

# ANY-1 Instruction Set

© 2021 Robert Finch

## Table of Contents

Instruction Formats .....	5
Example Instruction .....	6
Instructions.....	7
Arithmetic / Logical .....	7
ABS – Absolute Value.....	7
ADD - Addition .....	9
ADDIS – Add Immediate Shifted.....	<b>Error! Bookmark not defined.</b>
AND – Bitwise And.....	11
ANDIS – Bitwise And Immediate Shifted.....	<b>Error! Bookmark not defined.</b>
AISIP – Add Immediate Shifted to IP .....	13
BMM – Bit Matrix Multiply .....	14
BYTNDX – Byte Index .....	16
CNTLZ – Count Leading Zeros.....	21
CNTPOP – Count Population .....	20
CSRx – Control and Status Access .....	105
DEP – Deposit.....	23
DIV – Division.....	24
DIVR – Division .....	26
DIVU – Division Unsigned.....	26
EXT –Extract Bitfield .....	28
EXTU –Extract Bitfield Unsigned .....	29
FDP – Fused Dot Product .....	29
FFO –Find First One .....	30
MAX – Maximum Value .....	31
MIN – Minimum Value .....	32
MUL – Signed Multiply.....	33
MULF – Fast Unsigned Multiply.....	36
MUX – Multiplex .....	37
NEG - Negate.....	40

NOT – Logical Not .....	41
OR – Bitwise Or.....	42
ORIS – Bitwise Or Immediate Shifted .....	<b>Error! Bookmark not defined.</b>
PERM – Permute Bytes .....	44
SEQ – Set if Equal .....	45
SGE – Set if Greater Than or Equal.....	46
SGEU – Set if Greater Than or Equal Unsigned.....	49
SGT – Set if Greater Than .....	50
SGTU – Set if Greater Than Unsigned .....	51
SIGN – Sign.....	51
SLL –Shift Left Logical Pair .....	53
SLT – Set if Less Than .....	55
SLE – Set if Less Than or Equal.....	56
SLEU – Set if Less Than or Equal.....	56
SLTU – Set if Less Than Unsigned .....	58
SNE – Set if Not Equal .....	58
SRA –Shift Right Arithmetic Pair .....	60
SRL –Shift Right Logical Pair .....	61
SUB - Subtract.....	62
SUBF – Subtract From.....	63
U21NDX – UTF21 Index .....	64
WYDNDX – Wyde Index.....	65
XOR – Bitwise Exclusive Or.....	66
ZXB –Zero Extend Byte .....	67
ZXW –Zero Extend Wyde .....	67
ZXT –Zero Extend Tetra.....	68
Memory Operations .....	69
LDx – Load.....	69
LDB – Load Byte (8 bits) .....	73
LDBZ – Load Byte, Zero Extend (8 bits) .....	73
LDO – Load Octa (64 bits) .....	74
LDT – Load Tetra (32 bits).....	75
LDTZ – Load Tetra, Zero Extend (32 bits).....	75

LDW – Load Wyde (16 bits) .....	76
LDWZ – Load Wyde, Zero Extend (16 bits) .....	76
LEA – Load Effective Address.....	77
LSM – Load or Store Multiple.....	79
STx – Store .....	80
STB – Store Byte (8 bits).....	83
STBZ – Store Byte and Zero (8 bits) .....	83
STO – Store Octa (64 bits).....	84
STOZ – Store Octa and Zero (64 bits).....	84
STT – Store Tetra (32 bits) .....	86
STTZ – Store Tetra and Zero (32 bits).....	86
STW – Store Wyde (16 bits).....	86
STWZ – Store Wyde and Zero (16 bits) .....	86
Flow Control (Branch Unit) Operations .....	88
BEQ – Branch if Equal .....	88
BGE – Branch if Greater Than or Equal .....	90
BGEU – Branch if Greater Than or Equal Unsigned.....	91
BGT – Branch if Greater Than .....	92
BGTU – Branch if Greater Than Unsigned .....	93
BNE – Branch if Not Equal .....	94
BLE – Branch if Less Than or Equal .....	95
BLEU – Branch if Less Than or Equal Unsigned.....	95
BLT – Branch if Less Than.....	96
BLTU – Branch if Less Than Unsigned .....	96
BRA – Unconditional Branch.....	98
BRK – Break.....	105
CHK – Check Register Against Bounds .....	99
JAL – Jump and Link.....	101
JMP – Jump.....	103
PFI – Poll for Interrupt.....	103
RET – Return from Subroutine.....	104
REX – Redirect Exception.....	108
SYNC -Synchronize.....	108

Floating Point Instructions .....	105
Vector Specific Instructions.....	115
Arithmetic / Logical .....	115
V2BITS .....	115
VACC - Accumulate.....	<b>Error! Bookmark not defined.</b>
VBITS2V .....	116
VCIDX – Compress Index.....	117
VCMRSS – Compress Vector .....	118
VEINS / VMOVSV – Vector Element Insert .....	119
VEX / VMOVSV – Vector Element Extract .....	120
VSCAN .....	121
VSHLV – Shift Vector Left .....	122
VSHRV – Shift Vector Right.....	123
Memory Operations .....	124
CVLDx – Compressed Vector Load.....	124
CVSTx – Compressed Vector Store .....	126
Root Opcode Map .....	128
{SR3} Triadic Register Ops.....	129
{SR2} Dyadic Register Ops.....	129
{SR1} Monadic Register Ops .....	130

## Instruction Formats

Immediate Format:

31	20	19	14	13	8	7	0
Constant <sub>12</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		09h <sub>8</sub>	

Extended Immediate

31		8	7	0
Constant <sub>31..8</sub>		50 <sub>8</sub>		

31		8	7	0
Constant <sub>55..32</sub>		51 <sub>8</sub>		

31		8	7	0
Constant <sub>79..56</sub>		52 <sub>8</sub>		

Register Format:

SR1 (one source register)

31	26	25	20	19	14	13	8	7	0
0Ch <sub>6</sub>		Func <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		01h <sub>8</sub>	

SR2 (two source register)

31	26	25	20	19	14	13	8	7	0
Func <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		02h <sub>8</sub>	

58 Instruction Modifier

31	29	28	26	25	20	19	14	13	12	11	9	8	7	0
DT <sub>3</sub>		Rm <sub>3</sub>		Rd <sub>6</sub>		Rc <sub>6</sub>		A		m <sub>3</sub>		z		58h <sub>8</sub>

59 Bitfield Modifier

31	29	28	26	25	20	19	14	13	12	11	9	8	7	0
DT <sub>3</sub>		Rm <sub>3</sub>		Rd <sub>6</sub>		Rc <sub>6</sub>		A		m <sub>3</sub>		z		59h <sub>8</sub>

Branch

31	26	25	20	19	14	13	8	7	0
Constant <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Const <sub>6</sub>		48h <sub>8</sub>	

5A Branch Modifier

31	29	28	20	19	14	13	8	7	0
DT <sub>3</sub>		Constant <sub>9</sub>		Rc <sub>6</sub>		Rt <sub>6</sub>		5Ah <sub>8</sub>	

Instruction Modifier

31	29	28	26	25	14	13	12	11	9	8	7	0
C <sub>3</sub>		Rm <sub>3</sub>		Constant <sub>12</sub>		A		m <sub>3</sub>		z		5Bh <sub>8</sub>

## 5C Stride Modifier

31	21	20	19	14	13	12	11	9	8	7	0
Const <sub>11</sub>			I	Rc <sub>6</sub>	A		m <sub>3</sub>		z		5Ch <sub>8</sub>

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

m<sub>3</sub>: vector mask register (ignored for scalar operations).

