

# ANY-1 v3 Instruction Set

© 2021 Robert Finch

## Table of Contents

Instruction Formats .....	7
Register Specifiers .....	7
Constant Interpretation for Float Instructions .....	7
Vector Instruction Indicator .....	8
Root Opcode .....	8
Extended Immediate .....	8
Register Formats .....	9
R1 (one source register) .....	9
R2 (two source register) .....	9
Branch Instructions .....	10
Instruction Modifiers .....	10
IMOD Instruction Modifier - 58 .....	10
Branch Modifier – 5A .....	10
Stride Modifier – 5C .....	10
Example Instruction .....	11
Instructions.....	12
Arithmetic / Logical .....	12
ABS – Absolute Value.....	12
ADD - Addition .....	14
AND – Bitwise And.....	16
BMM – Bit Matrix Multiply .....	17
BYTNDX – Byte Index .....	18
CMP – Compare .....	19
CNTPOP – Count Population .....	21
CNTLZ – Count Leading Zeros.....	22
COM – Ones Complement.....	23
DEP – Deposit.....	24
DIF – Difference .....	25
DIV – Division.....	26

DIVR – Division .....	27
DIVSU – Division Signed-Unsigned .....	28
DIVU – Division Unsigned .....	29
EOR – Bitwise Exclusive Or .....	29
EXIn – Extended Immediate .....	30
EXT –Extract Bitfield .....	31
EXTU –Extract Bitfield Unsigned .....	32
FDP – Fused Dot Product .....	32
FFO –Find First One .....	33
MAX – Maximum Value .....	34
MIN – Minimum Value .....	35
MOD – Instruction Modifier .....	36
MUL – Multiply .....	38
MULF – Fast Unsigned Multiply .....	39
MULU – Unsigned Multiply .....	40
MUX – Multiplex .....	41
NABS –Negative Absolute Value .....	42
NEG - Negate .....	43
NOT – Logical Not .....	44
OR – Bitwise Or .....	45
PERM – Permute Bytes .....	46
PTRDIF – Difference Between Pointers .....	47
SEQ – Set if Equal .....	48
SGE – Set if Greater Than or Equal .....	50
SGEU – Set if Greater Than or Equal Unsigned .....	51
SGT – Set if Greater Than .....	52
SGTU – Set if Greater Than Unsigned .....	53
SIGN – Sign (Compare to Zero) .....	54
SLL –Shift Left Logical .....	55
SLLP –Shift Left Logical Pair .....	56
SLT – Set if Less Than .....	57
SLE – Set if Less Than or Equal .....	58
SLEU – Set if Less Than or Equal .....	58

SLTU – Set if Less Than Unsigned .....	60
SNE – Set if Not Equal .....	61
SQRT – Square Root .....	62
SRA –Shift Right Arithmetic Pair .....	63
SRL –Shift Right Logical .....	64
SRLP –Shift Right Logical Pair.....	65
SUB - Subtract .....	66
SUBF – Subtract From.....	67
U21NDX – UTF21 Index .....	70
WYDNDX – Wyde Index.....	71
XOR – Bitwise Exclusive Or .....	72
ZXB –Zero Extend Byte .....	73
ZXW –Zero Extend Wyde .....	73
ZXT –Zero Extend Tetra.....	74
Graphics .....	75
BLEND – Blend Colors .....	75
TRANSFORM – Transform Point.....	76
RW_COEEF – Read/Write Co-efficient.....	77
Memory Operations .....	78
CACHE – Cache Command .....	78
LDx – Load .....	79
LDB – Load Byte (8 bits) .....	82
LDBZ – Load Byte, Zero Extend (8 bits) .....	82
LDO – Load Octa (64 bits) .....	83
LDT – Load Tetra (32 bits).....	84
LDTZ – Load Tetra, Zero Extend (32 bits).....	84
LDW – Load Wyde (16 bits) .....	85
LDWZ – Load Wyde, Zero Extend (16 bits) .....	85
LEA – Load Effective Address.....	86
STx – Store .....	88
STB – Store Byte (8 bits).....	91
STBZ – Store Byte and Zero (8 bits) .....	91
STO – Store Octa (64 bits).....	92

STOZ – Store Octa and Zero (64 bits).....	92
STPTR – Store Pointer (64 bits) .....	93
STT – Store Tetra (32 bits) .....	94
STTZ – Store Tetra and Zero (32 bits).....	94
STW – Store Wyde (16 bits).....	94
STWZ – Store Wyde and Zero (16 bits) .....	94
Flow Control (Branch Unit) Operations .....	96
Branches.....	96
BAL – Branch and Link.....	96
BBS – Branch if Bit Set.....	97
BEQ – Branch if Equal .....	98
BGE – Branch if Greater Than or Equal .....	99
BGEU – Branch if Greater Than or Equal Unsigned.....	100
BGT – Branch if Greater Than .....	101
BGTU – Branch if Greater Than Unsigned .....	102
BNE – Branch if Not Equal .....	103
BLE – Branch if Less Than or Equal .....	104
BLEU – Branch if Less Than or Equal Unsigned.....	105
BLT – Branch if Less Than.....	106
BLTU – Branch if Less Than Unsigned .....	107
BRA – Unconditional Branch.....	108
BSR – Unconditional Branch to Subroutine .....	108
CHK – Check Register Against Bounds .....	110
JAL – Jump and Link.....	112
JALR – Jump and Link to Register.....	113
JMP – Jump.....	114
RET – Return from Subroutine.....	<b>Error! Bookmark not defined.</b>
System Instructions.....	116
BRK – Break.....	116
CSRx – Control and Special / Status Access .....	117
PEEK – Peek at Queue / Stack.....	118
PFI – Poll for Interrupt.....	119
POP – Pop from Queue / Stack.....	120

PUSH – Push on Queue / Stack .....	120
REX – Redirect Exception.....	122
RTE – Return from Exception .....	124
STAT – Get Status of Queue / Stack .....	125
SYNC -Synchronize.....	126
TLBRW – Read / Write TLB.....	128
WFI – Wait for Interrupt.....	129
Vector Specific Instructions.....	130
MFILL –Mask Fill .....	130
MFIRST – Find First Set Bit.....	130
MFM – Move from Mask .....	132
MFVL – Move from Vector Length .....	132
MLAST – Find Last Set Bit.....	133
MTM – Move to Mask.....	134
MTVL – Move to Vector Length.....	135
Arithmetic / Logical .....	136
V2BITS .....	136
VBITS2V .....	137
VCIDX – Compress Index.....	138
VCMRSS – Compress Vector .....	139
VEINS / VMOVSV – Vector Element Insert .....	140
VEX / VMOVSV – Vector Element Extract .....	141
VSCAN .....	142
VSLLV – Shift Vector Left Logical .....	143
VSRLV – Shift Vector Right Logical.....	144
Memory Operations .....	145
CVLDx – Compressed Vector Load.....	145
CVSTx – Compressed Vector Store .....	147
Root Opcode Map.....	149
{R1} Integer Monadic Register Ops – Func <sub>10</sub> .....	150
{R2} Integer Dyadic Register Ops – Func <sub>7</sub> .....	150
{R3} Triadic Register Ops.....	150
{F1} Floating-Point Monadic Ops – Funct <sub>7</sub> .....	151

---

{F2} Floating-Point Dyadic Ops – Funct <sub>7</sub> .....	151
{F3} Floating-Point Dyadic Ops – Funct <sub>7</sub> .....	151
{VM} Vector Mask Register Ops.....	152
{OSR2} System Ops.....	153

## Instruction Formats

ANY1 has relatively few instruction formats. The instruction format is a fixed 36-bits in size. It is highly desirable to keep the instruction size to a minimum as minimally sized instructions have better entropy characteristics. The instruction format contains more decode information than is present in some instruction sets. Particularly there are register type codes associated with register spec fields. This is to keep the size of the instruction decoder hardware to a minimum. Otherwise, a ginormous decoder would be required to handle all possible combinations of instructions and types of registers. A vector machine that supports multiple primitive data types leads to a design that potentially has a lot of variation of instructions.

## Register Specifiers

The seven-bit register specifier field of an instruction looks like:

2625	24	20
Tb <sub>2</sub>	Rb <sub>5</sub>	

Register specifiers are always located at the same fixed positions in all instructions. This increases performance and minimizes decoding hardware.

