGTU – Branch if Greater Than Unsigned

**Description**:

This instruction is an alternate mnemonic for the [BLTU](#_BLTU_–_Branch) instruction where the register operands have been swapped.

This instruction branches to the target address if the contents of Ra is less than Rb, otherwise program execution continues with the next instruction. The values in Ra and Rb are treated as unsigned values. The target address is formed as the sum of Rc and a displacement. If Rc is x63 then the program counter value is used.

**Formats Supported**: BR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| Constant6 | Rb6 | Ra6 | Const6 | 4Ah8 |

**Operation:**

Rt = IP + 8

If (Ra < Rb)

PC = Rc + Displacement

**Execution Units**: Branch

**Exceptions**: none

### BNE – Branch if Not Equal

**Description**:

This instruction branches to the target address if the contents of Ra and Rb are not equal, otherwise program execution continues with the next instruction. The target address is formed as the sum of Rc and a displacement. If Rc is x63 then the program counter value is used.

**Formats Supported**: BR

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 50 | 4948 | 47 44 | 43 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Displacement20..7 | U2 | Sz4 | Disp6..3 | Rc8 | Rb8 | Ra8 | Rt8 | 4Fh8 |

**Operation:**

Rt = IP + 8

If (Ra <> Rb)

PC = Rc + Displacement

**Execution Units**: Branch

**Exceptions**: none

### BLE – Branch if Less Than or Equal

**Description**:

This is an alternate mnemonic for the BGE instruction, where the register operands have been swapped.

This instruction branches to the target address if the contents of Ra is greater than or equal to Rb, otherwise program execution continues with the next instruction. The values in Ra and Rb are treated as signed values. The target address is formed as the sum of Rc and a displacement. If Rc is x63 then the program counter value is used.

**Formats Supported**: BR

**Operation:**

If (Ra >= Rb)

PC = Rc + Displacement

**Execution Units**: Branch

**Exceptions:** none

### BLEU – Branch if Less Than or Equal Unsigned

**Description**:

This is an alternate mnemonic for the BGEU instruction, where the register operands have been swapped.

This instruction branches to the target address if the contents of Ra is greater than or equal to Rb, otherwise program execution continues with the next instruction. The values in Ra and Rb are treated as unsigned values. The target address is formed as the sum of Rc and a displacement. If Rc is x63 then the program counter value is used.

**Formats Supported**: BR

**Operation:**

If (Ra >= Rb)

PC = Rc + Displacement

**Execution Units**: Branch

**Exceptions:** none

### BLT – Branch if Less Than

**Description**:

This instruction branches to the target address if the contents of Ra is less than Rb, otherwise program execution continues with the next instruction. The values in Ra and Rb are treated as signed values. The target address is formed as the sum of Rc and a displacement. If Rc is x63 then the program counter value is used.

**Formats Supported**: BR

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 50 | 4948 | 47 44 | 43 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Displacement20..7 | U2 | Sz4 | Disp6..3 | Rc8 | Rb8 | Ra8 | Rt8 | 48h8 |

**Operation:**

Rt = IP + 8

If (Ra < Rb)

PC = Rc + Displacement

**Execution Units**: Branch

**Exceptions**: none

### BLTU – Branch if Less Than Unsigned

**Description**:

This instruction branches to the target address if the contents of Ra is less than Rb, otherwise program execution continues with the next instruction. The values in Ra and Rb are treated as unsigned values. The target address is formed as the sum of Rc and a displacement. If Rc is x63 then the program counter value is used.

**Formats Supported**: BR

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 50 | 4948 | 47 44 | 43 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Displacement20..7 | U2 | Sz4 | Disp6..3 | Rc8 | Rb8 | Ra8 | Rt8 | 4Ah8 |

**Operation:**

Rt = IP + 8

If (Ra < Rb)

PC = Rc + Displacement

**Execution Units**: Branch

**Exceptions**: none

### BRA – Unconditional Branch

**Description**:

This instruction is an alternate mnemonic for the [BAL](#_BAL_–_Branch) instruction. The address range is 24 bits or ±8MB. The constant field is shifted left twice before use.

