[Draw your reader in with an engaging abstract. It is typically a short summary of the document. When you're ready to add your content, just click here and start typing.]

Thor2023

[Document subtitle]

Robert Finch

Table of Contents

Thor2023

Nomenclature

The ISA refers to primitive object sizes following the convention suggested by Knuth of using Greek.

Number of Bits		Instructions	Comment
8	byte	LDB, STB	UTF8 usage
16	wyde	LDW, STW	
24	char	LDC, STC	UTF24 usage
32	tetra	LDT, STT	
40	penta	LDP, STP	Instruction size
64	octa	LDO, STO	
96		LDN, STN	

The register used to address instructions is referred to as the instruction pointer or IP register. The instruction pointer is a synonym for instruction pointer or PC register.

Endian

Thor2023 is a little-endian machine. The difference between big endian and little endian is in the ordering of bytes in memory. Bits are also numbered from lowest to highest for little endian and from highest to lowest for big endian.

Shown is an example of a 32-bit word in memory.

Little Endian:

Address	3	2	1	0
Byte	3	2	1	0

Big Endian:

Address	3	2	1	0
Byte	0	1	2	3

For Thor2023 the root opcode is in byte zero of the instruction and bytes are shown from right to left in increasing order. As the following table shows.

Address 3	Address 2	Address 1	Address 0
Byte 3	Byte 2	Byte 1	Byte 0

31 24	23 16	15 8	7 5	4 0
Constant_8	Raspec_8	Rtspec_8	Sz_3	Opcode_5

Programming Model

Register File

Rn – General Purpose Registers

The register file contains 64 96-bit general purpose registers. The register file is *unified*; register may hold integer or floating-point values. The stack pointer is banked with a separate stack pointer for each operation mode.

Regno	ABI	ABI Usage	
0	0	Always zero	
1	A0	First argument / return value register	
2	A1	Second argument / return value register	
3	T0	Temporary register, caller save	
4	T1	Temporary register	
5	T2	Temporary register	
6	T3	Temporary register	
7	T4	Temporary register	
8	T5	Temporary register	
9	T6	Temporary register	
10	T7	Temporary register	
11	T8	Temporary register	
12	T9	Temporary register	
13	S0	Saved register, register variables	
14	S1	Saved register	
15	S2	Saved register	
16	S3	Saved register	
17	S4	Saved register	
18	S5	Saved register	
19	S6	Saved register	
20	S7	Saved register	
21	S8	Saved register	
22	S9	Saved register	
23	A2	Third argument register	
24	A3	Argument register	
25	A4	Argument register	
26	A5	Argument register	
27	A6	Argument register	
28	A7	Argument register	
29	A8	Argument register	
30	A9	Argument register	
31			

32	M0	Vector mask	
33	M1	Vector mask	
34	M2	Vector mask	
35	M3	Vector mask	
36	M4	Vector mask	
37	M5	Vector mask	
38	M6	Vector mask	
39	M7	Vector mask	
40			
41			
42			
43			
44			
45			
46			
47			
48			
49			
50			
51			
52			
53	PC	Program counter / Status Register	
54	SC	Stack canary; a LOAD does CCHK	
55	LC	Loop counter	
56	LR0	Subroutine link register #0; branch subroutine specific	
57	LR1	Subroutine link register #1	
58	LR2	Subroutine link register #2	
59	LR3	Subroutine link register #3	
60	GP1	Global Pointer #1	
61	GP0	Global Pointer #0	
62	FP	Frame Pointer	
63	SP	Stack Pointer	
63	ASP	Application Stack pointer	
63	SSP	System Stack pointer	

	AC	Application Control Register	

Predicate Registers

The original Thor machine had 16 four-bit dedicated predicate registers. Thor2023 by contrast stores predicate conditions in general purpose registers. Any GPR may be used to hold values used in predication. Original Thor predicates were a prefix byte containing the predicate register and condition present for every instruction. This has been superseded using the predicate instruction modifier, PRED, which allows up to eight following instructions to be predicated in the same manner. The PRED modifier is more storage efficient than predicating every instruction with predicate bits as most instructions do not require predication.

Mask Registers (m0 to m7)

Mask registers are used to mask off vector operations so that a vector instruction doesn't perform the operation on all elements of the vector. Vector instructions (loads and stores) that don't explicitly specify a mask register assume the use of mask register zero (m0). Mask registers are a subset of the general-purpose register array, allowing instructions that operate on GPRs to operate on the mask registers.

Thor2022 had dedicated mask registers leading to additional instructions required to manipulate them.

Register	Tag	Usage
m0	88	contains all ones by convention
m1	89	
m2	90	
m3	91	
m4	92	
m5	93	
m6	94	
m7	95	

Vector Length (VL register)

The vector length register controls how many elements of a vector are processed. The vector length register may not be set to a value greater than the number of elements supported by hardware. After the vector length is set a SYNC instruction should be used to ensure that following instructions will see the updated version of the length register.

Vector length has register tag #87.

15 8	7 0
0	$\text{Elements}_{7..0}$

Code Address Registers

Many architectures have registers dedicated to addressing code. Almost every modern architecture has a program counter or instruction pointer register to identify the location of instructions. Many architectures also have at least one link register or return address register holding the address of the next instruction after a subroutine call. There are also dedicated branch address registers in some architectures. These are all code addressing registers.

The original Thor lumped these registers together in a code address register array.

LRn – Link Registers

There are four registers in the Thor2023 architecture reserved for subroutine linkage. These registers are used to store the address of the calling instruction. They may be used to implement fast returns for several levels of subroutines or to used to call milli-code routines. The jump to subroutine, JSR, and branch to subroutine, BSR, instructions update a link register. The return from subroutine,. RTS, instruction may reload the program counter with an offset from the value contained in a link register. Typically, this is used to return to the next instruction.

PC – Program Counter

This register points to the currently executing instruction. The program counter increments as instructions are fetched, unless overridden by another flow control instruction.

LC - Loop Counter

The loop counter register is used in counted loops along the decrement and branch, DBcc, instruction.

SR - Status Register

The processor status register holds bits controlling the overall operation of the processor. The status register is not accessible in user mode.

31 24	23 21	20 16	15		13 12	11	10 8	7	6	5	4	3	2	1	0
CPL	~	~	T		OM		IPL			D	D	AR		RT	

CPL is the current privilege level the processor is operating at.

T indicates that trace mode is active.

S indicates the processor is in supervisor mode.

AR: Address Range indicates the number of address bits in use. 0 = near or short (32-bit) addressing is in use. When short addressing is in use only the low order 32-bit are significant and stored or loaded to or from the stack.