Register specifiers contain a one or two-bit type code and a five-bit register number. The meaning of the type code is in the following table:

Ty <sub>2</sub>	Meaning
0	Scalar register
1	Vector register
2,3	Six-bit constant value (bit 0 of Ty is the high order bit of the constant) Not available for Ra, Rt register specs

Note that allowing either scalar or vector registers to be specified in the register spec eliminates the need for special classes of instructions to handle scalar-scalar, vector-scalar, or vector-vector operations.

For signed operations the six-bit constant is treated as a signed value and extended to 64-bits. For unsigned operations (BLTU, BGEU, SLTU, SGEU,...) the six-bit constant is treated as an unsigned value and zero extended to 64-bits.

## Constant Interpretation for Float Instructions

For floating point instructions specifying a constant treats the constant as a positive six-bit floating point constant which is extended to 64-bits before use. The exponent specifies a three-bit range of -3 to +4.

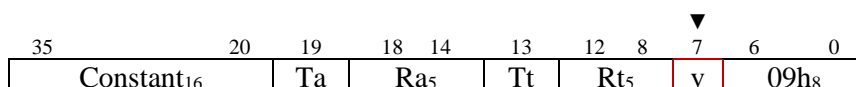
Bits 3 to 4	Bits 0 to 2
3-bit Exponent	3-bit significand

The significand has a hidden leading one bit.

## Vector Instruction Indicator

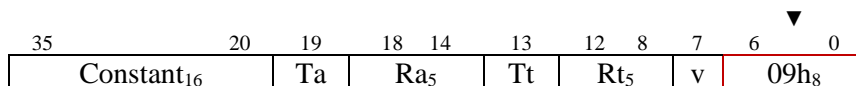
The processing core needs to know if an instruction is a vector instruction before it is fully decoded. Depending on if the instruction is a vector instruction, it may be re-decoded and sent into the pipeline multiple times. The processor needs to know very quickly and simply at the instruction fetch stage if the instruction is a vector operation. So, to help things along ANY1 encodes this information in bit 7 of all instructions. See the sample instruction below.

Immediate Format:



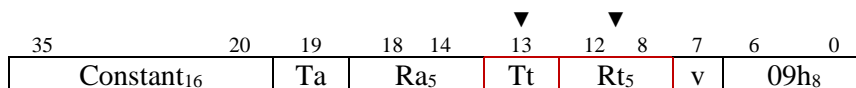
## Root Opcode

The root opcode determines the class of instructions executed. Some commonly executed instructions are also encoded at the root level to make more bits available for the instruction. The root opcode is always present in all instructions as the lowest seven bits of the instruction.



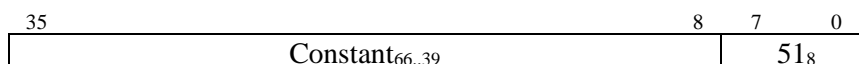
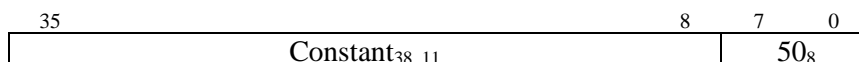
## Target Register Spec

Most instructions have a target register. The register spec for the target register is always in the same position, bits 8 to 13 of an instruction. The Tt field specifies the target register is a scalar (0) or vector (1) register.

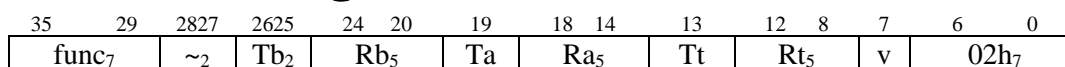


## Extended Immediate

The extended immediate instructions extend an immediate constant from bit 11 of the following instruction. Five root opcodes are reserved for extended immediates. See the [EXIn](#) description.







## Branch Instructions

Branch instructions make use of bits 8 to 13 and 27 to 35 to specify a 15-bit branch displacement for a range of  $\pm 71\text{kB}$ . The displacement is in terms of count of instructions skipped over. Note there are no vector branch instructions. Opcodes that would encode to vector branching are reserved for future use.

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Const <sub>6</sub>	0	4xh <sub>8</sub>						

## Instruction Modifiers

### IMOD Instruction Modifier - 58

This modifier adds two register spec fields allowing an instruction to use up to four source registers. It also allows a vector mask register to be optionally specified. Rounding mode for instructions supporting rounding is also possible to specify.

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0
~ <sub>5</sub>	Rm <sub>3</sub>	Tc	Td <sub>2</sub>	Rd <sub>6</sub>	Tc	Rc <sub>5</sub>	A	m <sub>3</sub>	z	58h <sub>8</sub>								

A: 00 = ignore mask and round  
 01 = apply vector mask  
 10 = apply rounding  
 11 = apply both vector mask and rounding.

### Branch Modifier – 5A

The branch modifier adds a link register to allow storing a return address. This allows conditional subroutine calls. Also, present is a branch target register spec. This allows conditional branching relative to a value in a register. Fifteen additional bits of branch displacement are provided, making the total branch displacement 30 bits (or  $\pm 2.4\text{GB}$ ).

35	21	20	19	18	14	13	12	8	7	6	0
Constant <sub>15</sub>	Tc <sub>2</sub>	Rc <sub>5</sub>	Tt	Rt <sub>5</sub>	0	5Ah <sub>7</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>						

### Stride Modifier – 5C

The stride modifier is used with vector load / store instructions to specify the stride of the operation.

35	21	20	19	18	14	13	12	11	9	8	7	0
Const <sub>15</sub>	Tc <sub>2</sub>	Rc <sub>5</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

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 ; scalar add of integers x2,x3

add.int.o v1,v2,v3,vm0 ; vector add of integers v2,v3

add.int.o v1,v2,x4,vm0 ; vector add scalar integers v2,x4

add.fp.o v1,v2,v3,vm0 ; 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.

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
06h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

v: 0 = scalar, 1 = vector op

### Float Instruction Format: R1

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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
20h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	34h <sub>7</sub>		

### Decimal Float Instruction Format: R1

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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
20h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	30h <sub>7</sub>		

### Operation:

If  $R_a < 0$   
 $R_t = -R_a$   
 else  
 $R_t = R_a$

### Vector Operation

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

if  $(V_m[x]) R_t[x] = R_a[x] < 0 ? -R_a[x] : R_a[x]$

**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  $(Vm[x]) \quad Vt[x] = Va[x] + Vb[x]$

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

## Integer Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</sub>		v	04h <sub>7</sub>	

1 clock cycle / N clock cycles (N = vector length)

## Integer Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
04h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

1 clock cycle / N clock cycles (N = vector length)

## Float Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
04h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>					

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

## Decimal Float Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
04h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	31h <sub>7</sub>					

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

## Posit Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
04h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	39h <sub>7</sub>					

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

**Vector Mask Instruction Format: R2 (MADD)**

35	29	28	25	24	22	21	18	17	15	14	11	10	8	7	0
04h <sub>7</sub>		0 <sub>4</sub>		Vmb <sub>3</sub>		0 <sub>4</sub>		Vma <sub>3</sub>		0 <sub>4</sub>		Vmt <sub>3</sub>		3Eh <sub>8</sub>	

1 clock cycle

**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.

## Integer Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	08h <sub>7</sub>				

1 clock cycle / N clock cycles (N = vector length)

## Integer Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
00h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>							

1 clock cycle / N clock cycles (N = vector length)

## Vector Mask Instruction Format: R2 (MADD)

35	29	28	25	24	22	21	18	17	15	14	11	10	8	7	0
00h <sub>7</sub>	0 <sub>4</sub>	Vmb <sub>3</sub>	0 <sub>4</sub>	Vma <sub>3</sub>	0 <sub>4</sub>	Vmt <sub>3</sub>	3Eh <sub>8</sub>								

1 clock cycle

## Operation:

$R_t = R_a \& \text{Imm}$

or

$R_t = R_a \& R_b$

## Vector Operation

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

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

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

**Exceptions:** none



# 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: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
func <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

Fn <sub>7</sub>	Function
30h	MOR
31h	MXOR
32h	MORT (MOR transpose)
33h	MXORT (MXOR transpose)

## 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).

If a vector BYTNDX instruction is issued and the target is a scalar register then the instruction searches all the vector elements and returns a value which varies from -1 to +511 in the scalar register. Thus, BYTNDX may be used to determine the length of a null termination string in the vector register.