**Formats Supported**: JAL

|  |  |  |
| --- | --- | --- |
| 31 10 | 9 8 | 7 0 |
| Constant22 | 02 | 41h8 |

**Flags Affected**: none

**Operation:**

IP = IP + Displacement

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

### CHK – Check Register Against Bounds

**Description**:

A register is compared to two values. If the register is outside of the bounds then an exception will occur.

**Instruction Format: RI**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | F7h8 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 20 | 19 14 | 13 10 | 9 8 | 7 0 |
| Constant12 | Ra6 | ~ | Cn2 | 22h8 |

|  |  |
| --- | --- |
| Cn2 | Interpretation |
| 0 | Ra <= Rc <= Constant |
| 1 | Ra < Rc <= Constant |
| 2 | Ra <= Rc < Constant |
| 3 | Ra < Rc < Constant |

**Instruction Format**: R3

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 26 | 25 20 | 19 14 | 1312 | 11 9 | 8 | 7 0 |
| DT3 | Rm3 | Rd6 | Rc6 | A | m3 | z | F7h8 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 10 | 9 8 | 7 0 |
| 22h6 | Rb6 | Ra6 | ~ | Cn2 | 03h8 |

|  |  |
| --- | --- |
| Cn2 | Interpretation |
| 0 | Ra <= Rb <= Rc |
| 1 | Ra < Rb <= Rc |
| 2 | Ra <= Rb < Rc |
| 3 | Ra < Rb < Rc |

**Supported Formats**: .o

**Clock Cycles**: 2

**Execution Units:** Integer ALU, Float, Decimal Float, Posit

**Exceptions**: bounds check

**Notes**:

The system exception handler will typically transfer processing back to a local exception handler.

### JAL – Jump and Link

**Description**:

This instruction may be used to both call a subroutine and return from it. The address of the instruction after the JAL is stored in the specified return address register (Rt) then a jump to the address specified in the instruction is made. The address range is 24 bits or ±8MB. The constant field is shifted left twice before use.

The return address register is assumed to be x1 if not otherwise specified. The JAL instruction does not require space in branch predictor tables.

**Formats Supported**: JAL

|  |  |  |
| --- | --- | --- |
| 31 10 | 9 8 | 7 0 |
| Constant22 | Rt2 | 40h8 |

**Flags Affected**: none

**Operation:**

Rt = IP + 4

IP = IP + displacement (BAL)

OR IP = displacement (JAL)

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

### JALR – Jump and Link to Register

**Description**:

This instruction may be used to both call a subroutine and return from it. The sum of the current IP and a small constant is stored in the specified return address register (Rt) then a jump to the address specified in the instruction plus an index register value is made.

The return address register is assumed to be x1 if not otherwise specified. The JALR instruction does not require space in branch predictor tables.

If x63 is specified for Ra then the current instruction pointer value is used.

**Formats Supported**: JALR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 20 | 19 14 | 13 10 | 9 8 | 7 0 |
| Constant12 | Ra6 | Cnst4 | Rt2 | 42h8 |

**Flags Affected**: none

**Operation:**

Rt = IP + Cnst4\*2

If Ra=63

IP = IP + displacement

Else

IP = Ra + Displacement

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

### JMP – Jump

**Description**:

This instruction is an alternate mnemonic for the [JAL](#_JAL_–_Jump) instruction. It may be used to jump directly to a specific address. The address range is 24 bits or ±8MB. The constant field is shifted left twice before use.

The return address register is assumed to be x0 (discarding the return address). The JMP instruction does not require space in branch predictor tables.

**Formats Supported**: JAL

|  |  |  |
| --- | --- | --- |
| 31 10 | 9 8 | 7 0 |
| Constant22 | 02 | 40h8 |

**Flags Affected**: none

**Operation:**

IP = displacement

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

### RET – Return from Subroutine

**Description**:

This instruction is an alternate mnemonic for the [JALR](#_JALR_–_Jump) instruction. Register Ra is assumed to be x1 and register Rt is assumed to be x0. The constant is assumed to be zero.

**Formats Supported**: JALR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 20 | 19 14 | 13 10 | 9 8 | 7 0 |
| Constant12 | 016 | 04 | 02 | 42h8 |

**Flags Affected**: none

**Operation:**

**Execution Units**: Branch

**Exceptions**: an unimplemented instruction exception may occur if a vector register is specified.

**Notes**:

Return address prediction hardware may make use of the RET instruction.