Rm<sub>3</sub>: rounding mode

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

Rn<sub>8</sub>

0 to 63          scalar registers

64 to 127      vector registers

128 to 255    Rn is a seven-bit constant

U <sub>2</sub>	Execution Unit	Qualifier	
0	Integer	.int	
1	Floating-point	.fp	
2	Decimal floating-point	.dfp	
3	Posit	.pos	

Sz <sub>4</sub>	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.

### Integer Instruction Format: R1

Both the source and target registers are treated as integer values.

31 26	25 20	19 14	13 8	7 0
6 <sub>6</sub>	~ <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	01h <sub>8</sub>

### Integer Vector Format: R1

31 26	2524	23 21	20	19 14	13 8	7 0
06h <sub>6</sub>	~ <sub>2</sub>	m <sub>3</sub>	z	Va <sub>6</sub>	Vt <sub>6</sub>	81h <sub>8</sub>

### Float Instruction Format: R1

Both the source and target registers are treated as float values.

31 26	25 20	19 14	13 8	7 0
20h <sub>6</sub>	~ <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	34h <sub>8</sub>

### Float Vector Format: R1

31 26	2524	23 21	20	19 14	13 8	7 0
20h <sub>6</sub>	~ <sub>2</sub>	m <sub>3</sub>	z	Va <sub>6</sub>	Vt <sub>6</sub>	B4h <sub>8</sub>

### 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

**Clock Cycles:** 1

**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

$$Rt = Ra + Rb$$

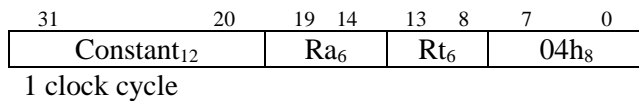
## Vector Operation

for  $x = 0$  to  $VL - 1$

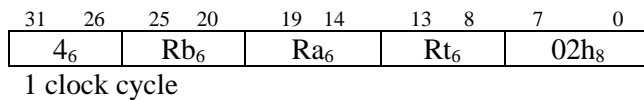
if ( $Vm[x]$ )  $Vt[x] = Va[x] + Vb[x]$

else if ( $z$ )  $Vt[x] = 0$

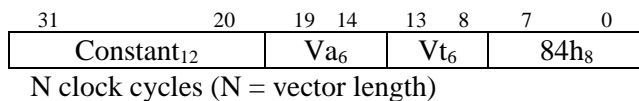
## Integer Instruction Format: RI



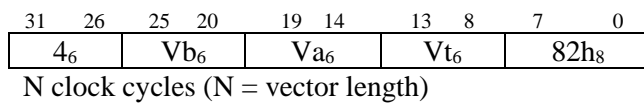
## Integer Instruction Format: R2



## Integer Vector Instruction Format: RI



## Integer Vector Instruction Format: R2



**Integer Vector Instruction Format: R2 (Add Scalar)**

31	26	25	20	19	14	13	8	7	0
04h <sub>6</sub>	Rb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	8Bh <sub>8</sub>					

N clock cycles (N = vector length)

**Float Instruction Format: R2**

31	26	25	20	19	14	13	8	7	0
4 <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	35h <sub>8</sub>					

25 clock cycles

**Float Vector Instruction Format: R2**

31	26	25	20	19	14	13	8	7	0
4 <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	B5h <sub>8</sub>					

N \* 25 clock cycles (N = vector length)

**Float Vector Instruction Format: R2 (Add Scalar)**

31	26	25	20	19	14	13	8	7	0
24h <sub>6</sub>	Rb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	B5h <sub>8</sub>					

N \* 25 clock cycles (N = vector length)

**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 or may be an immediate value specified in the instruction. The immediate constant is one extended before use.

## Instruction Format: RI

31	20	19 14	13 8	7	0
Constant <sub>12</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	08h <sub>8</sub>		

1 clock cycle

## Instruction Format: R2

31	26	25 20	19 14	13 8	7	0
8 <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	03h <sub>8</sub>		

1 clock cycle

## Vector Instruction Format: R2

31	26	25 20	19 14	13 8	7	0
8 <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	83h <sub>8</sub>		

N clock cycles ( N = vector length )

## Vector Instruction Format: R2 (Vector and Scalar)

31	26	25 20	19 14	13 8	7	0
30h <sub>6</sub>	Rb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	83h <sub>8</sub>		

N clock cycles ( N = vector length )

## Operation:

$R_t = R_a \& Imm$

or

$R_t = R_a \& R_b$

## Vector Operation

for  $x = 0$  to  $VL - 1$

if  $(V_m[x]) \ V_t[x] = V_a[x] \& V_b[x]$

else if  $(z) \ V_t[x] = 0$

**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	31	28	27	24	23	16	15	8	7	0
	Constant <sub>32</sub>		F <sub>4</sub>	Sh <sub>4</sub>		63 <sub>8</sub>			Rt <sub>8</sub>		Opcode <sub>8</sub>	

**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	13 12	11 9	8	7	0
DT <sub>3</sub>	0 <sub>3</sub>	0 <sub>6</sub>	Rc <sub>6</sub>	0	m <sub>3</sub>	z	F8h <sub>8</sub>	

31 26	25 20	19 14	13 8	7	0
30h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	03h <sub>8</sub>	

### Operation:

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

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

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

**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	49 48	47 44	43 41	40	39	32	31	24	23	16	15	8	7	0
Fn <sub>3</sub>	Rm <sub>3</sub>	03h <sub>8</sub>		U <sub>2</sub>	Sz <sub>4</sub>	m <sub>3</sub>	z	~ <sub>8</sub>		Rb <sub>8</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>		03h <sub>8</sub>	

Fn <sub>3</sub>	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

$$Rt.bit[i][j] = (Ra[i][0] \& Rb[0][j]) \mid (Ra[i][1] \& Rb[1][j]) \mid \dots \mid (Ra[i][15] \& Rb[15][j])$$

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Exceptions:** none

### Notes:

The bits are numbered with bit 63 of a register representing I<sub>j</sub> = 0,0 and bit 0 of the register representing I<sub>j</sub> = 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).

A second vector version of the instruction searches all the vector registers and returns a value which varies from -1 to +511 in a scalar register. Thus BYTNDX may be used to determine the length of a null termination string in the vector register.

## Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
0 <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	1Ah <sub>8</sub>

1 clock cycle

31 28	27 20	19 14	13 8	7 0
1	Imm <sub>8</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	1Ah <sub>8</sub>

1 clock cycle

## Vector Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
0 <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	9Ah <sub>8</sub>

N clock cycle (N = vector length)

31 28	27 20	19 14	13 8	7 0
1 <sub>4</sub>	Imm <sub>8</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	9Ah <sub>8</sub>

N clock cycle (N = vector length)

31 26	25 20	19 14	13 8	7 0
8 <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Rt <sub>6</sub>	9Ah <sub>8</sub>

N clock cycle (N = vector length)

31 28	27 20	19 14	13 8	7 0
3 <sub>4</sub>	Imm <sub>8</sub>	Va <sub>6</sub>	Rt <sub>6</sub>	9Ah <sub>8</sub>

N clock cycle (N = vector length)

## 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.

## Integer Instruction Format: R2

Both values are treated as signed numbers.

31 26	25 20	19 14	13 8	7 0
20h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>

1 clock cycle

## Integer Vector Instruction Format: R2

Both values are treated as signed numbers.