### Instruction Format: R2

35 32	31 27	26 25	24 20	19	18 14	13	12 8	7	6	0
0 <sub>4</sub>	~ <sub>5</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	1A <sub>7</sub>	

35 32	31	20	19	18 14	13	12 8	7	6	0
1 <sub>4</sub>	Constant <sub>12</sub>		Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	1A <sub>7</sub>	

**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.

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
20h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v						02h <sub>7</sub>

1 clock cycle

## Operation:

$R_t = R_a < R_b ? -1 : R_a = R_b ? 0 : 1$

## Vector Operation

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

if  $(V_m[x]) \ V_t[x] = V_a[x] < V_b[x] ? -1 : V_a[x] = V_b[x] ? 0 : 1$

## 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. If the comparison is unordered 2.0 is returned.

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
10h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</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.

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
15h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</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

### Decimal Float Instruction Format: R2 (DFCMP)

Both values are treated as double precision (64-bit) decimal floating point numbers. The result is returned as a float value of -1.0, 0.0 or +1.0. If the comparison is unordered 2.0 is returned.

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
10h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	31h <sub>7</sub>					

1 clock cycle

# CNTPOP – Count Population

CNTPOP r1,r2

CNTPOP v1,v2

CNTPOP r1,vm2

## Description:

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

## Vector Operation

for x = 0 to VL - 1

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

## Instruction Format: R1

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
02h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

## Vector Mask Instruction Format: R1

35	26	25 22	21 18	17 15	14 13	12 8	7	0
0Dh <sub>10</sub>	~ <sub>4</sub>	0 <sub>5</sub>	Vm <sub>3</sub>	Tt <sub>2</sub>	Rt <sub>5</sub>	3Eh <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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
00h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

## Vector Mask Instruction Format: R1

35	26	25 22	21 18	17 15	14 13	12	8	7	0
00h <sub>10</sub>	~ <sub>4</sub>	0 <sub>5</sub>	Vm <sub>3</sub>	Tt <sub>2</sub>	Rt <sub>5</sub>	3Eh <sub>8</sub>			

**R1 Supported Formats:** .o

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Exceptions:** none

# COM – Ones Complement

## Description:

Bitwise complement all the bits in the register. 1's become 0's and 0's become 1's.

## Instruction Format: R1

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
03h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

1 clock cycle

## Operation

$$Rt = \sim Ra$$

## Vector Operation

for x = 0 to VL-1

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

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

**Exceptions:** none

# DEP – Deposit

## Description:

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

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

## Formats Supported: R4

31 29	28 26	25 20	19 14	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	59h <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

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
18h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

**Supported Formats:** .o

**Clock Cycles:** 1

**Execution Units:** Integer

### Operation:

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

**Exceptions:** none

## DIV[O][Z] – 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. The register form of this instruction may cause a divide by zero exception if enabled in the instruction.

### Instruction Format: RI

35	20	19	18 14	13	12 8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	10h <sub>7</sub>		

### Instruction Format: R2

35	29	2827	2625	24 20	19	18 14	13	12 8	7	6	0
10h <sub>7</sub>	OZ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>		

### Float Instruction Format: R2

35	29	2827	2625	24 20	19	18 14	13	12 8	7	6	0
09h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>		

**Execution Units:** ALU

**Clock Cycles:** 20

**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.

### Integer Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>			Ta		Ra <sub>5</sub>		Tt		Rt <sub>5</sub>	
							v		21h <sub>7</sub>	

**Execution Units:** ALU

**Clock Cycles:** 20

**Exceptions:** none

## DIVSU – Division Signed-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. The first operand is treated as a signed value, the second operand as unsigned.

### Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	12h <sub>7</sub>				

### Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
12h <sub>7</sub>	OZ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>							

**Execution Units:** ALU

**Clock Cycles:** 20

**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

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	11h <sub>7</sub>				

### Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
11h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>							

**Execution Units:** ALU

**Clock Cycles:** 20

**Exceptions:** none

## EOR – Bitwise Exclusive Or

### Description:

This is an alternate mnemonic for the [XOR](#) instruction. 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. The immediate constant is zero extended before use.

## EXIn – Extended Immediate

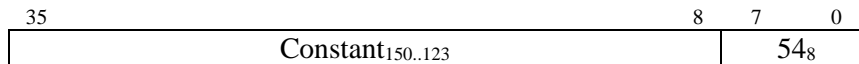
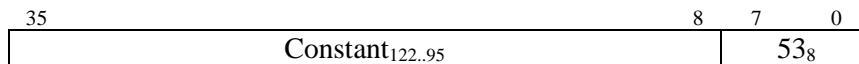
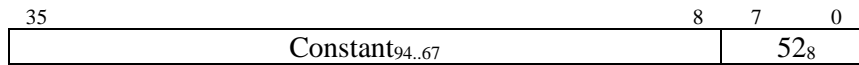
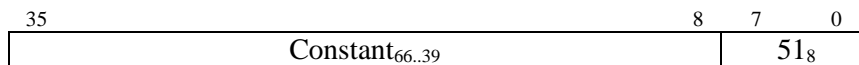
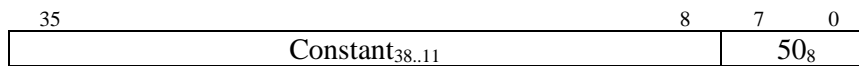
### Description:

These instructions are used to extend the constant field of the following instruction. The constant is extended from bit eleven. 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

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0
~ <sub>5</sub>	Rm <sub>3</sub>	Tc	Td <sub>2</sub>	Rd <sub>5</sub>	Tc	Rc <sub>5</sub>	A	m <sub>3</sub>	z	58h <sub>8</sub>								

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
04h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	1Ch <sub>7</sub>					

**Execution Units:** Integer ALU

**Exceptions:** none

**Notes:**





## 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

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0
~ <sub>5</sub>				Rm <sub>3</sub>	Tc	Td <sub>2</sub>	Rd <sub>6</sub>	Tc	Rc <sub>5</sub>	A	m <sub>3</sub>	z	58h <sub>8</sub>					

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
06h <sub>7</sub>				~ <sub>2</sub>	~ <sub>2</sub>	~ <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	1Ch <sub>7</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

### 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

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
29h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

## Float Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
03h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</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

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
28h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

### Float Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
02h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>					

### Operation:

```

IF Ra < Rb
    Rt = Ra
else
    Rt = Rb

```

## MOD – Instruction Modifier

### Description:

Used to modify the operation of the following instruction. Modifiers 50h to 52h are used to supply additional constant bits and are described as EXI instructions.

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

### Instruction Format: 58/D8 (IMOD)

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0					
~ <sub>5</sub>				Rm <sub>3</sub>		Tc		Td <sub>2</sub>		Rd <sub>6</sub>		Tc		Rc <sub>5</sub>		A		m <sub>3</sub>		z		58h <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 three 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

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 #0.

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

### Instruction Format: 5A (BRMOD)

35	21	2019	18	14	13	12	8	7	6	0
Constant <sub>15</sub>		Tc <sub>2</sub>	Rc <sub>5</sub>		Tt	Rt <sub>5</sub>		0	5Ah <sub>7</sub>	

The 5A 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 5A 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 5A modifier adds an additional fifteen bits to the branch displacement. This allows branching extended to  $\pm 2.4\text{GB}$ .

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

### Instruction Format: 5C/DC (STRIDE)

35	21	2019	18	14	1312	11	9	8	7	0
Const <sub>15</sub>		Tc <sub>2</sub>	Rc <sub>5</sub>		A	m <sub>3</sub>		z	5Ch <sub>8</sub>	

This format is used with vector load and store instructions to supply stride information and extend the address range of the load / store. Any additional constant modifiers (EXI0, EXI1, EXI2) should be placed before the stride modifier.

## MUL[O] – 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. The register form of the instruction may cause an overflow exception if the overflow enable bit in the instruction is set.