## System Instructions

### BRK – Break

**Description**:

This instruction initiates the processor debug routine. The processor enters debug mode. The cause code register is set to the value specified in the instruction. Interrupts are disabled. The instruction pointer is reset to the contents of tvec[4] and instructions begin executing. There should be a jump instruction placed at the break vector location. The address of the BRK instruction is stored in the EIP register.

**Instruction Format**: BRK

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 22 | 21 14 | 13 8 | 7 0 |
| ~6 | ~4 | Cause8 | 06 | 00h8 |

**Operation:**

PMSTACK = (PMSTACK << 4) | 10

CAUSE = Const8

EIP = IP

IP = tvec[4]

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

### CSRx – Control and Special / Status Access

**Description**:

The CSR instruction group provides access to control and special or status registers in the core. For the read operation the current value of the CSR is placed in the target register Rt.

This instruction is usually used with an extended immediate modifier, however it may be used without the modifier in which case only a read of user CSRs is possible.

**Instruction Format**: CSR

|  |  |  |  |
| --- | --- | --- | --- |
| 31 19 | 18 16 | 15 8 | 7 0 |
| ~ | Op3 | Const8 | F06 |

|  |  |  |  |
| --- | --- | --- | --- |
| 31 20 | 19 14 | 13 8 | 7 0 |
| Constant12 | Ra6 | Rt6 | 0Fh8 |

|  |  |  |
| --- | --- | --- |
| Op3 |  | Operation |
| 0 | CSRR | Only read the CSR, no update takes place, Ra should be R0. |
| 1 | CSRW | Write to CSR |
| 2 | CSRS | Set CSR bits |
| 3 | CSRC | Clear CSR bits |
| 4 to 7 |  | reserved |

CSRS and CSRC operations are only valid on registers that support the capability.

The Regno[15..12] field is reserved to specify the operating mode. Note that registers cannot be accessed by a lower operating mode.

**Execution Units:** Integer, the instruction may be available on only a single execution unit (not supported on all available integer units).

**Clock Cycles**: 1

**Exceptions**: privilege violation attempting to access registers outside of those allowed for the operating mode.

### PFI – Poll for Interrupt

**Description**:

The poll for interrupt instruction polls the interrupt status lines and performs an interrupt service if an interrupt is present. Otherwise, the PFI instruction is treated as a NOP operation. Polling for interrupts is performed by managed code. PFI provides a means to process interrupts at specific points in running software.

**Instruction Format: SYS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 11h6 | 06 | 06 | 06 | 44h8 |

**Clock Cycles**: 1 (if no exception present)

**Execution Units: Branch**

### REX – Redirect Exception

**Description**:

This instruction redirects an exception from an operating mode to a lower operating mode. This instruction if successful jumps to the target exception handler and does not return. If this instruction fails execution will continue with the next instruction.

This instruction may fail if exceptions are not enabled at the target level.

The location of the target exception handler is found in the trap vector register for that operating mode (tvec[xx]).

The cause (cause) and bad address (badaddr) registers of the originating mode are copied to the corresponding registers in the target mode.

**Instruction Format**: REX

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 59 | 58 56 | 55 48 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| ~ | Rm3 | 7Ah8 | Tm3 | m3 | z | Rc8 | Imm8 | Ra8 | 08 | 44h8 |

|  |  |
| --- | --- |
| Tm3 |  |
| 0 | redirect to user mode |
| 1 | redirect to supervisor mode |
| 2 | redirect to hypervisor mode |
| 3 | redirect to machine mode |
| 4 to 7 | not used |

**Clock Cycles**: 4

**Execution Units: Branch**

Example:

|  |
| --- |
| REX 1 ; redirect to supervisor handler  ; If the redirection failed, exceptions were likely disabled at the target level.  ; Continue processing so the target level may complete its operation.  RTE ; redirection failed (exceptions disabled ?) |

**Notes**:

Since all exceptions are initially handled in debug mode the debug handler must check for disabled lower mode exceptions.