IPL is the interrupt mask level

RT specifies the return type for an RTI instruction.

Decimal Mode

Setting the ‘D’ flag bit 5 in the SR register sets the processor in decimal operating mode. Arithmetic operations will use BCD numbers for both source and destination operands.

Decimal mode, ‘D’ flag bit 4, may also be applied to floating-point which will use decimal floating-point operations instead of binary.

Special Purpose Registers

M_CORENO (CSR 0x3001)

This register contains a number that is externally supplied on the coreno_i input bus to represent the hardware thread id or the core number.

M_TICK (CSR 0x3002)

This register contains a tick count of the number of clock cycles that have passed since the last reset. Note that this register should not be used for precise timing as the processor's clock frequency may vary for performance and power reasons. The TIME CSR may be used for wall-clock timing as it has its own timing source.

SC - Stack Canary

This special purpose register is available in the general register file as register 54. The stack canary register is used to alleviate issues resulting from buffer overflows on the stack. The canary register contains a random value which remains consistent throughout the run-time of a program. In the right conditions, the canary register is written to the stack during the function's prolog code. In the function's epilog code, the value of the canary on stack is checked to ensure it is correct, if not a check exception occurs.

AV – Application Vector Table Address

This register holds the address of the applications vector table. The vector table must be 16-byte aligned.

63 ... 4	3 ... 0
App Vector Table Address$_{63..4}$	0

VB – Vector Base Register

The vector base register provides the location of the vector table. The vector table must be octa aligned. On reset the VBR is loaded with zero. There is a separate vector base register for each operating mode.

63 ... 3	2	1 0
Vector Table Address$_{63..3}$	~	~

Operating Modes

The core operates in one of four basic modes: application/user mode, supervisor mode, hypervisor mode or machine mode. Machine mode is switched to when an interrupt or exception occurs, or when debugging is triggered. On power-up the core is running in machine mode. An RTI instruction must be executed to leave machine mode after power-up.

A subset of instructions is limited to machine mode.

Mode Bits	Mode
0	User / App

1	Supervisor
2	Hypervisor
3	Machine

Tags

<table>
<tr><td>Tag</td><td></td></tr>
<tr><td>0</td><td>Untagged</td></tr>
<tr><td>1</td><td>Address Pointer – 20 bit size + 64 bit pointer
<table>
<tr><td>Subtype</td><td></td></tr>
<tr><td>0</td><td>Unused</td></tr>
<tr><td>1</td><td>Return address</td></tr>
<tr><td>2</td><td>Frame Pointer</td></tr>
<tr><td>3</td><td>Pointer</td></tr>
<tr><td>4 to 7</td><td>Unassigned</td></tr>
<tr><td></td><td></td></tr>
</table>
</td></tr>
<tr><td>2</td><td>Integer 96 bits</td></tr>
<tr><td>3</td><td>Integer 64 - bits</td></tr>
<tr><td>4</td><td>Integer 32 - bits</td></tr>
<tr><td>5</td><td>Integer 16 - bits</td></tr>
<tr><td>6</td><td>Integer 8 - bits</td></tr>
<tr><td>8</td><td>Float 96 bits</td></tr>
<tr><td>9</td><td>Float 64 bits</td></tr>
<tr><td>10</td><td>Float 32 bits</td></tr>
<tr><td>11</td><td>Float 16-bits</td></tr>
<tr><td>12</td><td>Float 8-bits</td></tr>
<tr><td>16</td><td>String Descriptor – 24 bit length, 64 bit virtual address pointer</td></tr>
<tr><td>17</td><td>Character data, three 32-bit characters</td></tr>
<tr><td>18</td><td>Character data, four 24-bit characters</td></tr>
<tr><td>19</td><td>Character data, 12 8-bit characters</td></tr>
<tr><td></td><td></td></tr>
<tr><td>63</td><td>Instructions 40-bit parcels</td></tr>
</table>

Exceptions

External Interrupts

There is little difference between an externally generated exception and an internally generated one. An externally caused exception will set the exception cause code for the currently fetched instruction.

There are eight priority interrupt levels for external interrupts. When an external interrupt occurs the mask level is set to the level of the current interrupt. A subsequent interrupt must exceed the mask level to be recognized.

Effect on Machine Status

The operating mode is always switched to machine mode on exception. It is up to the machine mode code to redirect the exception to a lower operating mode when desired. Further exceptions at the same or lower interrupt level are disabled automatically. Machine mode code must enable interrupts at some point.

Exception Stack

The status register, program counter, and predicate group register are pushed onto an internal stack when an exception occurs. This stack is at least 16 entries deep to allow for nested interrupts and multiply nested traps and exceptions.

Exception Table

Vector	Usage
0	Reset value for system stack pointer
1	Reset value for program counter
2	Bus Error
3	Address Error
4	Unimplemented Instruction
5	
6	
7	
8	Privilege Violation
9	Instruction trace
10	
11	Stack Canary
12 to 23	reserved
24	Spurious interrupt
25	Auto vector #1
26	Auto vector #2
27	Auto vector #3
28	Auto vector #4
29	Auto vector #5
30	Auto vector #6
31	Auto vector #7

32	Breakpoint (BRK)
33 to 63	Trap #1 to 31
	Applications Usage
64	Divide by zero
65	Overflow
65 to 511	Unassigned usage

Reset

Reset is treated as an exception. The reset routine should exit using an RTI instruction. The status register should be setup appropriately for the return.

The core begins executing instructions at address $00…00. All registers are in an undefined state.

Precision

Exceptions in Thor2023 are precise. They are processed according to program order of the instructions. If an exception occurs during the execution of an instruction, then an exception field is set in the pipeline buffer. The exception is processed when the instruction commits which happens in program order. If the instruction was executed in a speculative fashion, then no exception processing will be invoked unless the instruction makes it to the commit stage.

Instruction Set

Overview

Thor was a variable length instruction set with instructions varying in length from one to eight bytes. Thor2023 is primarily a fixed length instruction with provision for additional instruction words used for constants. Reducing the variety of instruction sizes makes implementation of decoders more economical.