### Integer Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	06h <sub>7</sub>				

4 clock cycles

### Integer Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
06h <sub>7</sub>	O <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>							

4 clock cycles

**Exceptions:** overflow (if enabled)

### Float Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
08h <sub>7</sub>	O <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>							

25 clock cycles

**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.

## Integer Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</sub>		v	15h <sub>7</sub>	

1 clock cycle / N clock cycles (N = vector length)

## Integer Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
1Ch <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

1 clock cycle / 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.

## Integer Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	0Eh <sub>7</sub>				

4 clock cycles

## Integer Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
0Eh <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

4 clock cycles

**Exceptions:** none

## 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: R2

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0
~ <sub>5</sub>		~ <sub>3</sub>		Tc	~ <sub>2</sub>		~ <sub>6</sub>		Tc	Rc <sub>5</sub>		A	m <sub>3</sub>		z	58h <sub>8</sub>		

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
04h <sub>7</sub>		~ <sub>2</sub>		Tb <sub>2</sub>		Rb <sub>5</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</sub>		v	03h <sub>7</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	2524	23 21	20	19	14	13	8	7	0
07h <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
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	2524	23 21	20	19	14	13	8	7	0
21h <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  
 $Rt = -Ra$

## Clock Cycles: 1

**Execution Units:** Integer, 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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
04h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v		01h <sub>7</sub>	

1 clock cycle

## Operation:

$$Rt = !Ra$$

**Exceptions:** none

# OR – Bitwise Or

## Description:

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

## Integer Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	09h <sub>7</sub>				

1 clock cycle / N clock cycles (N = vector length)

## Integer Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
01h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>							

1 clock cycle / N clock cycles (N = vector length)

## Vector Mask Instruction Format: R2 (MADD)

35	29	28	25	24	22	21	18	17	15	14	11	10	8	7	0
01h <sub>7</sub>	0 <sub>4</sub>	Vmb <sub>3</sub>	0 <sub>4</sub>	Vma <sub>3</sub>	0 <sub>4</sub>	Vmt <sub>3</sub>	3Eh <sub>8</sub>								

1 clock cycle

## Operation

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

OR

$Rt = Ra \mid Rb$

## Vector Operation

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

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

**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 16-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.

### Integer Instruction Format: RI

The immediate format is normally used with a constant extension word as 24 bits are required to resolve the target positions.

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	17h <sub>7</sub>				

1 clock cycle

### Integer Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
17h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>							

1 clock cycle

**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. This instruction always uses a modifier to supply Rc or an immediate constant.

### Integer Instruction Format: R3

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0
~ <sub>5</sub>	~ <sub>3</sub>	Tc	~ <sub>2</sub>	~ <sub>6</sub>	Tc	Rc <sub>5</sub>	A	m <sub>3</sub>	z	58h <sub>8</sub>								

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
18h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	03h <sub>7</sub>					

1 clock cycle

### Operation:

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

Clock Cycles: 1

Execution Units: Integer

Exceptions:

None

## SEQ – Set if Equal



Float:

31262520	1914	138	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:

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
2Dh <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

### Float:

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

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
13h <sub>7</sub>	~ <sub>2</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Tb	Rb <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>					

### Decimal Float:

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

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
13h <sub>7</sub>	~ <sub>2</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Tb	Rb <sub>5</sub>	Tt	Rt <sub>5</sub>	v	31h <sub>7</sub>					

### Posit:

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

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
13h <sub>7</sub>	~ <sub>2</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Tb	Rb <sub>5</sub>	Tt	Rt <sub>5</sub>	v	39h <sub>7</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

### Integer:

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
2Fh <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

## SGT – Set if Greater Than

## 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

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	2Bh <sub>7</sub>				

### Integer Instruction Format: R2 (SLTU)

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
2Eh <sub>7</sub>	~ <sub>2</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Tb	Rb <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</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:

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
2Ah <sub>7</sub>	~ <sub>2</sub>	2 <sub>2</sub>	0 <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

### Float:

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
10h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>					

## Operation:

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

## Vector Operation

for x = 0 to VL - 1

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

# 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

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
19h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v					02h <sub>7</sub>	

## Vector Mask Instruction Format: R2 (MSLL)

35	29	2827	2625	24	20	19 17	16 14	13 11	10 8	7	0
0Eh <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	0 <sub>3</sub>	Vma <sub>3</sub>	0 <sub>3</sub>	Vmt <sub>3</sub>	3Eh <sub>8</sub>			

1 clock cycle

**Operation Size:** .o

**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

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0
~5	Rm <sub>3</sub>	Tc	Td <sub>2</sub>	Rd <sub>6</sub>	Tc	Rc <sub>5</sub>	A	m <sub>3</sub>	z	58h <sub>8</sub>								

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
10h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	03h <sub>7</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.

## Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	28h <sub>7</sub>				

## Integer Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
2Ch <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>							

## Float Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
12h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>							

## Decimal Float Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
12h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	31h <sub>7</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.

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
2Dh <sub>7</sub>	~ <sub>2</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Tb	Rb <sub>5</sub>	Tt	Rt <sub>5</sub>	v						02h <sub>7</sub>

#### Float:

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
13h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>					

#### Decimal Float:

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
13h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v						31h <sub>7</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.

This instruction is an alternate mnemonic for the SGEU instruction where the operands have been swapped.

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

### Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
2Fh <sub>7</sub>	~ <sub>2</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Tb	Rb <sub>5</sub>	Tt	Rt <sub>5</sub>	v					02h <sub>7</sub>	

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

### Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</sub>		v	2Ah <sub>7</sub>	

## Integer Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
2Eh <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</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.

### Integer Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	27h <sub>7</sub>				

### Integer/Posit Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
27h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

### Float / Decimal Float Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
14h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>					

# SQRT – Square Root

## Description:

This instruction takes the square root 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.

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
08h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

## Float Instruction Format: R1

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

35	29	28 27	26 24	23 21	20	19	18 14	13	12 8	7	6	0
08h <sub>7</sub>	~ <sub>2</sub>	Rm <sub>3</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	34h <sub>7</sub>		

## Decimal Float Instruction Format: R1

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

35	29	28 27	26 24	23 21	20	19	18 14	13	12 8	7	6	0
08h <sub>7</sub>	~ <sub>2</sub>	Rm <sub>3</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	30h <sub>7</sub>		

## Posit Instruction Format: R1

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

35	29	28 27	26 24	23 21	20	19	18 14	13	12 8	7	6	0
08h <sub>7</sub>	~ <sub>2</sub>	Rm <sub>3</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	38h <sub>7</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

## Instruction Format: R4

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0
~ <sub>5</sub>		Rm <sub>3</sub>		Tc	Td <sub>2</sub>		Rd <sub>5</sub>		Tc	Rc <sub>5</sub>		A	m <sub>3</sub>		z	58h <sub>8</sub>		

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
04h <sub>7</sub>		~ <sub>2</sub>		Tb <sub>2</sub>		Rb <sub>5</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</sub>		v	1Ch <sub>7</sub>	

**Operation Size:** .o

**Execution Units:** integer ALU

**Exceptions:** none

**Example:**

# SRL –Shift Right Logical

## Description:

Right shift an operand value by an operand value and place the result in the target register. Zeros are shifted into the most 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

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
21h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

## Vector Mask Instruction Format: R2 (MSRL)

35	29	2827	2625	24	20	19 17	16 14	13 11	10 8	7	0
0Fh <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	0 <sub>3</sub>	Vma <sub>3</sub>	0 <sub>3</sub>	Vmt <sub>3</sub>	3Eh <sub>8</sub>			

1 clock cycle

**Operation Size:** .o

**Execution Units:** integer ALU

**Exceptions:** none

**Example:**



# SRLP –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 Formats: R3

35	31	30	28	27	2625	24	20	19	18	14	1312	11	9	8	7	0
~ <sub>5</sub>	Rm <sub>3</sub>	Tc	Td <sub>2</sub>		Rd <sub>6</sub>	Tc	Rc <sub>5</sub>	A	m <sub>3</sub>	z	58h <sub>8</sub>					

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
05h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>		Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt		Rt <sub>5</sub>	v	1Ch <sub>7</sub>			

**Operation Size:** .o

**Execution Units:** integer ALU

**Exceptions:** none

**Example:**

# SUB - Subtract

## Description:

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

## Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
05h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v						02h <sub>7</sub>

## Float Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
05h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	35h <sub>7</sub>					

## Decimal Float Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
05h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	31h <sub>7</sub>					

## Posit Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
05h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	39h <sub>7</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

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>			Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	05h <sub>7</sub>		

### Operation:

$$Rt = Imm - Ra$$

**Exceptions:** none

## SXB –Sign Extend Byte

### Description:

Zero extend byte.