31 26	25 20	19 14	13 8	7 0
20h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	82h <sub>8</sub>

N clock cycles (N = vector length)

## Operation:

$Rt = Ra < Rb ? -1 : Ra = Rb ? 0 : 1$

## Vector Operation

for  $x = 0$  to  $VL - 1$

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

## Float Instruction Format: R2 (FCMP)

Both values are treated as double precision (64-bit) floating point numbers. The result is returned as a float value of -1.0, 0.0 or +1.0

31 26	25 20	19 14	13 8	7 0
10h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	35h <sub>8</sub>

1 clock cycle

## Float Vector Instruction Format:

31 26	25 20	19 14	13 8	7 0
10h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	B5h <sub>8</sub>

1 clock cycle

**Float Instruction Format: R2 (FCMPB)**

Both values are treated as double precision (64-bit) floating point numbers. The value returned is a bit vector as outlined in the table below. Note that the less than status is returned in both bits 1 and 63 so that a BLT may be used.

31	26	25	20	19	14	13	8	7	0
15h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	35h <sub>8</sub>					

1 clock cycle

The float comparison returns a bit vector containing the status of all possible relationships. This may then be tested with the BBS instruction.

Rt bit	Meaning
0	= equal
1	< less than
2	<= less than or equal
3	< magnitude less than
4	unordered
5 to 7	zero (reserved)
8	< > not equal
9	>= greater than or equal
10	> greater than
11	>= magnitude greater than or equal
12	ordered
13 to 62	zero (reserved)
63	less than

# 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 ( $Vm[x]$ )  $Vt[x] = \text{popcnt}(Va[x])$

## Instruction Format: R1

31 26	25 20	19 14	13 8	7 0
2h <sub>6</sub>	~ <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	01h <sub>8</sub>

## Vector Instruction Format: R1

31 26	25 20	19 14	13 8	7 0
2h <sub>6</sub>	~ <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	81h <sub>8</sub>

**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
0h <sub>6</sub>	~ <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	01h <sub>8</sub>

## Vector Instruction Format: R1

31 26	25 20	19 14	13 8	7 0
0h <sub>6</sub>	~ <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	81h <sub>8</sub>

**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
3 <sub>6</sub>	~ <sub>6</sub>			Ra <sub>6</sub>	Rt <sub>6</sub>		01h <sub>8</sub>		

1 clock cycle

## Vector Instruction Format: RI

31	26	25	20	19	14	13	8	7	0
3 <sub>6</sub>	~ <sub>6</sub>			Va <sub>6</sub>	Vt <sub>6</sub>		81h <sub>8</sub>		

N clock cycle (N = vector length)

## Operation

$$Rt = \sim Ra$$

OR

$$Rt = \sim Ra$$

## Vector Operation

for x = 0 to VL-1

$$\text{if } (Vm[x]) \text{ } Vt[x] = \sim Va[x]$$

$$\text{else if } (z) \text{ } Vt[x] = 0$$

$$\text{else } Vt[x] = Vt[x]$$

**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	13 12	11 9	8	7	0
DT <sub>3</sub>	Rm <sub>3</sub>	Rc <sub>6</sub>	Rd <sub>6</sub>	A	m <sub>3</sub>	z	F9h <sub>8</sub>	

31 26	25 20	19 14	13 8	7	0
3 <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	1Ch <sub>8</sub>	

DT <sub>3</sub>	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

31 26	25 20	19 14	13 8	7 0
18h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>

### Vector Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
18h <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	82h <sub>8</sub>

**Supported Formats:** .o

**Clock Cycles:** 1

**Execution Units:** Integer

**Operation:**

$$Rt = \text{Abs}(Ra - Rb)$$

**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
Constant <sub>12</sub>				Ra <sub>6</sub>		Rt <sub>6</sub>	
						10h <sub>8</sub>	

## Instruction Format: R2

31	26	25	20	19	14	13	8	7	0
10h <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		02h <sub>8</sub>	

## Vector Instruction Format: R2

31	26	25	20	19	14	13	8	7	0
10h <sub>6</sub>		Vb <sub>6</sub>		Va <sub>6</sub>		Vt <sub>6</sub>		82h <sub>8</sub>	

**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
	Constant <sub>32</sub>		T <sub>2</sub>	U <sub>2</sub>	Sz <sub>4</sub>	Ra <sub>8</sub>	Rt <sub>8</sub>	21h <sub>8</sub>	

### 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
Constant <sub>12</sub>		Ra <sub>6</sub>	Rt <sub>6</sub>	11h <sub>8</sub>	

### Instruction Format: R2

31	26	25 20	19 14	13 8	7	0
11h <sub>6</sub>		Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>	

### Execution Units: ALU

Clock Cycles: 67

Exceptions: none

## EXI0,EXI1,EXI2 – Extended Immediate

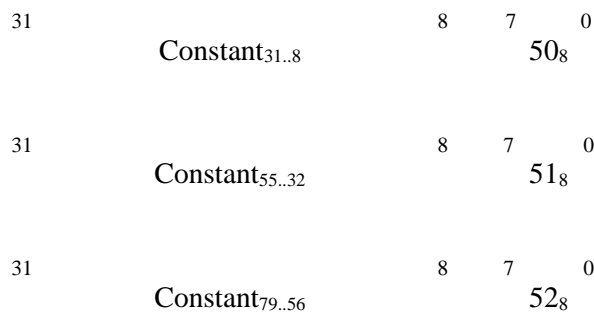
### Description:

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



## 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 13 12 11 9 8 7 0  
DT<sub>3</sub> Rm<sub>3</sub> Rd<sub>6</sub> Rc<sub>6</sub> A m<sub>3</sub> z F9h<sub>8</sub>

31 26 25 20 19 14 13 8 7 0  
4<sub>6</sub> Rb<sub>6</sub> Ra<sub>6</sub> Rt<sub>6</sub> 1Ch<sub>8</sub>

DT <sub>3</sub>	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

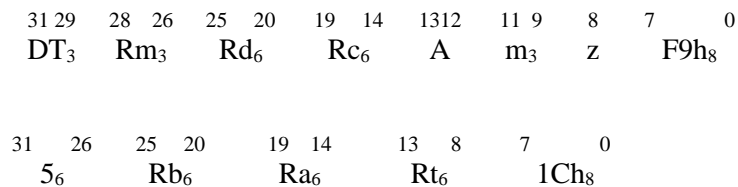
**Notes:**

## 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



DT <sub>3</sub>	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

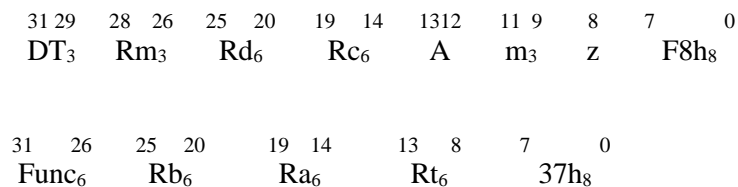
**Notes:**

## 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



## 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

$\begin{array}{cccccccccccc} 31 & 29 & 28 & 26 & 25 & 20 & 19 & 14 & 13 & 12 & 11 & 9 & 8 & 7 & 0 \\ \text{DT}_3 & & \text{Rm}_3 & & \text{Rd}_6 & & \text{Rc}_6 & & \text{A} & & \text{m}_3 & & \text{z} & & \text{F9h}_8 \end{array}$

$\begin{array}{cccccccc} 31 & 26 & 25 & 20 & 19 & 14 & 13 & 8 & 7 & 0 \\ 6_6 & & \text{Rb}_6 & & \text{Ra}_6 & & \text{Rt}_6 & & 1\text{Ch}_8 \end{array}$

DT <sub>3</sub>	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.