Instruction Descriptions

Opcode Maps

Major Opcode

	0	1	2	3	4	5	6	7
0x	0 TRAP	1	2 {R2}	3 {CSR}	4 ADDI	5 CMPI	6 MULI	7 DIVI
	8 ANDI	9 ORI	10 EORI	11 CHK	12 {FLT2}	13 {BIT}	14 {SHIFT}	15 FMA
1x	16 LOAD	17 LOADZ	18 STORE	19 BMAP	20 FADDI	21 FCMPI	22 FMULI	23 FDIVI
	24 JSR, JMP	25 CMPXCHG	26 {AMO}	27	28 Bcc	29 DBcc	30	31 PFX / NOP

{R2} Operations

	0	1	2	3	4	5	6	7
0x	0 CNTLZ	1 CNTLO	2 CNTPOP	3 ABS	4 ADD	5 CMP	6 MUL	7 DIV
	8 AND	9 OR	10 EOR	11	12	13 CHRNDX	14 CLMUL	15 SQRT
1x	16 DIF	17 PTRDIF	18 REVBIT	19 BMAP	20	21	22 SM4ED	23 SM4KS
	24 JMP / JSR	25	26 AES64DS	27 AES64DSM	28 AES64ES	29 AES64ESM	30 AES64KS1I	31 AES64KS2
2x	32 PRED	33 CARRY	34 VMASK	35 ATOM	36 ROUND	37	38	39
	40 V2BITS	41 BITS2V	42 VEX	43 VEINS	44 VGNDX	45	46	47
3x	48 MIN	49 MAX	50 BMM	51 MUX	52	53 AES64IM	54 SM3P0	55 SM3P1
	56 SHA256 SIG0	57 SHA256 SIG1	58 SHA256 SUM0	59 SHA256 SUM1	60 SHA512 SIG0	61 SHA512 SIG1	62 SHA512 SUM0	63 SHA512 SUM1

{SHIFT}

	0	1	2	3	4	5	6	7
0x	0 ASL	1 ASR	2 LSL	3 LSR	4 ROL	5 ROR	6	7
	8 ZXB	9 SXB	10	11	12	13	14	15
1x	16 VSHLV	17 VSHRV	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
2x	32 ASLI	33 ASRI	34 LSLI	35 LSRI	36 ROLI	37 RORI	38	39
	40 ZXBI	41 SXBI	42	43	44	45	46	47
3x	48 VSHLVI	49 VSHRVI	50	51	52	53	54	55
	56	57	58	59	60	61	62	63

{FLT2} Operations

	0	1	2	3	4	5	6	7
0x	0 FSCALEB	1 {FLT1}	2 FMIN	3 FMAX	4 FADD	5 FCMP	6 FMUL	7 FDIV
	8 FSEQ	9 FSLT	10 FSLE	11 FSNE	12	13	14 FNXT	15 FREM
1x	16							
	24							

{FLT1} Operations

	0	1	2	3	4	5	6	7
0x	0	1	2 FOTI	3 ITOF	4	5	6 FSIGN	7 FSIG
	8 FSQRT	9 FS2D	10 FS2T	11 FD2T	12	13	14 ISNAN	15 FINITE
1x	16	17	18	19	20	21 FTRUNC	22	23 FRES
	24	25 FD2S	26 FT2S	27 FT2D	28	29	30 FCLASS	31
2x	32 FABS	33	34 FNEG	35	36	37	38	39
	40							
3x	48							
	56							

{AMO} Operations

	0	1	2	3	4	5	6	7
0x	0 SWAP	1	2 MIN	3 MAX	4 ADD	5	6 ASL	7 LSR
	8 AND	9 OR	10 EOR	11	12 MINU	13 MAXU	14	15 CAS
1x	16 SWAPI	17	18 MIN	19 MAX	20 ADDI	21	22 ASLI	23 LSRI
	24 ANDI	25 ORI	26 EORI	27	28 MINU	29 MAXU	30	31 CAS

Operand Swapping

Many instructions allow first and second source operands to be swapped. This is indicated by the swap 'S' bit in the instruction. This is particularly useful for instructions that are non-commutative like SUB and DIV.

Operand Swap

Operand Order	**S**
Normal	0
1^{st} and 2^{nd} Swapped	1

Operand Sizes

Many instructions support four different operand sizes: byte, wyde, tetra and octa. The operand size is selected by suffixing the mnemonic with 'b' for byte, 'w' for wyde, 't' for tetra and 'o' for octa.

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa
4	.c	24-bit
5	.p	40-bit Penta
6	.n	96-bit
7		reserved

Arithmetic Operations

Representations

Int:96

95 0

96 bits

Int:64

63 0

64 bits

Int:40

39 0

40 bits

Int:32

31 0

32 bits

Int:16

15 0

16 bits

Int:8

7 0

8 bits

ABS – Absolute Value

Description:

This instruction computes the absolute value of the contents of the source operand and places the result in Rt.

Supported Operand Sizes: .b, .w, .t, .o

Integer Instruction Format: R2

ABS Rt, Ra – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
3_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

Operation:

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

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

ADD - Addition

Description:

Add two source operands and place the sum in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values unless the decimal mode flag is set in which case values are treated as BCD numbers. This instruction may be used with the CARRY modifier to perform extended precision addition.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra + Rb or Rt = Ra + Imm

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

ADD Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

ADD Rt,Ra,Imm$_{15}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	4_5

Clock Cycles: 1

AND – Bitwise And

Description:

Bitwise ‘and’ two source operands and place the result in the target register. The one’s complement of operands may be used by setting the appropriate ‘S’ bit in the instruction.

Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Clock Cycles: 1

Operation:

Rt = Ra & Rb or Rt = Ra & Imm

Instruction Formats:

AND Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
8_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

AND Rt,Ra,Imm$_{15}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	8_5

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

BMAP – Byte Map

Description:

First the target register is cleared, then bytes are mapped from the 12-byte source Ra into bytes in the target register. This instruction may be used to permute the bytes in register Ra and store the result in Rt. This instruction may also pack bytes, wydes or tetras. The map is determined by the low order 48-bits of register Rb or a 48-bit immediate constant. Bytes which are not mapped will end up as zero in the target register.

Instruction Formats:

BMAP Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
19_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

BMAP Rt,Ra,Imm$_{48}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$\sim_7$	S	$\sim_9$	Sa	Ra_6	St	Rt_6	V	Sz_3	19_5
Immediate$_{31..0}$								0_3	31_5
$\sim_{16}$			Imm$_{47..32}$					1_3	31_5

Clock Cycles: 1

Operation:

Vector Operation

Execution Units: First Integer ALU

Clock Cycles: 1

Exceptions: none

Notes:

CHK – Check Register Against Bounds

Description:

A register is compared to two values. If the register is outside of the bounds defined by Rb and an immediate value then an exception will occur. Ra must be greater than or equal to Rb and Ra must be less than the immediate.