### Integer Instruction Format: R1

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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
14h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Exceptions:** none

**Notes:**

## SXW –Sign Extend Wyde

### Description:

### Integer Instruction Format: R1

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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
15h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Exceptions:** none

**Notes:**

# SXT –Sign Extend Tetra

**Description:**

**Integer Instruction Format: R1**

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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
16h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Exceptions:** none

**Notes:**

## 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).

### Integer Instruction Format: RI

The RI instruction format may be used with an immediate extension word for full 21-bit constants.

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	23h <sub>7</sub>				

1 clock cycle

### Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
23h <sub>7</sub>	~2	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

**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).

### Integer Instruction Format: RI

The RI instruction format may be used with an immediate extension word for full 16-bit constants.

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</sub>		v	1Bh <sub>7</sub>	

1 clock cycle

### Instruction Format: R2

35	29	28	27	26	25	24	20	19	18	14	13	12	8	7	6	0
1Bh <sub>7</sub>		~ <sub>2</sub>		Tb <sub>2</sub>		Rb <sub>5</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</sub>		v	02h <sub>7</sub>	

### 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.

## Integer Instruction Format: RI

35	20	19	18	14	13	12	8	7	6	0
Constant <sub>16</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</sub>		v	0Ah <sub>7</sub>	

1 clock cycle / N clock cycles (N = vector length)

## Integer Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
02h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

1 clock cycle / N clock cycles (N = vector length)

## Vector Mask Instruction Format: R2 (MADD)

35	29	28 25	24 22	21 18	17 15	14 11	10 8	7	0
02h <sub>7</sub>	0 <sub>4</sub>	Vmb <sub>3</sub>	0 <sub>4</sub>	Vma <sub>3</sub>	0 <sub>4</sub>	Vmt <sub>3</sub>	3Eh <sub>8</sub>		

1 clock cycle

## 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:

Zero extend byte.

### Integer Instruction Format: R1

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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
0Ch <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Exceptions:** none

**Notes:**

## ZXW –Zero Extend Wyde

### Description:

### Integer Instruction Format: R1

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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
0Dh <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Exceptions:** none

**Notes:**

## ZXT –Zero Extend Tetra

**Description:**

**Integer Instruction Format: R1**

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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
0Eh <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Exceptions:** none

**Notes:**

## Graphics

### 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  $1.0 - \text{alpha}$  is used. Each component of the color is blended.

#### Instruction Format: R3

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0
$\sim_5$	Rm <sub>3</sub>			Tc	Td <sub>2</sub>		Rd <sub>6</sub>		Tc	Rc <sub>5</sub>			A	m <sub>3</sub>		z	58h <sub>8</sub>	

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
30h <sub>7</sub>		~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</sub>		v	03h <sub>7</sub>	

#### Operation:

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

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

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

**Clock Cycles:** 2

# TRANSFORM – Transform Point

## Description:

The point transform instruction transforms a point from one location to another using a transform function. The transform function has 12 co-efficients in the form of a matrix to used in the calculation.

Points are represented in 16.16 fixed-point format.

35	29	28 24	23 21	20	19	18 14	13	12 8	7	6	0
11h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	01h <sub>7</sub>		

**Clock Cycles:** 2

## RW\_COEFF – Read/Write Co-efficient

### Description:

RW\_COEFF reads and writes a coefficient value to be used for the transform matrix. Ra contains the number of the coefficient to read or write. Rb contains the new value for the coefficient. Coefficients are in 16.16 fixed point format.

### Instruction Format: R2

35	29	2827	2625	24	20	19	18	14	13	12	8	7	6	0
3Eh <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	v	02h <sub>7</sub>					

### Co-efficient Matrix:

AA	AB	AC	AT
BA	BB	BC	BT
CA	CB	CC	CT

Regno in Ra	Coefficient Accessed
0	AA
1	AB
2	AC
3	AT
4	BA
5	BB
6	BC
7	BT
8	CA
9	CB
10	CC
11	CT
12	CMD – bit 0, 1=transform, 0 = pass through

## 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

35 32	31 20	19	18 14	13	12 10	9 8	7	0
15 <sub>3..0</sub>	Const <sub>12</sub>	Ta	Ra <sub>5</sub>	0	DC <sub>3</sub>	IC <sub>2</sub>	60h <sub>8</sub>	

#### Commands:

IC <sub>2</sub>	Mne.	Operation
0	NOP	no operation
1	inline	invalidate line associated with given address
2	invall	invalidate the entire cache (address is ignored)
3		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	inline	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.

35	32	31	20	19	18	14	13	12	8	7	0
Func <sub>3..0</sub>	Const <sub>12</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</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.

35	32	31	30	29	27	26	25	24	20	19	18	14	13	12	8	7	0
Func <sub>4</sub>	~ <sub>2</sub>	Sc <sub>3</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</sub>	61h <sub>8</sub>								

z: 1= zero extend, 0 = sign extend

S	Multiplier
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

### Operation:

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

### Vector forms

### Stridden Form (LDS)

35	21	20	19	18	14	13	12	11	9	8	7	0
Const <sub>15</sub>	Tc <sub>2</sub>	Rc <sub>5</sub>	A	m <sub>3</sub>	z	5Ch <sub>8</sub>						

35	32	31	20	19	18	14	13	12	8	7	0
Func <sub>3..0</sub>	Const <sub>12</sub>	Ta	Ra <sub>5</sub>	Tt	Rt <sub>5</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

Func <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.

35	21	2019	18	14	1312	11	9	8	7	0
Const <sub>15</sub>	Tc <sub>2</sub>	Rc <sub>5</sub>	A	m <sub>3</sub>	z	5Ch <sub>8</sub>				

35	32	31	20	19	18	14	13	12	8	7	0
Func <sub>3..0</sub>		Const <sub>12</sub>		Ta	Ra <sub>5</sub>		Tt	Rt <sub>5</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:

$$Rt = \text{Memory}_8[d + Ra]$$

**OR**

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

**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:

$$Rt = \text{Memory}_8[d + Ra]$$

**OR**

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

**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:

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

**OR**

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

### 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:

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

**OR**

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

**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:

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

**OR**

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

**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:

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

**OR**

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

### 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:

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

**OR**

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

### 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>		6A <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

# 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

35	32	31	27	26	25	24	20	19	18	14	13	8	7	0
Func <sub>3..0</sub>	C <sub>5</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</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.

35	31	30	28	27	26	25	24	20	19	18	14	13	12	11	9	8	7	0
~ <sub>5</sub>	Rm <sub>3</sub>	Tc	Td <sub>2</sub>	Rd <sub>6</sub>	Tc	Rc <sub>5</sub>	A	m <sub>3</sub>	z	58h <sub>8</sub>								

35	32	3130	29 27	2625	24 20	19	18 14	13	8	7	0
Func <sub>4</sub>	~ <sub>2</sub>	Sc <sub>3</sub>	Tb <sub>2</sub>	Rb <sub>6</sub>	Ta	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)

35	21	2019	18	14	1312	11	9	8	7	0
Const <sub>15</sub>		Tc <sub>2</sub>	Rc <sub>5</sub>	A	m <sub>3</sub>	z	5Ch <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>	F2h <sub>8</sub>					

Data is stored to 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 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]

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.

35	21	2019	18 14	1312	11 9	8	7	0
Const <sub>15</sub>	Tc <sub>2</sub>	Rc <sub>5</sub>	A	m <sub>3</sub>	z	5Ch <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.

### Instruction Format: ST

### Register Indirect Operation:

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

### Indexed Operation:

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

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

### Description:

This instruction stores a byte (8 bit) value to memory. After the byte is stored to memory the register is zeroed out.

### Instruction Format: ST

### Register Indirect Operation:

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

### Indexed Operation:

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

## 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

#### *Displacement*

The conditional branch displacement is in terms of instruction count skipped over to increase the range a branch may cover. A displacement of one represents nine nybbles. Code using conditional branches must be sequentially laid out in memory, instructions adjacent to each other with no “holes” in the layout.