## Integer Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
29h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>

## Integer Vector Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
29h <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	82h <sub>8</sub>

## Float Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
03h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	35h <sub>8</sub>

## Float Vector Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
03h <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	B5h <sub>8</sub>

## 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.

## Integer Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
28h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>

## Integer Vector Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
28h <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	82h <sub>8</sub>

## Float Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
02h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	35h <sub>8</sub>

## Float Vector Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
02h <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	B5h <sub>8</sub>

## 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	13 12	11 9	8	7	0
DT <sub>3</sub>	Rm <sub>3</sub>	Rd <sub>6</sub>	Rc <sub>6</sub>	A	m <sub>3</sub>	z	5	8h <sub>8</sub>

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.

DT <sub>3</sub>	Interpretation: Instruction is operating on:
0	Integer
1	Floating point (double precision)
2	Decimal floating point
3	Posit (64-bit)
4-7	reserved

### Instruction Format: F9

31 29	28 26	25 20	19 14	13 12	11 9	8	7	0
DT <sub>3</sub>	Rm <sub>3</sub>	Rd <sub>6</sub>	Rc <sub>6</sub>	A	m <sub>3</sub>	z	5	9h <sub>8</sub>

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
DT <sub>3</sub>	Constant <sub>9</sub>	Rc <sub>6</sub>	Rt <sub>6</sub>	5Ah <sub>8</sub>		

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  $\pm 4\text{MB}$ .

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
Constant <sub>12</sub>				Ra <sub>6</sub>		Rt <sub>6</sub>	

4 clock cycles

## Instruction Format: R2

31	26	25	20	19	14	13	8	7	0
6 <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		02h <sub>8</sub>	

4 clock cycles

## Vector Instruction Format: RI

31	20	19	14	13	8	7	0
Constant <sub>12</sub>				Ra <sub>6</sub>		Rt <sub>6</sub>	

4 \* N clock cycles ( N = vector length)

## Vector Instruction Format: R2

31	26	25	20	19	14	13	8	7	0
6 <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		82h <sub>8</sub>	

4 \* N clock cycles ( N = vector length)

## 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
1Ch <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>
02h <sub>8</sub>			

1 clock cycle

## Instruction Format: RI

31 20	19 14	13 8	7 0
Constant <sub>12</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	15h <sub>8</sub>

1 clock cycle

## Vector Instruction Format: R2

31 26 25 20	19 14	13 8	7 0
1Ch <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>
82h <sub>8</sub>			

N clock cycles ( N = vector length)

## Instruction Format: RI

31 20	19 14	13 8	7 0
Constant <sub>12</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	95h <sub>8</sub>

N clock cycles ( N = vector length)

## 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
Constant <sub>12</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	0Eh <sub>8</sub>		

## Instruction Format: R2

31	26	25 20	19 14	13 8	7	0
Eh <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>		

## Vector Instruction Format: RI

31	20	19 14	13 8	7	0
Constant <sub>12</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	8Eh <sub>8</sub>		

## Vector Instruction Format: R2

31	26	25 20	19 14	13 8	7	0
Eh <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	82h <sub>8</sub>		

## 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	13 12	11 9	8	7	0
~3	~3	~6	Rc <sub>6</sub>	A	m <sub>3</sub>	z	F8h <sub>8</sub>	

31 26	25 20	19 14	13 8	7	0
04h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	03h <sub>8</sub>	

**Exceptions:** none

**Execution Units:** integer ALU

# NABS –Negative Absolute Value

## Description:

Take the negative absolute value of the number in register Ra and place the result into target register Rt. No rounding of the number occurs.

## Integer Instruction Format: R1

Both the source and target registers are treated as integer values.

31	26	25	20	19	14	13	8	7	0
7 <sub>6</sub>	~ <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	01h <sub>8</sub>					

## Integer Vector Format: R1

31	26	25	20	19	14	13	8	7	0
7 <sub>6</sub>	~ <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	81h <sub>8</sub>					

## Float Instruction Format: R1

Both the source and target registers are treated as float values.

31	26	25	20	19	14	13	8	7	0
21h <sub>6</sub>	~ <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	34h <sub>8</sub>					

## Float Vector Format: R1

31	26	25	20	19	14	13	8	7	0
21h <sub>6</sub>	~ <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	B4h <sub>8</sub>					

## Operation:

If  $Ra < 0$   
 $Rt = Ra$   
 else  
 $Rt = -Ra$

## Clock Cycles: 1

## Execution Units: Floating Point

# 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
0 <sub>12</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	05h <sub>8</sub>		

## Vector Instruction Format: R2

31	20	19 14	13 8	7	0
0 <sub>12</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	85h <sub>8</sub>		

## 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.

## Integer Instruction Format: R1

31	26	25	20	19	14	13	8	7	0
4 <sub>6</sub>	~ <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	01h <sub>8</sub>					

1 clock cycle

## Integer Vector Instruction Format: R1

31	26	25	24	23	21	20	19	14	13	8	7	0
04h <sub>6</sub>	~ <sub>2</sub>	m <sub>3</sub>	z	Va <sub>6</sub>	Vt <sub>6</sub>	81h <sub>8</sub>						

N clock cycles (N = vector length)

## 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
Constant <sub>12</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	09h <sub>8</sub>		

## Instruction Format: R2

31	26	25 20	19 14	13 8	7	0
9 <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	03h <sub>8</sub>		

## Vector Instruction Format: RI

31	20	19 14	13 8	7	0
Constant <sub>12</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	89h <sub>8</sub>		

## Vector Instruction Format: R2

31	26	25 20	19 14	13 8	7	0
9 <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	83h <sub>8</sub>		

## Vector Instruction Format: R2 (Vector or with Scalar)

31	26	25 20	19 14	13 8	7	0
31h <sub>6</sub>	Rb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	83h <sub>8</sub>		

## Operation

$Rt = Ra \mid \text{Immediate}$

OR

$Rt = Ra \mid Rb$

## 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: SR2, PERM

63 61	60 58	57	50	49 48	47 44	43 41	40	39	32	31	24	23	16	15	8	7	0
0 <sub>3</sub>	Rm <sub>3</sub>		~ <sub>8</sub>	0 <sub>2</sub>	Sz <sub>4</sub>	m <sub>3</sub>	Z		~ <sub>8</sub>	Rb <sub>8</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>			17h <sub>8</sub>

63 61	60 58	57	50	49 48	47 44	43 41	40	39		24	23	16	15	8	7	0
1 <sub>3</sub>	Rm <sub>3</sub>	Imm <sub>23..16</sub>		0 <sub>2</sub>	Sz <sub>4</sub>	m <sub>3</sub>	Z		Imm <sub>15..0</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>			17h <sub>8</sub>

**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
~	Rm <sub>3</sub>	30h <sub>8</sub>	Sz <sub>4</sub>	m <sub>3</sub>	Z	Rc <sub>8</sub>	Rb <sub>8</sub>	Ra <sub>8</sub>	Rt <sub>8</sub>	03h <sub>8</sub>						

### Operation:

$$Rt = \text{Abs}(Ra - Rb) \gg Rc$$

**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. Comparing float values returns an integer.

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
Constant <sub>12</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		26h <sub>8</sub>	

## Instruction Format: R2

### Integer:

31	26	25	20	19	14	13	8	7	0
26h <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		02h <sub>8</sub>	

### Float:

31	26	25	20	19	14	13	8	7	0
11h <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		35h <sub>8</sub>	

## Vector Instruction Format: RI

31	20	19	14	13	8	7	0
Constant <sub>12</sub>		Va <sub>6</sub>		Vt <sub>6</sub>		A6h <sub>8</sub>	

## Vector Instruction Format: R2

### Integer:

31	26	25	20	19	14	13	8	7	0
26h <sub>6</sub>		Vb <sub>6</sub>		Va <sub>6</sub>		Vt <sub>6</sub>		82h <sub>8</sub>	

### Float:

31	26	25	20	19	14	13	8	7	0
11h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	B5h <sub>8</sub>					

## 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

#### Integer:

31	26	25	20	19	14	13	8	7	0
2Dh <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		02h <sub>8</sub>	

#### Float:

The float version is an alternate mnemonic for [SLE](#) where the operands have been swapped.