Instruction Formats:

CHK Ra, Rb, Cn – Register direct

39	38	37 32	31	30	29	28 23	22	21 16	15	14 12	11 9	8	7 5	4 0
~	Vc	Rc_6	0	Vb	Sb	Rb_6	Sa	Ra_6	~	$\sim_3$	Cn_3	V	Sz_3	11_5

Clock Cycles: 1

cn_3	exception when not
0	Ra >= Rb and Ra < Rc
1	Ra >= Rb and Ra <= Rc
2	Ra > Rb and Ra < Rc
3	Ra > Rb and Ra <= Rc
4	not (Ra >= Rb and Ra < Rc)
5	not (Ra >= Rb and Ra <= Rc)
6	not (Ra > Rb and Ra < Rc)
7	not (Ra > Rb and Ra <= Rc)

CHKI Ra, Imm, Cn

39 32	31	30	29	28 23	22	21 16	15 12	11 9	8	7 5	4 0
$Imm_{11..4}$	1	Vb	Sb	Rb_6	Sa	Ra_6	$Imm_{3..0}$	Cn_3	V	Sz_3	11_5

Clock Cycles: 1

cn_3	exception when not
0	Ra >= Rb and Ra < Imm
1	Ra >= Rb and Ra <= Imm
2	Ra > Rb and Ra < Imm
3	Ra > Rb and Ra <= Imm
4	not (Ra >= Rb and Ra < Imm)
5	not (Ra >= Rb and Ra <= Imm)
6	not (Ra > Rb and Ra < Imm)
7	not (Ra > Rb and Ra <= Imm)

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: bounds check

Notes:

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

CLMUL – Carry-less Multiply

Description:

Compute the low order product bits of a carry-less multiply.

Instruction Formats:

CLMUL Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 4

CLMUL Rt,Ra,Imm$_8$

39 34	33 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
4_6	$\sim_2$	1	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 4

Exceptions: none

Execution Units: First Integer ALU

Operations

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]

Exceptions: none

CMP - Comparison

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as signed and unsigned integers.

Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Operation:

Rt = Ra ? Rb or Rt = Ra ? Imm or Rt = Imm ? Ra

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Formats:

ADD Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
5_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

ADD Rt,Ra,Imm$_{15}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	5_5

Clock Cycles: 1

Rt bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	
1	LT	< less than	
2	LE	<= less than or equal	
3	LO / CS	< unsigned less than	
4	LS	<= unsigned less than or equal	
5	AND	And	
6	OR	Or	
7	T	1	
8	NE	< > not equal	
9	GE	>= greater than or equal	
10	GT	> greater than	
11	HS / CC	unsigned greater than or equal	
12	HI	unsigned greater than	
13	NAND	nand	
14	NOR	Nor	
15	SR	Branch subroutine	

CNTLZ – Count Leading Zeros

Description:

This instruction counts the number of consecutive zero bits beginning at the most significant bit towards the least significant bit.

Supported Operand Sizes: .b, .w, .t, .o

Integer Instruction Format: R1

CNTLZ Rt, Ra, Rb – Register direct

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
0_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTPOP – Count Population

Description:

This instruction counts the number of bits set in a register.

Supported Operand Sizes: .b, .w, .t, .o

Integer Instruction Format: R1

CNTPOP Rt, Ra, Rb – Register direct

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
2_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CSR – Control and Special Registers Operations

Description:

Perform an operation on a CSR.

Operation	**Op$_3$**	
Read CSR	0	
Write CSR	1	
Or to CSR (set bits)	2	
And complement to CSR (clear bits)	3	
Exclusive Or to CSR (flip bits)	4	

Supported Operand Sizes: N/A

Regno		
$000	reserved	Not used
$002	sr	Status register (privileged)
$120	Tick	Tick count (read only)
$121	Coreno	Core number (read only) (privileged)
$127		

Instruction Formats:

OR Rt, Ra, CSR

ANDC Rt, Ra, CSR

EOR Rt, Ra, CSR

CSR Rt,Ra,#Regno$_{12}$

39 38	37 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
0	Regno$_{13..8}$	S	Regno$_{7..0}$	Sa	Ra$_6$	St	Rt$_6$	V	Op$_3$	3$_5$

Clock Cycles: 1

CSR Rt, #Regno$_{12}$, #Imm

39 38	37 32	31	30 23	22 16	15	14 9	8	7 5	4 0
1	Regno$_{13..8}$	S	Regno$_{7..0}$	Imm$_7$	St	Rt$_6$	V	Op$_3$	3$_5$

DIVS – Signed Division

Description:

Divide source dividend operand by divisor operand and place the quotient in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra / Rb or Rt = Ra / Imm or Rt = Imm / Ra

Instruction Formats:

DIVS Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
7_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 100

DIVS Rt,Ra,Imm$_{16}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	7_5

Clock Cycles: 100

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

DIVU – Unsigned Division

Description:

Divide source dividend operand by divisor operand and place the sum in the target register. All registers are integer registers. Arithmetic is unsigned twos-complement values.

Immediate mode is not available for this instruction.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra / Rb or Rt = Ra / Imm or Rt = Imm / Ra

Instruction Formats:

DIVU Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
7_6	1_2	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 100

Execution Units: All Integer ALU's

Exceptions: none

Notes:

EOR – Bitwise Exclusive Or

Description:

Bitwise exclusive ‘or’ two source operands and place the sum in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Operation:

Rt = Ra ^ Rb or Rt = Ra ^ Imm

Instruction Formats:

EOR Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
10_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

EOR Rt,Ra,Imm$_{15}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	10_5

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ENOR – Bitwise Exclusive Nor

Description:

Bitwise exclusive 'nor' two source operands and place the result in the target register.

Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Operation:

Rt = ~(Ra ^ Rb) or Rt = ~(Ra ^ Imm)

Instruction Formats:

ENOR Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
10_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	1	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

ENOR Rt,Ra,Imm$_{15}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	1	Rt_6	V	Sz_3	10_5

Clock Cycles: 1

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

PFX – Constant Postfix

Description:

The PFX instruction postfix is used to build large constants for use in the preceding instruction as the immediate constant for the instruction. There are three postfix instructions which extend the constant from different bit locations. They should be used in the order PFX0, PFX1. A postfix may be omitted if the omitted bits match what would be included.

Postfixes are normally caught at the decode stage and do not progress further in the pipeline. They are treated as a NOP instruction.

Supported Operand Sizes: N/A

Instruction Format:

This format extends the constant from bit 0 with the 32 bits specified in the instruction and sign extends the value to the width of the constant prefix buffer.

39 8	7 5	4 0
$Immediate_{32}$	0_3	31_5

Instruction Format:

This format extends the previous constant value by 32 bits beginning at bit 32 and sign extends the value to the width of the machine.

39 8	7 5	4 0
$Immediate_{32}$	1_3	31_5

Instruction Format:

This format extends the previous constant value by 32 bits beginning at bit 64 and sign extends the value to the width of the machine.