The displacement for the branch-and-link instruction is the number of nybbles to the target address from the current one. This allows subroutines to be aligned at any nybble address.

#### *Modifier*

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 30 bits allowing a  $\pm 2.4\text{GB}$  branch range.

### BAL – Branch and Link

#### **Description:**

This instruction may be used to call a subroutine using relative addressing. 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 26 bits or  $\pm 16\text{MB}$ .

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

35	10	9	8	7	0
Constant <sub>26</sub>	Rt <sub>2</sub>	41h <sub>8</sub>			

**Flags Affected:** none

#### **Operation:**

$$\text{Rt} = \text{IP} + 9$$

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

**Execution Units:** Branch

**Exceptions:** none

**Notes:**



## BBS – Branch if Bit Set

### Description:

This instruction branches to the target address if the bit number identified by the Rb specifier in the instruction is set in Ra. Rb may be a value in a register or a six-bit unsigned immediate value. 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 x31 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

35	29	2827	26	22	2120	19	15	1413	12	8	7	6	0
Const <sub>7</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Ty <sub>2</sub>	Const <sub>5</sub>	0	4Dh <sub>8</sub>					

### Operation:

If (Ra[Rb])

$$IP = IP + \text{Displacement}12*9$$

With Modifier

$$Rt = IP + 9$$

If (Ra[Rb])

$$IP = Rc + \text{Displacement}26*9$$

**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 x31 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

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Const <sub>6</sub>	0	4Eh <sub>8</sub>						

### Operation:

If (Ra = Rb)

$$IP = IP + \text{Displacement}_{15*9}$$

With Modifier

$$Rt = IP + 9$$

If (Ra = Rb)

$$IP = Rc + \text{Displacement}_{30*9}$$

**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.

### Formats Supported: BR

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Const <sub>6</sub>	0	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 x31 then the program counter value is used.

### Formats Supported: BR

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Const <sub>6</sub>	0	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 x31 then the program counter value is used.

### Formats Supported: BR

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Const <sub>6</sub>	0	48h <sub>8</sub>						

### Operation:

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 x31 then the program counter value is used.

## Formats Supported: BR

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Const <sub>6</sub>	0	4A <sub>h8</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.

### Formats Supported: BR

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Const <sub>6</sub>	0	4Fh <sub>8</sub>						

### Operation:

If (Ra  $\neq$  Rb)

IP = IP + 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.

### Formats Supported: BR

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Tb	Rb <sub>5</sub>	Const <sub>6</sub>	0	49h <sub>8</sub>						

### Operation:

If (Ra >= Rb)

PC = Rc + Displacement

**Execution Units:** Branch

**Exceptions:** none



## BLEU – Branch if Less Than or Equal Unsigned

### Description:

This is an alternate mnemonic for the [BGEU](#) instruction, where the register operands have been swapped.

This instruction branches to the target address if the contents of Ra is greater than or equal to Rb, otherwise program execution continues with the next instruction. The values in Ra and Rb are treated as unsigned values.

### Formats Supported: BR

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Tb	Rb <sub>5</sub>	Const <sub>6</sub>	0	4Bh <sub>8</sub>						

### Operation:

If (Ra >= Rb)

IP = IP + 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.

### Formats Supported: BR

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Const <sub>6</sub>	0	48h <sub>8</sub>						

### Operation:

If (Ra < Rb)

IP = IP + 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.

### Formats Supported: BR

35	27	26	25	24	20	19	18	14	13	8	7	6	0
Const <sub>9</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	Const <sub>6</sub>	0	4Ah <sub>8</sub>						

### Operation:

if (Ra < Rb)  
     IP = IP + Displacement

### Execution Units: Branch

### Exceptions: none

## BRA – Unconditional Branch

### Description:

This instruction is an alternate mnemonic for the [BAL](#) instruction. The address range is 26 bits or  $\pm 16\text{MB}$ .

### Formats Supported: JAL

35	10	9	8	7	0
Constant <sub>26</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:**

## BSR – Unconditional Branch to Subroutine

### Description:

This instruction is an alternate mnemonic for the [BAL](#) instruction. The address range is 26 bits or  $\pm 16\text{MB}$ .

### Formats Supported: JAL

35	10	9	8	7	0
Constant <sub>26</sub>		1 <sub>2</sub>		41h <sub>8</sub>	

**Flags Affected:** none

### Operation:

$\text{Rt} = \text{IP} + 9$

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

**Execution Units:** Branch

**Exceptions:** none

**Notes:**

# CALL – Call Subroutine

## Description:

This instruction changes program flow to the target address specified and stores the return address on the stack. The target address may be either an absolute address or an instruction pointer relative address. The ‘A’ bit of the instruction selects absolute addressing if set.

## Formats Supported: CALL

35	10	9	8	7	0
Constant <sub>26</sub>		0	A	7Ch <sub>8</sub>	

**Flags Affected:** none

## Operation:

$SP = SP - 8$

$Memory[SP] = IP + 9$

If (A)

$IP = \text{constant}$

else

$IP = IP + \text{constant}$

**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

35 31	30 28	27	26 25	24 20	19	18 14	13 12	11 9	8	7	0
~ <sub>5</sub>	Rm <sub>3</sub>	Tc	Td <sub>2</sub>	Rd <sub>6</sub>	Tc	Rc <sub>5</sub>	A	m <sub>3</sub>	z	58h <sub>8</sub>	

35	20	19	18 14	13 10	9 8	7	6	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	0 <sub>4</sub>	Cn <sub>2</sub>	v	22h <sub>7</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

35 31	30 28	27	26 25	24 20	19	18 14	13 12	11 9	8	7	0
~ <sub>5</sub>	Rm <sub>3</sub>	Tc	Td <sub>2</sub>	Rd <sub>6</sub>	Tc	Rc <sub>5</sub>	A	m <sub>3</sub>	z	58h <sub>8</sub>	

35	29	28 27	26 25	24 20	19	18 14	13 10	9 8	7	6	0
22h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Ta	Ra <sub>5</sub>	0 <sub>4</sub>	Cn <sub>2</sub>	v	03h <sub>7</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 26 bits or 16MB.

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

35	10	9	8	7	0
Constant <sub>26</sub>			Rt <sub>2</sub>	40h <sub>8</sub>	

**Flags Affected:** none

## Operation:

$$Rt = IP + 9$$

$$IP = \text{displacement}$$

**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 address of the next instruction 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 x31 is specified for Ra then the current instruction pointer value is used.

### Formats Supported: JALR

35	20	19	18	14	13	12	8	7	0
Constant <sub>16</sub>	Ta	Ra <sub>5</sub>	0	Rt <sub>5</sub>	42h <sub>8</sub>				

**Flags Affected:** none

### Operation:

$Rt = IP + 9$

If Ra=31

$IP = IP + \text{displacement}$

Else

$IP = Ra + \text{Displacement}$

**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 26 bits or 16MB.

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

35	10	9	8	7	0
Constant <sub>26</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 returns from a subroutine by transferring program execution to the address popped off the stack. A constant is added to the stack pointer. The assembler assumes a constant of eight if not specified.

### Formats Supported: RET

35	20	19	18 14	13	12 8	7	0
Constant <sub>16</sub>	0 <sub>1</sub>	30 <sub>5</sub>	0 <sub>1</sub>	30 <sub>5</sub>	7B <sub>8</sub>		

**Flags Affected:** none

**Operation:**

**Execution Units:** Branch, Memory

**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

35	22	21	14	13	12	8	7	0
~ <sub>14</sub>		Cause <sub>8</sub>		0 <sub>2</sub>	0 <sub>5</sub>		00h <sub>8</sub>	

#### Operation:

PMSTACK = (PMSTACK << 4) | 10

CAUSE = Const<sub>8</sub>

EIP = IP

IP = tvec[4]

#### Execution Units: Branch

#### Clock Cycles:

#### Exceptions: none

#### Notes:

## CSRx – Control and Special / Status Access

### Description:

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

### Instruction Format: CSR

35	22	2120	19	15	1413	12	8	7	6	0
Constant <sub>14</sub>	Ta <sub>2</sub>	Ra <sub>5</sub>	Op <sub>2</sub>	Rt <sub>5</sub>	v	0Fh <sub>7</sub>				

Op <sub>2</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

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.