### RTE – Return from Exception

**Description**:

Restore the previous interrupt enable setting and operating level and transfer program execution back to the address in the exception address register (EIP). One of sixty-four semaphore registers specified by the Rb field of the instruction may also be cleared. Semaphore register zero is always cleared by this instruction.

This instruction may be encoded to return a short distance past the exception address point. This may be useful to return to the next instruction or return to a point past inline parameters. The Ra field specifies a return offset in terms of instruction words.

There is really only a single instruction to return from any mode for an exception. Although there are several additional mnemonics.

**Instruction Format: SYS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 13h6 | Rb6 | Ra6 | Opt6 | 44h8 |

Opt6[0]: 0 = Ra is reg spec, 1 = Ra is six-bit unsigned immediate

Opt6[1]: 0 = Rb is reg spec, 1 = Rb is six-bit unsigned immediate

**Flags Affected**: none

**Operation:**

PMSTACK = PMSTACK >> 4

Semaphore[0] = 0

Semaphore[Rb] = 0

IP = EIP + Ra

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

### SYNC -Synchronize

Description:

All instructions for a particular unit before the SYNC are completed and committed to the architectural state before instructions of the unit type after the SYNC are issued. This instruction is used to ensure that the machine state is valid before subsequent instructions are executed.

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 6361 | 60 58 | 57 50 | 4948 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| ~3 | Op3 | ??h8 | U2 | Sz4 | m3 | z | Rc8 | Rb8 | Ra8 | Rt8 | 44h8 |

### TLBRW – Read / Write TLB

**Description**:

This instruction both reads and writes the TLB. Which translation entry to update comes from the value in Ra. The update value comes from the value in Rb. Rb contains the virtual page number, ASID, and physical page number. The current value of the entry selected by Ra is copied to Rt. The TLB will be written only if bit 63 of Ra is set.

The entry number for Ra comes from virtual address bits 14 to 23.

Page numbers are in terms of a 16kB page size.

**Instruction Format: SYS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 1Eh6 | Rb6 | Ra6 | Rt6 | 44h8 |

**Clock Cycles**: 5

**Execution Units: Memory**

Ra Value Format

|  |  |  |  |
| --- | --- | --- | --- |
| 63 | 62 12 | 11 10 | 9 0 |
| w | ~ | way | entry no |

Rb/Rt Value Format

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 63 56 | 55 | 54 | 53 | 52 48 | 47 32 | 31 20 | 19 0 |
| ASID | G | D | A | UCRWX | VPN | ~ | PPN |

|  |  |  |  |
| --- | --- | --- | --- |
| Bits |  | Meaning | |
| 0 to 19 | PPN | Physical page number | |
| 20 to 31 | ~ | reserved (expansion of physical page number) | |
| 32 to 49 | VPN | Virtual page number high address order bits 24 to 39 | |
| 48 | X | 1 = page is executable | These three combined indicate page present (P) 0 = not present |
| 49 | W | 1 = page is writeable |
| 50 | R | 1 = page is readable |
| 51 | C | 1 = page is cachable | |
| 52 | U | reserved for system usage | |
| 53 | A | Accessed, set if translation was used | |
| 54 | D | Dirty, set if a write occurred to the page | |
| 55 | G | Global, global translation indicator | |
| 56 to 63 | ASID | ASID address space identifier | |

**Exceptions:** none

### WFI – Wait for Interrupt

**Description**:

The WFI instruction waits for an external interrupt to occur before proceeding. While waiting for the interrupt, the processor clock is stopped placing the processor in a lower power mode.

**Instruction Format: SYS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 12h6 | 06 | 06 | 06 | 44h8 |

**Clock Cycles**: 1 (if no exception present)

**Execution Units: Branch**

# Vector Specific Instructions

## Arithmetic / Logical

### V2BITS

Synopsis

Convert Boolean vector to bits.

**Description**

The least significant bit of each vector element is copied to the corresponding bit in the target register. The target register is a scalar register.

**Instruction Format: R1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 26 | 25 20 | 19 14 | 13 8 | 7 0 |
| 18h6 | ~6 | Ra6 | Rt6 | 81h8 |

**Operation**

For x = 0 to VL-1

if (Vm[x])

Rt[x] = Va[x].LSB

else if (z)

Rt[x] = 0

**Exceptions:** none

### VACC - Accumulate

**Synopsis**

Register accumulation. Rt = Va + Rb

**Description**

A vector register (Va) and scalar register (Rb) are added together and placed in the target scalar register Rt. Rb and Rt may be the same register which results in an accumulation of the values in the register.