39 8	7 5	4 0
$Immediate_{32}$	2_3	31_5

MULS – Multiply Signed

Description:

Multiply two source operands and place the sum in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values. The ‘S’ flag indicates to perform an unsigned multiply.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra * Rb or Rt = Ra * Imm

Instruction Formats:

MULS Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
6_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 12

MULS Rt,Ra,Imm$_{16}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	6_5

Clock Cycles: 12

Clock Cycles: 12

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

MULU – Unsigned Multiplication

Description:

Multiply two source operands and place the product in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values. The ‘S’ flag indicates to perform an unsigned multiply. Unsigned multiply can be used during index calculations.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra * Rb or Rt = Ra * Imm

Instruction Formats:

MULU Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
6_6	$\sim_2$	1	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 12

MULU Rt,Ra,Imm$_{16}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	1	$Imm_{8..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	6_5

Clock Cycles: 12

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

NAND – Bitwise And and Invert

Description:

Bitwise ‘nand’ two source operands and place the result in the target register.

Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Clock Cycles: 1

Operation:

Rt = ~(Ra & Rb)

Instruction Formats:

NAND Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
8_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	1	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

NAND Rt,Ra,Imm$_{15}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	1	Rt_6	V	Sz_3	8_5

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

NOR – Bitwise Or and Invert

Description:

Bitwise ‘or’ two source operands invert the result and place the result in the target register. All registers are integer registers.

Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Operation:

Rt = ~(Ra | Rb)

Instruction Formats:

NOR Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
9_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	1	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

NOR Rt,Ra,Imm$_{15}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	1	Rt_6	V	Sz_3	9_5

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

OR – Bitwise Or

Description:

Bitwise ‘or’ two source operands and place the sum in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Operation:

Rt = Ra | Rb or Rt = Ra | Imm

Instruction Formats:

OR Rt, Ra, Rb – Register direct

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
9_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

OR Rt,Ra,Imm$_{15}$

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..9}$	S	$Imm_{8..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	9_5

Clock Cycles: 1

Clock Cycles: 2

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

REVBIT – Reverse Bit Order

Description:

This instruction reverses the order of bits in Ra and stores the result in Rt.

Integer Instruction Format: R2

REVBIT Rt, Ra – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
18_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

Operation:

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

SQRT – Square Root

Description:

This instruction computes the square root value of the contents of the source operand and places the result in Rt.

Supported Operand Sizes: .b, .w, .t, .o

Integer Instruction Format: R2

SQRT Rt, Ra – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
15_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

Operation:

Rt = SQRT(Ra)

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

Floating-Point Operations

Precision

Floating point operations are always performed at the greatest precision available. Lower precision formats are available for storage.

For decimal floating-point three storage formats are supported. 96-bit triple precision, 64-bit double precision, and 32-bit single precision values.

Representations

Binary Floats

Triple Precision, Float:96

The core uses a 96-bit triple precision binary floating-point representation.

96-bit values are more compact than 128-bit ones which reduces the amount of hardware required and data being transferred. They have enough significant digits for a wide variety of applications. 64-bit values are not sufficient for some applications. The question then is how much larger of a representation to use. 80-bits is popular, offering about 19 significant digits which is good for a wide variety of applications. 96-bit floats offer about 24 significant digits.

95	94 80	79 0
S	Exponent$_{15}$	Significand$_{80}$

Double Precision, Float:64

63	62 52	51 0
S	Exponent$_{11}$	Significand$_{52}$

Single Precision, Float:32

31	30 23	22 0
S	Exponent$_{8}$	Significand$_{23}$

Half Precision, Float:16

15	14 10	9 0
S	Exponent$_{5}$	Significand$_{10}$

Decimal Floats

The core uses a 96-bit densely packed decimal triple precision floating-point representation.

95	94 90	89 80	79 0
S	Combo$_5$	Exponent$_{10}$	Significand$_{80}$

The significand stores 25 densely packed decimal digits. One whole digit before the decimal point.

The exponent is a power of ten as a binary number with an offset of 1535. Range is 10^{-1535} to 10^{1536}

64-bit double precision decimal floating point:

63	62 58	57 50	49 0
S	Combo$_5$	Exponent$_8$	Significand$_{50}$

The significand stores 16 DPD digits. One whole digit before the decimal point.

32-bit single precision decimal floating point:

31	30 26	25 20	19 0
S	Combo$_5$	Exponent$_6$	Significand$_{20}$

The significand store 7 DPD digits. One whole digit before the decimal point.

Rounding Modes

Binary Float Rounding Modes

Rm3	Rounding Mode
000	Round to nearest ties to even
001	Round to zero (truncate)
010	Round towards plus infinity
011	Round towards minus infinity
100	Round to nearest ties away from zero
101	Reserved
110	Reserved
111	Use rounding mode in float control register

Decimal Float Rounding Modes

Rm3	Rounding Mode
000	Round ceiling
001	Round floor
010	Round half up
011	Round half even
100	Round down
101	Reserved
110	Reserved
111	Use rounding mode in float control register

Operand Sizes

Sz_3	Ext.	Operand
0	.h	16-bit half
1	.s	32-bit single
2	.d	64-bit double
3	.t	96-bit triple
4		Reserved
5		reserved
6		reserved
7		reserved

FABS – Absolute Value

Description:

This instruction computes the absolute value of the contents of the source operand and places the result in Rt. The sign bit of the value is cleared. No rounding occurs.

Integer Instruction Format: R1

FABS Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_5	$\sim_3$	0	0	0	32_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 1

Operation:

FPt = Abs(FPa)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FADD –Float Addition

Description:

Add two source operands and place the sum in the target register. All registers values are treated as 96-bit floating-point values. An immediate value is converted to 96-bit triple precision from half, single, or double precision.

Supported Operand Sizes:

Operation:

Rt = Ra + Rb or Rt = Ra + Imm

Clock Cycles: 8

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Formats:

FADD Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_5	Rm_3	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

FADD Rt,Ra,Imm$_{16}$

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5

FADD Rt,Ra,Imm$_{32}$

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$\sim_8$	S	$\sim_8$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5
$Immediate_{32}$								0_3	31_5

FADD Rt,Ra,Imm$_{64}$

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$\sim_8$	S	$\sim_8$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5
$Immediate_{31..0}$								0_3	31_5
$Immediate_{63..32}$								1_3	31_5

FADD Rt,Ra,Imm$_{64}$

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$\sim_8$	S	$\sim_8$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5
$Immediate_{31..0}$								0_3	31_5
$Immediate_{63..32}$								1_3	31_5
$Immediate_{95..64}$								2_3	31_5

FCMP - Comparison

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as floating-point values. This instruction may compare against lower precision immediate values to conserve code space.