## PEEK – Peek at Queue / Stack

### Description:

This instruction returns the top value into Rt from the hardware queue specified in Ra. The hardware queue position is not advanced. Unused value bits should read as zero. Used the STAT instruction to get the queue status.

### Instruction Format: PEEKQ

31	26	25	20	19	14	13	8	7	0
0Ah <sub>6</sub>	0 <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	44h <sub>8</sub>					

### Instruction Format: PEEKQI

31	26	25	20	19	14	13	8	7	0
0Eh <sub>6</sub>	0 <sub>6</sub>	Qno <sub>6</sub>	Rt <sub>6</sub>	44h <sub>8</sub>					

### Instruction Format: PEEKQ

**Exceptions:** none

## 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. Rt is loaded with the cause code in the low order eight bits, and the interrupt level in bits eight to eleven of the register.

### Instruction Format: SYS

35	29	28	14	13	12	8	7	6	0
11h <sub>7</sub>	~15			0	Rt <sub>5</sub>		0	07h <sub>7</sub>	

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

### Operation:

```

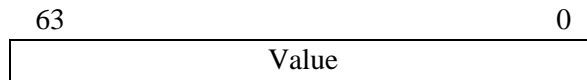
if (irq <> 0)
    Rt[7:0] = cause code
    Rt[11:8] = irq level
    PMSTACK = (PMSTACK << 4) | 10
    CAUSE = Const8
    EIP = IP
    IP = tvec[4]
  
```

**Execution Units:** Branch

## POP – Pop from Queue / Stack

### Description:

This instruction pops a value into Rt from the hardware queue specified in Ra. The hardware queue position is advanced. Unused value bits should read as zero. To check the queue status, use the STAT instruction.



Value: the value that was pushed to the queue

### Instruction Format: POP

31 26 25 20	19 14	13 8	7 0
09h <sub>6</sub>	0 <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>
			07h <sub>8</sub>

### Instruction Format: POPI

31 26 25 20	19 14	13 8	7 0
0Dh <sub>6</sub>	0 <sub>6</sub>	Qno <sub>6</sub>	Rt <sub>6</sub>
			07h <sub>8</sub>

**Exceptions:** none

### Notes:

Queue #15 is the instruction trace que

## PUSH – Push on Queue / Stack

### Description:

This instruction pushes an N-bit value in Ra onto the hardware queue specified in Rb. Where N is implementation defined between 1 and 64 bits. To check the queue status, use the STATQ instruction.

### Instruction Format: PUSH

31 26 25 20	19 14	13 8	7 0
08h <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	0 <sub>6</sub>
			07h <sub>8</sub>

### Instruction Format: PUSHI

31 26 25 20	19 14	13 8	7 0
0Ch <sub>6</sub>	Qno <sub>6</sub>	Ra <sub>6</sub>	0 <sub>6</sub>
			07h <sub>8</sub>



**Instruction Format:** PUSHQ

**Exceptions:** none

# 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>	07h <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:

```

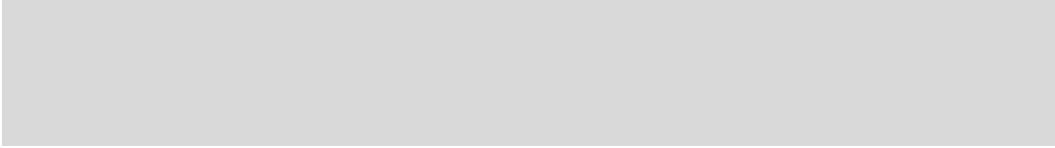
REX 1          ; redirect to supervisor handler

; 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 constant<sub>12</sub> field specifies a return offset in terms of half-bytes.

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

### Instruction Format: SYS

35	29	2827	2625	24	20	19	8	7	0
13h <sub>7</sub>	~ <sub>2</sub>	Tb <sub>2</sub>	Rb <sub>5</sub>	Constant <sub>12</sub>			07h <sub>8</sub>		

**Flags Affected:** none

### Operation:

PMSTACK = PMSTACK >> 4  
 Semaphore[0] = 0  
 Semaphore[Rb] = 0  
 IP = EIP + Ra

**Execution Units:** Branch

**Clock Cycles:**

**Exceptions:** none

**Notes:**

# STAT – Get Status of Queue / Stack

## Description:

This instruction returns a queue status value into Rt from the hardware queue specified in Ra. The hardware queue position is not advanced. Unused value bits should read as zero.

63	62	61	54	53	48	47	0	9	0
Qe	Dv	~	~	~	~	~	~	~	Data Count

## Fields

Qe: empty. If set, this bit indicates that the queue/stack is empty.

Dv: data valid. If this bit is set it indicates that valid data is present at the queue.

Dc: data count: The number of items left in the queue

## Instruction Format: POP

31	26	25	20	19	14	13	8	7	0
0Bh <sub>6</sub>	0 <sub>6</sub>	Ra <sub>6</sub>	Rt <sub>6</sub>	07h <sub>8</sub>					

## Instruction Format: POPI

31	26	25	20	19	14	13	8	7	0
0Fh <sub>6</sub>	0 <sub>6</sub>	Qno <sub>6</sub>	Rt <sub>6</sub>	07h <sub>8</sub>					

**Exceptions:** none

# 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
~ <sub>3</sub>	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>	07h <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> 07h<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

35	29	28		7	6	0
12h <sub>7</sub>	~22				07h <sub>7</sub>	

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

**Execution Units:** Branch

## Vector Specific Instructions

### MFILL –Mask Fill

#### Description

Fill vector mask register with bits.

The first Ra bits of the vector mask register (Vmt) are set to one. The remaining bits of the mask register are set to zero.

#### Instruction Format: R1

31 26	25 20	19 14	13 11	10 8	7 0
0Ch <sub>6</sub>	~ <sub>6</sub>	Ra <sub>6</sub>	0 <sub>3</sub>	Vmt <sub>3</sub>	80h <sub>8</sub>

#### Operation

Vmt = 0

for x = 0 to VLMAX

if (x < Ra) Vmt[x] = 1

#### Execution Units: ALUs

### MFIRST – Find First Set Bit

#### Description

The position of the first bit set in the mask register is copied to the target register. If no bits are set the value is 128. The search begins at the least significant bit and proceeds to the most significant bit.

#### Instruction Format: R1

31 26	25 20	19 17	16 14	13 8	7 0
0Eh <sub>6</sub>	~ <sub>6</sub>	0 <sub>3</sub>	Vm <sub>3</sub>	Rt <sub>6</sub>	80h <sub>8</sub>

#### Operation

Rt = first set bit number of (Vm)

#### Exceptions: none

#### Execution Units: ALUs



## MFMM – Move from Mask

### Description

Move a mask register to a general-purpose register.

### Instruction Format: R1

31 26	25 20	19 17	16 14	13 8	7 0
11h <sub>6</sub>	~ <sub>6</sub>	0 <sub>3</sub>	Vm <sub>3</sub>	Rt <sub>6</sub>	80h <sub>8</sub>

### Operation

$V_{mt} = R_a$

**Execution Units:** ALUs

## MFVL – Move from Vector Length

### Description

Move vector length register to a general-purpose register.

### Instruction Format: R1

31 26	25 20	19 17	16 14	13 8	7 0
13h <sub>6</sub>	~ <sub>6</sub>	0 <sub>3</sub>	0 <sub>3</sub>	Rt <sub>6</sub>	80h <sub>8</sub>

### Operation

$V_{mt} = R_a$

**Execution Units:** ALUs

# MLAST – Find Last Set Bit

## Description

The position of the last bit set in the mask register is copied to the target register. If no bits are set the value is 128. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

## Instruction Format: R1

31	26	25	20	19	17	16	14	13	8	7	0
0Fh <sub>6</sub>		~ <sub>6</sub>		0 <sub>3</sub>		Vm <sub>3</sub>		Rt <sub>6</sub>		80h <sub>8</sub>	

## Operation

Rt = last set bit number of (Vm)

**Exceptions:** none

**Execution Units:** ALUs

# MTM – Move to Mask

## Description

Move a general-purpose register to a mask register.

## Instruction Format: R1

31	26	25	20	19	14	13	11	10	8	7	0
10h <sub>6</sub>		~ <sub>6</sub>		Ra <sub>6</sub>		0 <sub>3</sub>		Vmt <sub>3</sub>		80h <sub>8</sub>	

## Operation

$Vmt = Ra$

**Execution Units:** ALUs

# MTVL – Move to Vector Length

## Description

Move a general-purpose register to the vector length register.

## Instruction Format: R1

31 26	25 20	19 14	13 11	10 8	7 0
12h <sub>6</sub>	~ <sub>6</sub>	Ra <sub>6</sub>	0 <sub>3</sub>	0 <sub>3</sub>	80h <sub>8</sub>

## Operation

$$V_{mt} = R_a$$

**Execution Units:** ALUs

## 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

35	29	28	24	23	21	20	19	18	14	13	12	8	7	0
18h <sub>7</sub>		~ <sub>5</sub>		m <sub>3</sub>		z	1 <sub>1</sub>		Ra <sub>5</sub>	0 <sub>1</sub>		Rt <sub>5</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

35	29	28 24	23 21	20	19	18 14	13	12 8	7	0
19h <sub>7</sub>	~ <sub>5</sub>	m <sub>3</sub>	z	0 <sub>1</sub>	Ra <sub>5</sub>	1 <sub>1</sub>	Rt <sub>5</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	2524	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  $VL - 1$

$V_t[x] = sum$

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 (CVLDSx)**

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>		Vt <sub>6</sub>		E6h <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. Rc may specify either a register or a six-bit unsigned constant. 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.

**Operation:**