**Instruction Format:** V2

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Rt = Va[x] + Rb

**Example**

ldi x1,#0 ; clear results

vfmul.s v1,v2,v3 ; multiply inputs (v2) times weights (v3)

vfacc.s x1,v1,x1 ; accumulate results

fadd.s x1,x1,x2 ; add bias (r2 = bias amount)

fsigmoid.s x1,x1 ; compute sigmoid

### VBITS2V

Synopsis

Convert bits to Boolean vector.

**Description**

Bits from a general register are copied to the corresponding vector target register.

**Operation**

For x = 0 to VL-1

if (Vm[x]) Vt[x] = Ra[x]

**Exceptions:** none

### VCIDX – Compress Index

**Synopsis**

Vector compression.

**Description**

A value in a register Ra is multiplied by the element number and copied to elements of vector register Vt guided by a vector mask register.

**Operation**

y = 0

for x = 0 to VL - 1

if (Vm[x])

Vt[y] = Ra \* x

y = y + 1

### VCMPRSS – Compress Vector

**Synopsis**

Vector compression.

**Description**

Selected elements from vector register Va are copied to elements of vector register Vt guided by a vector mask register.

**Operation**

y = 0

for x = 0 to VL - 1

if (Vm[x])

Vt[y] = Va[x]

y = y + 1

### VEINS / VMOVSV – Vector Element Insert

**Synopsis**

Vector element insert.

**Description**

A general-purpose register Rb is transferred into one element of a vector register Vt. The element to insert is identified by Ra.

**Operation**

Vt[Ra] = Rb

Exceptions: none

### VEX / VMOVS – Vector Element Extract

**Synopsis**

Vector element extract.

**Description**

A vector register element from Vb is transferred into a general-purpose register Rt. The element to extract is identified by Ra.

**Operation**

Rt = Vb[Ra]

**Exceptions**: none

### VSCAN

**Synopsis**

.

**Description**

Elements of Vt are set to the cumulative sum of a value in register Ra. The summation is guided by a vector mask register.

**Operation**

sum = 0

for x = 0 to VL - 1

Vt[x] = sum

if (Vm[x])

sum = sum + Ra

### VSHLV – Shift Vector Left

**Synopsis**

Vector shift left.

**Description**

Elements of the vector are transferred upwards to the next element position. The first is loaded with the value zero. This is also called a slide operation.

**Operation**

For x = VL-1 to Amt

Vt[x] = Va[x-amt]

For x = Amt-1 to 0

Vt[x] = 0

**Exceptions:** none

### VSHRV – Shift Vector Right

**Synopsis**

Vector shift right.

**Description**

Elements of the vector are transferred downwards to the next element position. The last is loaded with the value zero. This is also called a slide operation.

**Operation**

For x = 0 to VL-Amt

Vt[x] = Va[x+amt]

For x = VL-Amt +1 to VL-1

Vt[x] = 0

**Exceptions:** none

## Memory Operations

### CVLDx – Compressed Vector Load

**Description**:

**Formats Supported**:

**Stridden Form**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 50 | 4948 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Const21..8 | U2 | Sz4 | m3 | z | Const7..0 | Rb8 | Ra8 | Rt8 | 65h8 |

Data is loaded from memory locations beginning at the sum of Ra and a constant and separated by the stride amount in the stride register Rb. Rb may also be a constant in the range -62 to 63. If Rb = -63 then the Sz4 field is used to determine the stride.

***Operation:***

y = 0

for x = 0 to vector length

if Rb is a constant

if Rb = -63

stride = Sz4

else

stride = Rb

else

stride = [Rb]

n = stride \* y

if (Vm[x])

Vt[y] = Memory[d+Ra + n]

y = y + 1

for y = y to vector length

Vt[y] = z ? 0 : Vt[y]

n = 0

If the vector mask bit is clear and the ‘z’ bit is set in the instruction then the corresponding element of the vector register is loaded with zero. If the vector mask bit is clear and the ‘z’ bit is clear in the instruction then the corresponding element of the vector register is left unchanged (no value is loaded from memory).