31	26	25	20	19	14	13	8	7	0
13h <sub>6</sub>		Ra <sub>6</sub>		Rb <sub>6</sub>		Rt <sub>6</sub>		35h <sub>8</sub>	



## 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
2Fh <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		02h <sub>8</sub>	

## 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
Constant <sub>12</sub>		Ra <sub>6</sub>	Rt <sub>6</sub>		29h <sub>8</sub>

## 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
Constant <sub>12</sub>		Ra <sub>6</sub>	Rt <sub>6</sub>		2Bh <sub>8</sub>

## SIGN – Sign (Compare to Zero)

### Synopsis

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

### Description

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

### Instruction Format: RI

#### Integer:

31	26	25	20	19	14	13	8	7	0
2Ah <sub>6</sub>		0 <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		02h <sub>8</sub>	

#### Float:

31	26	25	20	19	14	13	8	7	0
10h <sub>6</sub>		0 <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		35h <sub>8</sub>	

### 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
19h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>

## Instruction Formats: R2I

31 26	25 20	19 14	13 8	7 0
1Ah <sub>6</sub>	Imm <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>

## Vector Instruction Formats: R2

31 26	25 20	19 14	13 8	7 0
19h <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	82h <sub>8</sub>

## Vector Instruction Formats: R2I

31 26	25 20	19 14	13 8	7 0
1Ah <sub>6</sub>	Imm <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	82h <sub>8</sub>

**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	13 12	11 9	8	7	0
DT <sub>3</sub>	Rm <sub>3</sub>	Rd <sub>6</sub>	Rc <sub>6</sub>	A	m <sub>3</sub>	z	F9h <sub>8</sub>	

31 26	25 20	19 14	13 8	7	0
10h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	03h <sub>8</sub>	

31 26	25 20	19 14	13 8	7	0
11h <sub>6</sub>	Imm <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	03h <sub>8</sub>	

**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.

## Integer Instruction Format: R2

31	26	25	20	19	14	13	8	7	0
2Ch <sub>6</sub>	Ra <sub>6</sub>	Rb <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>					

## Integer Vector Instruction Format: R2

31	26	25	20	19	14	13	8	7	0
2Ch <sub>6</sub>	Ra <sub>6</sub>	Rb <sub>6</sub>	Rt <sub>6</sub>	82h <sub>8</sub>					

## Float Instruction Format: R2

31	26	25	20	19	14	13	8	7	0
12h <sub>6</sub>	Ra <sub>6</sub>	Rb <sub>6</sub>	Rt <sub>6</sub>	35h <sub>8</sub>					

## Float Vector Instruction Format: R2

31	26	25	20	19	14	13	8	7	0
12h <sub>6</sub>	Ra <sub>6</sub>	Rb <sub>6</sub>	Rt <sub>6</sub>	B5h <sub>8</sub>					

## 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.

### Instruction Format: R2

#### Integer:

The integer register form of instruction is an alternate mnemonic for [SGE](#) where the operands have been swapped.

31 26	25 20	19 14	13 8	7 0
2Dh <sub>6</sub>	Ra <sub>6</sub>	Rb <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>

#### Float:

31 26	25 20	19 14	13 8	7 0
13h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	35h <sub>8</sub>

#### Integer Vector:

The integer register form of instruction is an alternate mnemonic for [SGE](#) where the operands have been swapped.

31 26	25 20	19 14	13 8	7 0
2Dh <sub>6</sub>	Va <sub>6</sub>	Vb <sub>6</sub>	Vt <sub>6</sub>	82h <sub>8</sub>

#### Float Vector:

31 26	25 20	19 14	13 8	7 0
13h <sub>6</sub>	Vb <sub>6</sub>	Va <sub>6</sub>	Vt <sub>6</sub>	B5h <sub>8</sub>

## 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	43 41	40	39	32	31	24	23	16	15	8	7	0
~	Rm <sub>3</sub>	0Ch <sub>8</sub>	Sz <sub>4</sub>	m <sub>3</sub>	z	2Eh <sub>8</sub>	Rb <sub>8</sub>	Ra <sub>8</sub>	Rt <sub>8</sub>	03h <sub>8</sub>						

## 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
	Constant <sub>12</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>				27h <sub>8</sub>

### Instruction Format: R2

31	26	25	20	19	14	13	8	7	0
27h <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>			02h <sub>8</sub>

# SRA –Shift Right Arithmetic Pair

## Description:

This is an alternate mnemonic for the signed field extract [EXT](#) 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 13 12 11 9 8 7 0  
DT<sub>3</sub> Rm<sub>3</sub> 63<sub>6</sub> Rc<sub>6</sub> A m<sub>3</sub> z F9h<sub>8</sub>

31 26 25 20 19 14 13 8 7 0  
4<sub>6</sub> Rb<sub>6</sub> Ra<sub>6</sub> Rt<sub>6</sub> 1Ch<sub>8</sub>

DT <sub>3</sub>	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

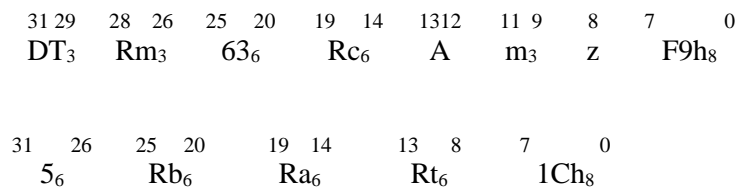
## Description:

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

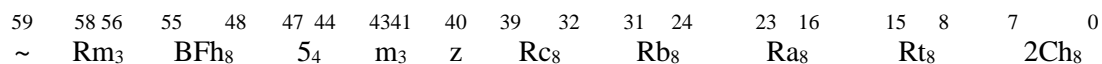
This instruction may also be used to perform a right rotate of a single register by specifying the same register for Ra and Rb.

## Instruction Format: R4



DT <sub>3</sub>	Meaning
10	Rd is a six bit immediate, Rc is a reg
11	Both Rc, Rd are six bit immediates

## Instruction Format: SR4



**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
5 <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	02h <sub>8</sub>

## 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
Constant <sub>12</sub>		Ra <sub>6</sub>	Rt <sub>6</sub>	05h <sub>8</sub>	

### 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
$0_6$		$Rb_6$		$Ra_6$		$Rt_6$		$23h_8$	

### Supported Formats: .o

### Clock Cycles: 1

### Execution Units: Integer ALU

### Operation:

$Rt = \text{Index of } (Rb \text{ in } Ra)$

### 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
$0_6$		$Rb_6$		$Ra_6$		$Rt_6$		$1Bh_8$	

### R2 Supported Formats: .o

### Clock Cycles: 1

### Execution Units: Integer ALU

### Operation:

$$Rt = \text{Index of } (Rb \text{ in } Ra)$$

### 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
Constant <sub>12</sub>		Ra <sub>6</sub>	Rt <sub>6</sub>	0Ah <sub>8</sub>

## Instruction Format: R2

31 26	25 20	19 14	13 8	7 0
Ah <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	03h <sub>8</sub>