Supported Operand Sizes:

Operation:

Rt = Ra ? Rb or Rt = Ra ? Imm or Rt = Imm ? Ra

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

FCMP Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
5_5	$\sim_3$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 1

FCMP Rt,Ra,Imm$_{13}$

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	21_5

Clock Cycles: 1

Rt bit	Mnem.	Meaning	Test
		Float Compare Results	
0	EQ	equal	!nan & eq
1	NE	not equal	!eq
2	GT	greater than	!nan & !eq & !lt & !inf
3	UGT	Unordered or greater than	Nan || (!eq & !lt & !inf)
4	GE	greater than or equal	Eq || (!nan & !lt & !inf)
5	UGE	Unordered or greater than or equal	Nan || (!lt || eq)
6	LT	Less than	Lt & (!nan & !inf & !eq)
7	ULT	Unordered or less than	Nan | (!eq & lt)
8	LE	Less than or equal	Eq | (lt & !nan)
9	ULE	unordered less than or equal	Nan | (eq | lt)
10	GL	Greater than or less than	!nan & (!eq & !inf)
11	UGL	Unordered or greater than or less than	Nan | !eq
12	ORD	Greater than less than or equal / ordered	!nan
13	UN	Unordered	Nan
14			
15			

FDIV –Float Division

Description:

Divide two source operands and place the quotient in the target register. All registers values are treated as 96-bit floating-point values.

Supported Operand Sizes:

Operation:

Rt = Ra / Rb or Rt = Ra / Imm or Rt = Imm / Ra

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Formats:

FDIV Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
7_5	Rm_3	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 150

FDIV Rt,Ra,Imm$_{13}$

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	23_5

Clock Cycles: 150

FMUL –Float Multiplication

Description:

Multiply two source operands and place the product in the target register. All registers values are treated as 96-bit floating-point values.

Supported Operand Sizes:

Operation:

Rt = Ra * Rb or Rt = Ra * Imm

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

FDIV Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
6_5	Rm_3	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 8

FDIV Rt,Ra,Imm$_{13}$

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	22_5

Clock Cycles: 8

FNEG – Negate Value

Description:

This instruction computes the negative value of the contents of the source operand and places the result in Rt. The sign bit of the value is inverted. No rounding occurs.

Integer Instruction Format: R1

FNEG Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_5	$\sim_3$	0	0	0	34_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 1

Operation:

Rt = -Ra

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FSCALEB –Scale Exponent

Description:

Add the source operand to the exponent.

Supported Operand Sizes:

Operation:

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Formats:

FSCALEB Rt, Ra, Rb – Register direct

39 36	35 33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
0_4	0_3	0	Vb	Sb	Rb_6	Va	Sa	Ra_6	Vt	St	Rt_6	Sz_3	12_5

Clock Cycles: 1

FSCALEB Rt, Ra, #Imm – Immediate

39 36	35 33	32	31 24	23	22	21 16	15	14	13 8	7 5	4 0
0_4	1_3	0	Imm_8	Va	Sa	Ra_6	Vt	St	Rt_6	Sz_3	12_5

$\sim_{16}$	$Immediate_{15..0}$	0_3	31_5

Clock Cycles: 1

FSUB –Float Subtraction

Description:

Subtract two source operands and place the difference in the target register. All registers values are treated as 88-bit floating-point values. This is an alternate mnemonic for the FADD instruction where the second source operand, Rb is assumed negated.

Supported Operand Sizes:

Operation:

Rt = Ra + -Rb or Rt = Ra + -Imm

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

FSUB Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_5	Rm_3	0	Vb	1	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 8

FSUB Rt,Ra,Imm$_{13}$

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5

Clock Cycles: 8

s

FTRUNC – Truncate Fraction

Description:

This instruction truncates off the fractional portion of the number leaving only the integer portion. No rounding occurs.

Integer Instruction Format: R1

FTRUNC Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_5	$\sim_3$	0	0	0	21_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 1

Operation:

Rt = Trunc(Ra)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

String Operations

Representations

Strings

95 92	91 64	63 0
Typ	Length$_{28}$	Pointer$_{64}$

UTF8 Chars

95 0
12 characters

UTF24 Chars

95 0
4 characters

CHRNDX – Character Index

Description:

This instruction searches Ra, which is treated as an array of characters, for a character value specified by Rb and places the index of the character into the target register Rt. If the character 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 +11 for UTF8 characters or -1 to +3 for UTF24 characters. The index of the first found byte is returned (closest to zero).

Supported Operand Sizes: .b, .c

Instruction Formats:

CHRNDX Rt, Ra, Rb – Register direct

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
13_6	~	0	Vb	Sb	Rb_6	Va	Sa	Ra_6	Vt	St	Rt_6	Sz_3	2_5

Clock Cycles: 1

CHRNDX Rt,Ra,Imm$_{15}$

39 34	33	32	31 24	23	22	21 16	15	14	13 8	7 5	4 0
13_6	~	1	Imm_8	Va	Sa	Ra_6	Vt	St	Rt_6	Sz_3	2_5

Clock Cycles: 1

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Bit Manipulation Operations

BCLR – Clear Bit

Description:

A bit in the source operand is cleared and the result placed in the target register. The specified bit to clear is modulo the operand size. The previous value of the bit is available with the CARRY modifier.

Supported Operand Sizes: .b, .w, .t, .o

Flag Updates: none

Operation:

Rt = Ra &~bit Rb or Ra = Ra &~bit imm

Instruction Formats:

BCLR Ct, Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
0_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

BCLR Ct, Rt, Ra,Imm$_7$

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
32_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

BCHG – Change Bit

Description:

A bit in the source operand is changed and placed in the target register. The specified bit to change is modulo the operand size.

Supported Operand Sizes: .b, .w, .t, .o

Flag Updates: none

Operation:

Rt[Rb] = ~Ra[Rb] or Rt[Imm] = ~Ra[Imm]

Instruction Formats:

BCHG Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
2_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

BCHG Rt, Ra,Imm$_7$

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
34_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

BPCHG – Change Bit Pair

Description:

A bit pair in the source operand is changed and placed in the target register. The pair is exclusively or'd with 11b. There are four bit-pairs per byte indicated as pair #0 to #3. The bit pair specified is taken modulo the operand size.