Elements are loaded only up to the length specified in the vector length register.

|  |  |  |
| --- | --- | --- |
| Vm[x] | z | Result |
| 0 | 0 | Vt[x] = Vt[x] (unchanged) |
| 0 | 1 | Vt[x] = 0 (set to zero) |
| 1 | 0 | Vt[x] = memory, sign extended |
| 1 | 1 | Vt[x] = memory, zero extended |

***Operation:***

n = 0

y = 0

for x = 0 to vector length

if (Vm[x])

Vt[y] = Memory[d+Ra + n]

n = n + sizeof precision

y = y + 1

for y = y to vector length

Vt[y] = z ? 0 : Vt[y]

**Indexed Form**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 50 | 4948 | 47 44 | 4341 | 40 | 39 32 | 31 24 | 23 16 | 15 8 | 7 0 |
| Const21..8 | U2 | Sz4 | m3 | z | Const7..0 | Rb8 | Ra8 | Rt8 | 66h8 |

Data is loaded from memory addresses beginning with the sum of Ra and a vector element from Vb.

***Operation:***

y = 0

for x = 0 to vector length

if (Vm[x])

Vt[y] = Memory[d+Ra + Vb[x]]

y = y + 1

for y = y to vector length

Vt[y] = z ? 0 : Vt[y]

**Exceptions**: none

### CVSTx – Compressed Vector Store

**Description**:

**Formats Supported**:

**Register Indirect with Displacement**

Data is stored to consecutive memory addresses beginning with the sum of Ra and an immediate

Elements are stored only up to the length specified in the vector length register.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 42 | 4140 | 39 36 | 35 33 | 32 | 31 20 | 19 14 | 13 8 | 7 | 6 0 |
| Const6 | U2 | Sz4 | m3 | z | Constant12 | Ra6 | Vs6 | 1 | 74h7 |

|  |  |  |
| --- | --- | --- |
| Vm[x] | z | Result |
| 1 | 0 | memory = Vs[x] |
| 1 | 1 | memory = Vs[x], Vs[x] = 0 |

***Operation:***

n = 0

for x = 0 to vector length

if (Vm[x])

Memory[d+Ra + n] = Vs[x]

if (z) Vs[x] = 0

n = n + sizeof precision

**Stridden Form**

The stridden form works much the same as the register indirect form except that data is stored to memory locations separated by the stride amount in the stride register.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 42 | 4140 | 39 36 | 35 33 | 32 | 31 26 | 25 20 | 19 14 | 13 8 | 7 | 6 0 |
| Const6 | U2 | Sz4 | m3 | z | Const6 | Rb6 | Ra6 | Vs6 | 1 | 75h7 |

***Operation:***

y = 0

for x = 0 to vector length

n = Rb \* y

if (Vm[x])

Memory[d+Ra + n] = Vs[x]

if (z) Vs[x] = 0

y = y + 1

**Indexed Form**

Data is stored to memory addresses beginning with the sum of Ra and a vector element from Vb.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 42 | 4140 | 39 36 | 35 33 | 32 | 31 26 | 25 20 | 19 14 | 13 8 | 7 | 6 0 |
| Const6 | U2 | Sz4 | m3 | z | Const6 | Vb6 | Ra6 | Vs6 | 1 | 76h7 |

***Operation:***

y = 0

for x = 0 to vector length

if (Vm[x])

Memory[d+Ra + Vb[y]] = Vs[x]