```

y = 0
for x = 0 to vector length
    if Rb is a constant
        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

```

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

### Indexed Form (CVLDxVX)

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>		Vt <sub>6</sub>		E7h <sub>8</sub>	

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

### Operation:

```

y = 0
for x = 0 to vector length
    if (Vm[x])
        Vt[y] = Memory[d+Ra + Vc[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:**

## Stridden Form (CVSTSx)

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>	Vb <sub>6</sub>	Ra <sub>6</sub>	Cnst <sub>6</sub>	F6h <sub>8</sub>						

Data is stored to memory at addresses beginning with the sum of Ra and a vector element from Vb. The store location is adjusted by a stride amount contained in Rc or a six-bit unsigned immediate.

**Operation:**

```

y = 0
for x = 0 to vector length
    n = Rc * y
    if (Vm[x])
        Memory[d+Ra + n] = Vs[x]
        if (z) Vs[x] = 0
        y = y + 1
  
```

## Indexed Form (CVSTxVX)

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	26	25	20	19	14	13	8	7	0
Func <sub>3..0</sub>	C <sub>2</sub>	Vb <sub>6</sub>	Ra <sub>6</sub>	Cnst <sub>6</sub>	F7h <sub>8</sub>						

Data is stored to 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])
        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/R4}	ADD	SUBF	MUL	{SYS}
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}			{VM}	NOP
<b>Branch Unit</b>								
01000	FBLT	FBGE	DFBLT	DFBGE			FBEQ	FBNE
01001	BLT	BGE	BLTU	BGEU		BBS	BEQ	BNE
<b>Instruction Modifiers (Prefixes)</b>								
01010	EXI	EXI	EXI					
01011	IMOD	BTFLD	BRMOD		STRIDE			
<b>Memory Unit</b>								
01100	LDx	LDxX			LDxZ	LDxXZ		
01101								LSM
01110	STx	STxX						
01111	JAL	BAL	JALR	RTS	CALL			
<b>Vector ALU</b>								
10000		{R1}	{R2}	{R3}	ADD	SUBF	MUL	
10001	AND	OR	XOR			{SET}	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		STRIDE			
11100			LDSx	LDxVX			CVLDSx	CVLDxVX
11101								
11110			STSx	STxVX			CVSTSx	CVSTxVX
11111								

## {R1} Integer Monadic Register Ops – Func<sub>10</sub>

	000	001	010	011	100	101	110	111
xxxx000	CNTLZ	CNTLO	CNTPOP	COM	NOT	NEG	ABS	NABS
xxxx001	SQRT			TST	ZXB	ZXW	ZXT	
xxxx010	PTRINC	TRANSFORM			SXB	SXW	SXT	
xxxx011	V2BITS	BITS2V			VCMPRSS	VCIDX	VSCAN	
xxxx100								
xxxx101								
xxxx110								
xxxx111								

## {R2} Integer Dyadic Register Ops – Func<sub>7</sub>

	000	001	010	011	100	101	110	111
x000	AND	OR	XOR		ADD	SUB	MUL	
x001	NAND	NOR	XNOR			MULF	MULU	MULH
x010	DIV	DIVU	DIVSU	REM	REMU	REMSU	MULSU	PERM
x011	DIF	SLL	SLLI	WYDNDX	MULF	MULSUH	MULUH	RGF
x100	CMP	SRL	SRLI	U21NDX			SEQ	SNE
x101	MIN	MAX			SLT	SGE	SLTU	SGEU
x110	BMM.or	BMM.xor	BMM	BMM				
x111	VSLLV	VSLRV	VEX	VEINS			RD_COEFF	WR_COEFF

## {R3/R4} Triadic Register Ops

	000	001	010	011	100	101	110	111
x000					MUX			
x001								
x010	SLLP	SLLPI						
x011	PTRDIF							
x100			CHK					
x101								
x110	BLEND							FDP
x111								

### {F1} Floating-Point Monadic Ops – Funct<sub>7</sub>

	000	001	010	011	100	101	110	111
x000	FMOV	FRSQRT	FTOI	ITOF			FSIGN	FMAN
x001	FSQRT	FS2D	FS2Q	FD2Q	FSTAT		ISNAN	FINITE
x010	FTX	FCX	FEX	FDX	FRM	TRUNC	FSYNC	FRES
x011	FSIGMOID	FD2S	FQ2S	FQ2D			FCLASS	UNORD
x100	FABS	FNABS	FNEG					
x101								
x110								
x111								

### {F2} Floating-Point Dyadic Ops – Funct<sub>7</sub>

	000	001	010	011	100	101	110	111
x000	SCALEB		FMIN	FMAX	FADD	FSUB		
x001	FMUL	FDIV	FREM	FNXT				
x010	FCMP	FSEQ	FSLT	FSLE	FSNE	FCMPB	FSETM	
x011	CPYSGN	SGNINV	SGNAND	SGNOR	SGNXOR	SGNXNOR	FCLASS	
x100								
x101								
x110								
x111								

### {F3} Floating-Point Dyadic Ops – Funct<sub>7</sub>

	000	001	010	011	100	101	110	111
x000	FMA	FMS	FNMA	FNMS				
x001								
x010								
x011								
x100								
x101								
x110								
x111								

## {DF2} Decimal Floating-Point Dyadic Ops – Funct<sub>7</sub>

	000	001	010	011	100	101	110	111
x000	SCALEB		DFMIN	DFMAX	DFADD	DFSUB		
x001	DFMUL	DFDIV	DFREM	DFNXT				
x010	DFCMP	DFSEQ	DFSLT	DFSLE	DFSNE	DFCMPB	DFSETM	
x011	CPYSGN	SGNINV	SGNAND	SGNOR	SGNXOR	SGNXNOR	FCLASS	
x100								
x101								
x110								
x111								



## {VM} Vector Mask Register Ops

	000	001	010	011	100	101	110	111
x000	MAND	MOR	MXOR		MADD	SUB	MSLL	MSRL
x001	MNAND	MNOR	MXNOR		MFILL	MPOP	MFIRST	MLAST
x010	MTM	MFM	MTVL					
x011								
x100								
x101								
x110								
x111								

## {OSR2} System Ops

	000	001	010	011	100	101	110	111
x000	LLAL	LLAH			LPAL	LPAH		
x001	PUSHQ	POPQ	PEEKQ	STATQ		POPQI	PEEKQI	STATQI
x010	REX	PFI	WAI	RTE	SETKEY			
x011	SETTO	GETTO	GETZL			MVSEG	TLBRW	SYNC
x100								
x101								
x110								
x111								