## Operation

$$Rt = Ra \wedge \text{Immediate}$$

OR

$$Rt = Ra \wedge Rb$$

## Vector Operation

for  $x = 0$  to  $VL-1$

if  $(Vm[x]) \quad Vt[x] = Va[x] \wedge Vb[x] \wedge Vc[x]$

else if  $(z) \quad Vt[x] = 0$

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

$31 \quad 28 \quad 27 \quad 20 \quad 19 \quad 14 \quad 13 \quad 11 \quad 10 \quad 8 \quad 7 \quad 0$   
 $15_{3..0} \quad \text{Const}_8 \quad \text{Ra}_6 \quad \text{DC}_3 \quad \text{IC}_3 \quad 60h_8$

#### Commands:

IC <sub>3</sub>	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

DC <sub>3</sub>	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
Func <sub>3..0</sub>		Const <sub>8</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		60h <sub>8</sub>	

### 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
Func <sub>3..0</sub>		0	S	Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		61h <sub>8</sub>	

z: 1= zero extend, 0 = sign extend

Sc <sub>3</sub>	Multiplier
0	1
1	2
2	4
3	8
4	16

### Operation:

$Rt = \text{Memory}[d + Ra + Rb * Sc]$

### Vector forms

### Stridden Form (LDS)

31	21	20	19	14	13	12	11	9	8	7	0
Const <sub>11</sub>			I	Rc <sub>6</sub>		A	m <sub>3</sub>		z	5Ch <sub>8</sub>	

31	28	27	20	19	14	13	8	7	0
Func <sub>3..0</sub>		Const <sub>8</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		E2h <sub>8</sub>	

Data is loaded from memory addresses separated by the stride amount specified by register field Rc, 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

U <sub>2</sub>	Unit
0	integer
1	floating-point
2	decimal-float
3	posit

Sz <sub>4</sub>	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 Vc.

31	21	20	19	14	13	12	11	9	8	7	0
Const <sub>11</sub>		0	Vc <sub>6</sub>	A			m <sub>3</sub>		z		DCh <sub>8</sub>

31	28	27	20	19	14	13	8	7	0
Func <sub>3..0</sub>		Const <sub>8</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		E3h <sub>8</sub>	

***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:

$$Rd = \text{Memory}_8[d + Ra + Rb * Sc]$$

**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:

$$Rd = \text{Memory}_8[d + Ra + Rb * Sc]$$

**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:

$$Rt = \text{Memory}_{64}[d + Ra + Rb * Sc]$$

**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:

$$Rt = \text{Memory}_{32}[d + Ra + Rb * Sc]$$

**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:

$$Rt = \text{Memory}_{32}[d + Ra + Rb * Sc]$$

**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:

$$Rt = \text{Memory}_{16}[d + Ra + Rb * Sc]$$

**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:

$$Rt = \text{Memory}_{16}[d + Ra + Rb * Sc]$$

**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	
Func <sub>3..0</sub>				1	S	Rb <sub>6</sub>		Ra <sub>6</sub>		Rt <sub>6</sub>		61h <sub>8</sub>

### Operation:

$$Rt = d + Ra + Rb * Sc$$

### Vector forms

#### Stridden Form (LDS)

63		50	49	48	47	44	43	41	40	39	32	31	24	23	16	15	8	7	0	
Const <sub>21..8</sub>				U <sub>2</sub>		Sz <sub>4</sub>		m <sub>3</sub>		z		Const <sub>7..0</sub>		Rb <sub>8</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>		69h <sub>8</sub>

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

U <sub>2</sub>	Unit
0	integer
1	floating-point
2	decimal-float
3	posit

Sz <sub>4</sub>	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	43 41	40	39	32	31	24	23	16	15	8	7	0
	Const <sub>23..8</sub>		Sz <sub>4</sub>	m <sub>3</sub>	z	Const <sub>7..0</sub>	Vb <sub>8</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>		6Ah <sub>8</sub>		

**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
Rg <sub>8</sub>		Rf <sub>8</sub>		Re <sub>8</sub>		Rd <sub>8</sub>		Rc <sub>8</sub>		Rb <sub>8</sub>		Ra <sub>8</sub>		6Fh <sub>8</sub>	

**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	27	26	25	20	19	14	13	8	7	0
Func <sub>3..0</sub>		C <sub>2</sub>	Rb <sub>6</sub>		Ra <sub>6</sub>		Const <sub>6</sub>		70h <sub>8</sub>		

### 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	13	12	11	9	8	7	0
DT <sub>3</sub>		Rm <sub>3</sub>		Rd <sub>6</sub>		Rc <sub>6</sub>		A		m <sub>3</sub>		z	F8h <sub>8</sub>	

31	28	27	26	25	20	19	14	13	8	7	0
Func <sub>3..0</sub>		~	S	Rb <sub>6</sub>		Ra <sub>6</sub>		~ <sub>6</sub>		71h <sub>8</sub>	

Sc	Multiplier
0	1
1	Store size

### Operation:

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

## Vector forms

### Stridden Form (STS)

31	21	20	19	14	13	12	11	9	8	7	0
Const <sub>11</sub>		I	Rc <sub>6</sub>		A		m <sub>3</sub>		z	5Ch <sub>8</sub>	

31	28	27	26	25	20	19	14	13	8	7	0
Func <sub>3..0</sub>		C <sub>2</sub>	Rb <sub>6</sub>		Ra <sub>6</sub>		Const <sub>6</sub>		F2h <sub>8</sub>		

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]

U <sub>2</sub>	Unit
0	integer
1	floating-point
2	decimal-float
3	posit

Sz <sub>4</sub>	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.

31	21	20	19	14	1312	11	9	8	7	0
Const <sub>11</sub>		0	Vc <sub>6</sub>	A		m <sub>3</sub>		z	DCh <sub>8</sub>	

31	28	2726	25	20	19	14	13	8	7	0
Func <sub>3..0</sub>	C <sub>2</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Const <sub>6</sub>		F3h <sub>8</sub>	

***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:

$$\text{Memory}_8[d + \text{Ra} + \text{Rb} * \text{Sc}] = \text{Rs}$$

## 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:

$$\begin{aligned} \text{Memory}_8[d + \text{Ra} + \text{Rb} * \text{Sc}] &= \text{Rs} \\ \text{Rs} &= 0 \end{aligned}$$

## 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:

$$\text{Memory}_{64}[\text{d} + \text{Ra} + \text{Rb} * \text{Sc}] = \text{Rs}$$

## 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 octa is stored to memory the register is zeroed out.

### Instruction Format: ST

### Operation:

$$\text{Memory}_{64}[\text{d} + \text{Ra} + \text{Rb} * \text{Sc}] = \text{Rs}$$

$$\text{Rs} = 0$$

## 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:

$$\text{Memory}_{32}[\text{d} + \text{Ra} + \text{Rb} * \text{Sc}] = \text{Rs}$$

## 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:

$$\text{Memory}_{32}[\text{d} + \text{Ra} + \text{Rb} * \text{Sc}] = \text{Rs}$$

$$\text{Rs} = 0$$

## 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:

$$\text{Memory}_{16}[\text{d} + \text{Ra} + \text{Rb} * \text{Sc}] = \text{Rs}$$

## 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:**

$$\text{Memory}_{16}[\text{d} + \text{Ra} + \text{Rb} * \text{Sc}] = \text{Rs}$$

$$\text{Rs} = 0$$

## 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  $\pm 32\text{MB}$  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  $\pm 8\text{MB}$ . 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
Constant <sub>22</sub>		Rt <sub>2</sub>		41h <sub>8</sub>	

**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
Constant <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Const <sub>6</sub>	4Eh <sub>8</sub>

### Operation:

If (Ra = Rb)

IP = IP + Displacement<sub>14</sub>

With Modifier

Rt = IP + 4

If (Ra = Rb)

IP = Rc + Displacement<sub>23</sub>

**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
Constant <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Const <sub>6</sub>		49h <sub>8</sub>	