if (z) Vs[x] = 0

y = y + 1

**Exceptions**: none

# Root Opcode Map

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| **ALU** | | | | | | | | |
| 00000 | BRK | {R1} | {R2} | {R3} | ADD | SUBF | MUL |  |
| 00001 | AND | OR | XOR |  |  | {SET} | MULU | CSR |
| 00010 | DIV | DIVU | DIVSU |  |  | MULF | MULSU | PERM |
| 00011 | REM | REMU | BYTNDX | WYDNDX | {BTFLD} |  |  |  |
| 00100 | REMSU | DIVR | CHK | U21NDX | SAND | SOR | SEQ | SNE |
| 00101 | SLT | SGT | SLTU | SGTU |  |  |  |  |
| 00110 | MADD | MSUB | NMADD | NMSUB |  |  |  | FDP |
| 00111 |  |  |  |  |  |  |  | NOP |
| **Branch Unit** | | | | | | | | |
| 01000 | JAL | BAL | JALR |  | {SYS} |  |  |  |
| 01001 | BLT | BGE | BLTU | BGEU |  | BBS | BEQ | BNE |
| **Instruction Modifiers (Prefixes)** | | | | | | | | |
| 01010 | EXI | EXI | EXI |  |  |  |  |  |
| 01011 | IMOD | BTFLD | BRMOD | FB |  |  |  |  |
| **Memory Unit** | | | | | | | | |
| 01100 | LDx | LDxX | LDS | LDVX |  |  | CVLDS | CVLDVX |
| 01101 |  |  |  |  |  |  |  | LSM |
| 01110 | STx | STxX | SDS | STVX |  |  | CVSTS | CVSTVX |
| 01111 |  |  |  |  |  |  |  |  |
| **Vector ALU** | | | | | | | | |
| 10000 |  | {R1} | {R2} | {R3} | ADD | SUBF | MUL |  |
| 10001 | AND | OR | XOR |  |  |  | MULU |  |
| 10010 | DIV | DIVU | DIVSU |  |  | MULF | MULSU | PERM |
| 10011 | REM | REMU | BYTNDX | WYDNDX | {BTFLD} |  |  |  |
| 10100 | REMSU | DIVR | CHK | U21NDX |  |  | SEQ | SNE |
| 10101 | SLT | SGT | SLTU | SGTU |  |  |  |  |
| 10110 | MADD | MSUB | NMADD | NMSUB |  |  |  | FDP |
| 10111 |  |  |  |  |  |  |  | NOP |
| 11000 |  |  |  |  |  |  |  |  |
| 11001 |  |  |  |  |  |  |  |  |
| 11010 |  |  |  |  |  |  |  |  |
| 11011 | IMOD | BTFLD | BRMOD | FB |  |  |  |  |
|  | | | | | | | | |
| 11100 |  |  |  |  |  |  |  |  |
| 11101 |  |  |  |  |  |  |  |  |
| 11110 |  |  |  |  |  |  |  |  |
| 11111 |  |  |  |  |  |  |  |  |

## {SR3} Triadic Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 000 |  |  |  |  | MUX |  |  |  |
| 001 |  |  |  |  |  |  |  |  |
| 010 | SLLP | SLLPI |  |  |  |  |  |  |
| 011 | PTRDIF |  |  |  |  |  |  |  |
| 100 |  |  | CHK |  |  |  |  |  |
| 101 |  |  |  |  |  |  |  |  |
| 110 | BLEND |  |  |  |  |  |  |  |
| 111 |  |  |  |  |  |  |  |  |

## {SR2} Dyadic Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 000 | AND | OR | XOR | BMM | ADD | SUB | MUL |  |
| 001 | NAND | NOR | XNOR |  |  | MULF | MULU | MULH |
| 010 | DIV | DIVU | DIVSU | REM | REMU | REMSU | MULSU | PERM |
| 011 | DIF | SLL | SLLI |  | MULF | MULSUH | MULUH | RGF |
| 100 |  |  |  |  |  |  | SEQ | SNE |
| 101 | MIN | MAX | CMP |  | SLT | SGE | SLTU | SGEU |
| 110 |  |  |  |  |  |  |  |  |
| 111 |  |  |  |  |  |  |  |  |

## {SR1} Monadic Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 00 | CNTLZ | CNTLO | CNTPOP | COM | NOT | NEG | ABS |  |
| 01 |  |  |  | TST |  |  |  |  |
| 10 | PTRINC |  |  |  |  |  |  |  |
| 11 | V2BITS | BITS2V | VEX |  |  |  |  |  |

{OSR2} System Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 00 | LLAL | LLAH |  |  | LPAL | LPAH |  |  |
| 01 | PUSHQ | POPQ | PEEKQ | STATQ | SETKEY | GCCLR |  |  |
| 10 | REX | PFI | WAI | RTE |  |  |  |  |
| 11 | SETTO | GETTO | GETZL |  | MVMAP | MVSEG | TLBRW | SYNC |