### 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
Constant <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Const <sub>6</sub>		4Bh <sub>8</sub>	

### Operation:

$$Rt = IP + 8$$

If (Ra >= Rb)

$$PC = Rc + \text{Displacement}$$

### Execution Units: Branch

### Exceptions: none

## BGT – Branch if Greater Than

### Description:

This instruction is an alternate mnemonic for the [BLT](#) 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
Constant <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Const <sub>6</sub>	48h <sub>8</sub>

### Operation:

$$Rt = IP + 8$$

If (Ra < Rb)

$$PC = Rc + \text{Displacement}$$

### Execution Units: Branch

### Exceptions: none

# BGTU – Branch if Greater Than Unsigned

## Description:

This instruction is an alternate mnemonic for the [BLTU](#) 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
Constant <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	Const <sub>6</sub>	4Ah <sub>8</sub>

## Operation:

$$Rt = IP + 8$$

If (Ra < Rb)

$$PC = Rc + \text{Displacement}$$

## 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	49	48	47	44	43	40	39	32	31	24	23	16	15	8	7	0
Displacement <sub>20..7</sub>		U <sub>2</sub>		SZ <sub>4</sub>		Disp <sub>6..3</sub>		Rc <sub>8</sub>		Rb <sub>8</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>		4Fh <sub>8</sub>		

## Operation:

$$Rt = IP + 8$$

If (Ra <> Rb)

$$PC = Rc + \text{Displacement}$$

## 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  $\geq$  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  $\geq$  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	49	48	47	44	43	40	39	32	31	24	23	16	15	8	7	0
Displacement <sub>20..7</sub>				U <sub>2</sub>	Sz <sub>4</sub>	Disp <sub>6..3</sub>		Rc <sub>8</sub>		Rb <sub>8</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>		48h <sub>8</sub>	

### Operation:

$$Rt = IP + 8$$

If (Ra < Rb)

$$PC = Rc + \text{Displacement}$$

### 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	49	48	47	44	43	40	39	32	31	24	23	16	15	8	7	0
Displacement <sub>20..7</sub>				U <sub>2</sub>	Sz <sub>4</sub>	Disp <sub>6..3</sub>		Rc <sub>8</sub>		Rb <sub>8</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>		4Ah <sub>8</sub>	

### Operation:

$$Rt = IP + 8$$

If (Ra < Rb)

$$PC = Rc + \text{Displacement}$$

### Execution Units: Branch

Exceptions: none





## BRA – Unconditional Branch

### Description:

This instruction is an alternate mnemonic for the [BAL](#) instruction. The address range is 24 bits or  $\pm 8\text{MB}$ . The constant field is shifted left twice before use.

### Formats Supported: JAL

31		10		9	8		7		0
Constant <sub>22</sub>			0 <sub>2</sub>		41h <sub>8</sub>				

**Flags Affected:** none

### Operation:

$\text{IP} = \text{IP} + \text{Displacement}$

**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	13 12	11 9	8	7	0
DT <sub>3</sub>	Rm <sub>3</sub>	Rd <sub>6</sub>	Rc <sub>6</sub>	A	m <sub>3</sub>	z	F7h <sub>8</sub>	

31	20	19 14	13 10	9 8	7	0
Constant <sub>12</sub>		Ra <sub>6</sub>	~	Cn <sub>2</sub>	22h <sub>8</sub>	

Cn <sub>2</sub>	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	13 12	11 9	8	7	0
DT <sub>3</sub>	Rm <sub>3</sub>	Rd <sub>6</sub>	Rc <sub>6</sub>	A	m <sub>3</sub>	z	F7h <sub>8</sub>	

31	26	25 20	19 14	13 10	9 8	7	0
22h <sub>6</sub>		Rb <sub>6</sub>	Ra <sub>6</sub>	~	Cn <sub>2</sub>	03h <sub>8</sub>	

Cn <sub>2</sub>	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  $\pm 8\text{MB}$ . 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
Constant <sub>22</sub>		Rt <sub>2</sub>		40h <sub>8</sub>	

**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
Constant <sub>12</sub>		Ra <sub>6</sub>	Cnst <sub>4</sub>	Rt <sub>2</sub>		42h <sub>8</sub>

**Flags Affected:** none

## Operation:

$Rt = IP + Cnst_4 * 2$

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](#) instruction. It may be used to jump directly to a specific address. The address range is 24 bits or  $\pm 8\text{MB}$ . 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
Constant <sub>22</sub>		0 <sub>2</sub>		40h <sub>8</sub>	

**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](#) 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
Constant <sub>12</sub>		01 <sub>6</sub>	0 <sub>4</sub>	0 <sub>2</sub>		42h <sub>8</sub>

**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
~ <sub>6</sub>		~ <sub>4</sub>		Cause <sub>8</sub>		0 <sub>6</sub>		00h <sub>8</sub>	

#### Operation:

$$\text{PMSTACK} = (\text{PMSTACK} \ll 4) \mid 10$$

$$\text{CAUSE} = \text{Const}_8$$

$$\text{EIP} = \text{IP}$$

$$\text{IP} = \text{tvec}[4]$$

#### Execution Units: Branch

#### Clock Cycles:

#### Exceptions: none

#### Notes:

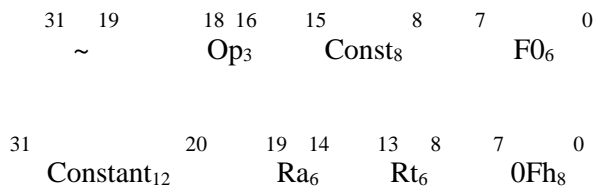
## CSRx – Control and Special / Status Access

### 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



Op <sub>3</sub>	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<sub>[15..12]</sub> 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
11h <sub>6</sub>		0 <sub>6</sub>		0 <sub>6</sub>		0 <sub>6</sub>		44h <sub>8</sub>	

**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	43 41	40	39	32	31 24	23 16	15 8	7	0
~	Rm <sub>3</sub>	7Ah <sub>8</sub>	Tm <sub>3</sub>	m <sub>3</sub>	z	Rc <sub>8</sub>	Imm <sub>8</sub>	Ra <sub>8</sub>	0 <sub>8</sub>	44h <sub>8</sub>			

Tm<sub>3</sub>

- 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:

```

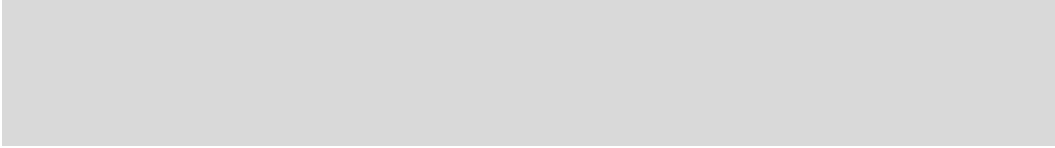
REX 1          ; redirect to supervisor handler

; 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
13h <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Opt <sub>6</sub>		44h <sub>8</sub>	
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
$\sim_3$	Op <sub>3</sub>	??h <sub>8</sub>	U <sub>2</sub>	Sz <sub>4</sub>	m <sub>3</sub>	z	Rc <sub>8</sub>	Rb <sub>8</sub>	Ra <sub>8</sub>	Rt <sub>8</sub>	44h <sub>8</sub>	





## 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  
1Eh<sub>6</sub> Rb<sub>6</sub> Ra<sub>6</sub> Rt<sub>6</sub> 44h<sub>8</sub>

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
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
12h <sub>6</sub>		0 <sub>6</sub>		0 <sub>6</sub>		0 <sub>6</sub>		44h <sub>8</sub>	

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

**Execution Units:** Branch

## Vector Specific Instructions

### Arithmetic / Logical

#### V2BITS

##### Description

Convert Boolean vector to bits. 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	2524	23 21	20	19 14	13 8	7	0
18h <sub>6</sub>	~ <sub>2</sub>	m <sub>3</sub>	z	Va <sub>6</sub>	Rt <sub>6</sub>	81h <sub>8</sub>		

##### Operation

For x = 0 to VL-1

if (Vm[x])

$Rt[x] = Va[x].LSB$

else if (z)

$Rt[x] = 0$

**Exceptions:** none

# VBITS2V

## Description

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

## Instruction Format: R1

31	26	25	24	23	21	20	19	14	13	8	7	0
19h <sub>6</sub>	~ <sub>2</sub>	m <sub>3</sub>	z	Ra <sub>6</sub>	Vt <sub>6</sub>	81h <sub>8</sub>						

## Operation

For  $x = 0$  to  $VL-1$

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

**Exceptions:** none

# VCIDX – Compress Index

## 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.

## Instruction Format: R1

31 26	25 24	23 21	20	19 14	13 8	7 0
2Dh <sub>6</sub>	~ <sub>2</sub>	m <sub>3</sub>	z	Ra <sub>6</sub>	Vt <sub>6</sub>	8lh <sub>8</sub>

## Operation

$y = 0$

for  $x = 0$  to  $VL - 1$

if (Vm[x])

$Vt[y] = Ra * x$

$y = y + 1$

# VCMPRSS – Compress Vector

## Description

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

## Instruction Format: R1

31	26	2524	23 21	20	19 14	13 8	7	0
2Ch <sub>6</sub>	~ <sub>2</sub>	m <sub>3</sub>	z	Va <sub>6</sub>	Vt <sub>6</sub>	8lh <sub>8</sub>		

## 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  $V_t$  are set to the cumulative sum of a value in register  $R_a$ . The summation is guided by a vector mask register.

## Operation

sum = 0

for x = 0 to VL - 1

$V_t[x] = \text{sum}$

if ( $V_m[x]$ )

sum = sum +  $R_a$

## VSLLV – Shift Vector Left Logical

### 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

# VSRLV – Shift Vector Right Logical

## 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	49	48	47	44	43	41	40	39	32	31	24	23	16	15	8	7	0
	Const <sub>21..8</sub>		U <sub>2</sub>		Sz <sub>4</sub>		m <sub>3</sub>		z		Const <sub>7..0</sub>		Rb <sub>8</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>		65h <sub>8</sub>

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 Sz<sub>4</sub> 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
	Const <sub>21..8</sub>		U <sub>2</sub>	Sz <sub>4</sub>	m <sub>3</sub>	z	Const <sub>7..0</sub>		Rb <sub>8</sub>		Ra <sub>8</sub>		Rt <sub>8</sub>			66h <sub>8</sub>

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
Const <sub>6</sub>	U <sub>2</sub>	Sz <sub>4</sub>	m <sub>3</sub>	z		Constant <sub>12</sub>			Ra <sub>6</sub>	Vs <sub>6</sub>	1		74h <sub>7</sub>

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
Const <sub>6</sub>	U <sub>2</sub>	Sz <sub>4</sub>	m <sub>3</sub>	z		Const <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		Vs <sub>6</sub>		1		75h <sub>7</sub>

### 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
Const <sub>6</sub>	U <sub>2</sub>	Sz <sub>4</sub>	m <sub>3</sub>	z	Const <sub>6</sub>	Vb <sub>6</sub>	Ra <sub>6</sub>	Vs <sub>6</sub>	1	76h <sub>7</sub>						

**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
<b>ALU</b>								
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	{DF1}	{DF2}	{DF3}	{DF4}	{F1}	{F2}	{F3}	{F4}
00111	{PST1}	{PST2}	{PST3}	{PST4}				NOP
<b>Branch Unit</b>								
01000	JAL	BAL	JALR		{SYS}			
01001	BLT	BGE	BLTU	BGEU		BBS	BEQ	BNE
<b>Instruction Modifiers (Prefixes)</b>								
01010	EXI	EXI	EXI					
01011	IMOD	BTFLD	BRMOD	FB	STRIDE			
<b>Memory Unit</b>								
01100	LDx	LDxX					CVLDS	CVLDVX
01101								LSM
01110	STx	STxX					CVSTS	CVSTVX
01111								
<b>Vector ALU</b>								
10000		{R1}	{R2}	{R3}	ADD	SUBF	MUL	
10001	AND	OR	XOR	{R2S}			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	{DF1}	{DF2}	{DF3}	{DF4}	{F1}	{F2}	{F3}	{F4}
10111	{PST1}	{PST2}	{PST3}	{PST4}				NOP
11000								
11001								
11010								
11011	IMOD	BTFLD	BRMOD	FB				
11100			LDSx	LDxVX				
11101								
11110			STSx	STxVX				
11111								



## {R3} 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								

## {R2} 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	SQRT	MULF	MULSUH	MULUH	RGF
100	CMP						SEQ	SNE
101	MIN	MAX			SLT	SGE	SLTU	SGEU
110								
111	VSLLV	VSLRV	VEX	VEINS				

## {R2S} Dyadic Register Ops (Vector-Scalar)

	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	SQRT	MULF	MULSUH	MULUH	RGF
100	CMP						SEQ	SNE
101	MIN	MAX			SLT	SGE	SLTU	SGEU
110								
111	VSLLV	VSLRV	VEX	VEINS				

## {R1} Monadic Register Ops

	000	001	010	011	100	101	110	111
000	CNTLZ	CNTLO	CNTPOP	COM	NOT	NEG	ABS	NABS
001				TST				
010	PTRINC							
011	V2BITS	BITS2V			VCMPRSS	VCIDX	VSCAN	
100								
101								
110								
111								

## {OSR2} System Ops

	000	001	010	011	100	101	110	111
000	LLAL	LLAH			LPAL	LPAH		
001	PUSHQ	POPQ	PEEKQ	STATQ	SETKEY			
010	REX	PFI	WAI	RTE				
011	SETTO	GETTO	GETZL			MVSEG	TLBRW	SYNC
100								
101								
110								
111								

## Floating-Point Monadic Ops – {FLT1} Funct<sub>5</sub>

	000	001	010	011	100	101	110	111
000	FMOV	FRSQRT	FTOI	ITOF			FSIGN	FMAN
001		FS2D	FS2Q	FD2Q	FSTAT	FSQRT	ISNAN	FINITE
010	FTX	FCX	FEX	FDX	FRM	TRUNC	FSYNC	FRES
011	FSIGMOID	FD2S	FQ2S	FQ2D			FCLASS	UNORD
100	FABS	FNABS	FNEG					
101								
110								
111								

## Floating-Point Dyadic Ops – {FLT2} Funct<sub>5</sub>

	000	001	010	011	100	101	110	111
Vector-Vector or Scalar-Scalar								
000	SCALEB		FMIN	FMAX	FADD	FSUB		
001	FMUL	FDIV	FREM	FNXT	FAND	FOR		
010	FCMP	FSEQ	FSLT	FSLE	FSNE	FCMPB		
011	CPYSGN	SGNINV	SGNAND	SGNOR	SGNXOR	SGNXNOR	FCLASS	
Vector-Scalar								
100	SCALEB		FMIN	FMAX	FADD	FSUB		
101	FMUL	FDIV	FREM	FNXT	FAND	FOR		
110	FCMP	FSEQ	FSLT	FSLE	FSNE	FCMPB		
111	CPYSGN	SGNINV	SGNAND	SGNOR	SGNXOR	SGNXNOR	FCLASS	