Bit Pair Value	Updated Value
00	11
01	10
10	01
11	00

Supported Operand Sizes: .b, .w, .t, .o

Flag Updates: none

Operation:

Rt[Rb] = ~Ra[Rb] or Rt[Imm] = ~Ra[Imm]

Instruction Formats:

BPCHG Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
6_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

BPCHG Rt, Ra,Imm$_7$

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
38_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

BPCLR – Clear Bit Pair

Description:

A pair of bits in the source operand is cleared and the result placed in the target register.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra &~bit Rb or Ra = Ra &~bit imm

Instruction Formats:

BPCLR Ct, Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

BPCLR Ct, Rt, Ra,Imm$_7$

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
36_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

BPTST – Test Bit Pair

Description:

Test a bit pair in the source operand and place the bit status in the flags of the predicate register. The bit tested is modulo the operation size. All combinations of bit pair value may be detected via flags.

Supported Operand Sizes: .b, .w, .t, .o

Flag Updates:

Predicate Register Pt_3 is always updated.

Bit Pair Value	Flag Setting
00	Zero flag is set, cf, nf, and vf are cleared
01	Carry flag is set, zf, nf, and vf are cleared
10	Negative flag is set, zf, cf, and vf are cleared
11	Overflow flags is set, nf, zf and cf are cleared

If the bit pair is zero the zero flag is set otherwise it is cleared.
If the bit pair is
The negative flag is set to the value of the high order bit of the pair.
The overflow flag is set to the exclusive or of the two bits in the pair.

Operation:

Instruction Formats:

BPTST Rt, Ra, Rb

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
7_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

BPTST Rt, Ra,Imm$_7$

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
39_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

BSET – Set Bit

Description:

A bit in the source operand is set and placed in the target register.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra | bit Rb or Rt = Ra or Bit[Imm]

Instruction Formats:

BSET Ct, Rt, Ra, Rb

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

BSET Ct, Rt, Ra,Imm$_7$

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
33_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

BTST – Test Bit

Description:

Test a bit in the source operand and place the bit status in the target register. The bit tested is modulo the operation size.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra[Rb] or Rt = Ra[Imm]

Instruction Formats:

BTST Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
3_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

BTST Rt, Ra,Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
35_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Shift and Rotate Operations

ASL – Arithmetic Shift Left

Description:

Shift the first source operand to the left by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. The least significant bit is filled with the value of 'N' specified in the instruction.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra << Rb or Rt = Ra << Imm

Instruction Formats:

ASL Rt, Ra, Rb

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
0_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ASL Rt, Ra, Imm_7

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
32_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ASR – Arithmetic Shift Right

Description:

Shift the first source operand to the right, preserving the sign bit, by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values.

Supported Operand Sizes: .b, .w, .l

Operation:

Rt = Ra >> Rb or Rt = Ra >> Imm

Instruction Formats:

ASR Rt, Ra, Rb

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ASR Rt, Ra, Imm$_7$

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
33_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

SBX – Sign Bit Extend

Description:

Sign extend a value beginning at a specified bit to the width of the register and place the result in the target register. All registers are integer registers.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Instruction Formats:

SXB Rt, Ra, Rb

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
9_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

SXB Rt, Ra, Imm_7

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
41_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

LSL – Logical Shift Left

Description:

Shift the first source operand to the left by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. Fill the least significant bit with the value specified by ‘N’ in the instruction.

Supported Operand Sizes: .b, .w, .l

Operation:

Rt = Ra << Rb or Rt = Ra << Imm

Instruction Formats:

LSL Rt, Ra, Rb

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
2_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

LSL Rt, Ra, Imm$_7$

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
34_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

LSR – Logical Shift Right

Description:

Shift the first source operand to the right by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. Fill the least significant bit with the value specified by ‘N’ in the instruction.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra >> Rb or Rt = Ra >> Imm

Instruction Formats:

LSR Rt, Ra, Rb

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
3_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

LSR Rt, Ra, Imm_7

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
35_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ROL – Rotate Left

Description:

Rotate the first source operand to the left by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. The least significant bit is set to the value of the most significant bit exclusively or'd with the value 'N' from the instruction.

Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra << Rb or Rt = Ra << Imm

Instruction Formats:

ROL Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ROL Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
36_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ROR – Rotate Right

Description:

Rotate the first source operand through the carry to the right by the number of bits specified by the second source operand and place the result in the target register. All registers are integer registers. Arithmetic is signed twos-complement values. The most significant bit is set to the value of the least significant bit exclusively or'd with the value 'N' from the instruction.

Supported Operand Sizes: .b, .w, .l

Operation:

Rt = Ra >> Rb or Rt = Ra >> Imm

Instruction Formats:

ROR Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
5_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ROR Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
37_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

VSHLV – Shift Vector Left

Description

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

Instruction Formats:

VSHLV Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
16_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

VSHLV Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
48_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Operation

Amt = Rb

For x = VL-1 to Amt

Vt[x] = Va[x-amt]

For x = Amt-1 to 0

Vt[x] = 0

Exceptions: none

VSHRV – Shift Vector Right

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.

VSHLR Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
17_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

VSHLR Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
49_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Operation

Amt = Rb

For x = 0 to VL-Amt

Vt[x] = Va[x+amt]

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

Vt[x] = 0

Exceptions: none

ZBX – Zero Bit Extend

Description:

Zero extend a value beginning at a specified bit to the width of the register and place the result in the target register. All registers are integer registers.

Supported Operand Sizes: .b, .w, .l

Operation:

Rt = Zero Extend(Ra)

Instruction Formats:

ZXB Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
8_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ZXB Rt, Ra, Imm$_7$

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
40_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Flow Control Instructions

Bcc – Conditional Branch

Bcc Pn, label

Description:

Branch if the predicate condition is met. The displacement is relative to the address of the branch instruction. The branch range is +/- 8MB.

Instruction Format:

39 … 16	15 … 10	9 … 5	4 … 0
Disp$_{23..0}$	Rn$_6$	Cond$_5$	28$_5$

<table>
<tr><th>Cond$_5$</th><th>Mnem.</th><th>Meaning</th><th>Test</th></tr>
<tr><td></td><td></td><td>Integer Compare Results</td><td></td></tr>
<tr><td>0</td><td>EQ</td><td>= equal</td><td></td></tr>
<tr><td>1</td><td>LT</td><td>< less than</td><td></td></tr>
<tr><td>2</td><td>LE</td><td><= less than or equal</td><td></td></tr>
<tr><td>3</td><td>LO / CS</td><td>< unsigned less than</td><td></td></tr>
<tr><td>4</td><td>LS</td><td><= unsigned less than or equal</td><td></td></tr>
<tr><td>5</td><td>ODD</td><td>Odd</td><td></td></tr>
<tr><td>6</td><td>Z</td><td>0</td><td></td></tr>
<tr><td>7</td><td>MI</td><td>< 0</td><td></td></tr>
<tr><td>8</td><td>NE</td><td>< > not equal</td><td></td></tr>
<tr><td>9</td><td>GE</td><td>>= greater than or equal</td><td></td></tr>
<tr><td>10</td><td>GT</td><td>> greater than</td><td></td></tr>
<tr><td>11</td><td>HS / CC</td><td>unsigned greater than or equal</td><td></td></tr>
<tr><td>12</td><td>HI</td><td>unsigned greater than</td><td></td></tr>
<tr><td>13</td><td>EVEN</td><td>Even</td><td></td></tr>
<tr><td>14</td><td>NZ</td><td>Not 0</td><td></td></tr>
<tr><td>15</td><td>SR</td><td>Branch subroutine</td><td></td></tr>
<tr><th>Cond$_5$</th><th>Mnem.</th><th>Meaning</th><th>Test</th></tr>
<tr><td></td><td></td><td>Float Compare Results</td><td></td></tr>
<tr><td>16</td><td>EQ</td><td>equal</td><td>!nan & eq</td></tr>
<tr><td>17</td><td>NE</td><td>not equal</td><td>!eq</td></tr>
<tr><td>18</td><td>GT</td><td>greater than</td><td>!nan & !eq & !lt & !inf</td></tr>
<tr><td>19</td><td>UGT</td><td>Unordered or greater than</td><td>Nan || (!eq & !lt & !inf)</td></tr>
<tr><td>20</td><td>GE</td><td>greater than or equal</td><td>Eq || (!nan & !lt & !inf)</td></tr>
<tr><td>21</td><td>UGE</td><td>Unordered or greater than or equal</td><td>Nan || (!lt || eq)</td></tr>
<tr><td>22</td><td>LT</td><td>Less than</td><td>Lt & (!nan & !inf & !eq)</td></tr>
<tr><td>23</td><td>ULT</td><td>Unordered or less than</td><td>Nan | (!eq & lt)</td></tr>
<tr><td>24</td><td>LE</td><td>Less than or equal</td><td>Eq | (lt & !nan)</td></tr>
<tr><td>25</td><td>ULE</td><td>unordered less than or equal</td><td>Nan | (eq | lt)</td></tr>
<tr><td>26</td><td>GL</td><td>Greater than or less than</td><td>!nan & (!eq & !inf)</td></tr>
<tr><td>27</td><td>UGL</td><td>Unordered or greater than or less than</td><td>Nan | !eq</td></tr>
<tr><td>28</td><td>ORD</td><td>Greater than less than or equal / ordered</td><td>!nan</td></tr>
<tr><td>29</td><td>UN</td><td>Unordered</td><td>Nan</td></tr>
<tr><td>30</td><td></td><td></td><td></td></tr>
<tr><td>31</td><td></td><td></td><td></td></tr>
</table>

Clock Cycles: 4

BRA – Unconditional Branch

Description:

Unconditionally branch to a new program address. The displacement is relative to the address of the branch instruction. The branch range is +/- 64MB.

Instruction Format:

39 16	15 10	9 5	4 0
$Disp_{23..0}$	$\sim_6$	7_5	28_5

Clock Cycles: 3

BRK – Breakpoint

Description:

Execute the breakpoint exception. This is a form of the TRAP instruction.

Instruction Format:

39 5	4 0
0	0_5

BSR – Branch to Subroutine

Description:

Branch to a subroutine placing the address of the next instruction in a register. The displacement is relative to the address of the branch instruction. The branch range is +/- 8MB.

Instruction Format:

39 16	15 10	9 5	4 0
$Disp_{23..0}$	Rt_6	15_5	28_5

Clock Cycles: 3

DBcc – Decrement and Branch

DBcc Rn, label

Description:

Decrement the loop counter and branch if the condition is false and the loop counter is not equal to minus one. The displacement is relative to the address of the branch instruction. The branch range is +/- 8MB.

Instruction Format:

39 16	15 10	9 5	4 0
$Disp_{23..0}$	Rn_6	$Cond_5$	29_5

JMP – Jump to Address

Description:

Compute the effective address and jump to it. If Ra=53 then the program counter is used.

Flag Updates:

None.

Operation:

PC = Ra + Rb or PC = Ra + Imm

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Formats:

JMP (Ra, Rb) – Register direct

39 34	33 29	28	27	26 21	20	19 14	13	12 7	6 5	4 0
24_6	$\sim_5$	0	Sb	Rb_6	Sa	Ra_6	0	0_6	Sz_2	2_5

Clock Cycles: 1

JMP Imm_{18} (Ra)

39 29	28	27	26 21	20	19 14	13	12 7	6 5	4 0
$Imm_{17..7}$	S	I_6	$Imm_{5..0}$	Sa	Ra_6	0	0_6	Sz_2	24_5

Clock Cycles: 1

JSR – Jump to Subroutine

Description:

Compute the effective address and jump to it. The address of the instruction is stored in a register. If Ra=53 then the program counter is used.

Flag Updates:

None.

Operation:

Rt = PC

PC = Ra + Rb or PC = Ra + Imm

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Formats:

JSR (Ra, Rb) – Register direct

39 34	33 29	28	27	26 21	20	19 14	13	12 7	6 5	4 0
24_6	$\sim_5$	0	Sb	Rb_6	Sa	Ra_6	0	Rt_6	Sz_2	2_5

Clock Cycles: 1

JSR Imm_{18} (Ra)

39 29	28	27	26 21	20	19 14	13	12 7	6 5	4 0
$Imm_{17..7}$	S	I_6	$Imm_{5..0}$	Sa	Ra_6	0	Rt_6	Sz_2	24_5

Clock Cycles: 1

NOP – No Operation

NOP

Description:

This instruction does not perform any operation. Ty_3 0 to 3 indicates a postfix instruction, and these codes should not be used for other NOPs.

Instruction Format:

39 5	7 5	4 0
$Payload_{32}$	Ty_3	31_5

RTE – Return From Exception

Instruction Formats:

RTE #Rpt

39 34	33 31	3029	28	27	26 21	20	19 14	13	12 7	6 5	4 0
4_6	$\sim_3$	$\sim_2$	0	~	$\sim_6$	~	$\sim_6$	R_6	Rpt_6	D_2	1_5

Field Description:

Rpt_7 is the number of bytes to skip past the return address. This is to allow inline subroutine arguments. Up to 128 bytes may be skipped over. For externally triggered interrupts this field should be zero.

D_2 specifies the number of internal stack entries to unstack. It may be used to perform a multi-level return. Legal values for D are 1,2 or 3. In most cases a single entry is unstacked. If two entries are unstack a two-up level return will occur.

Operation:

Optionally pop the status register, condition code group register, and program counter from the internal stack. Add Rpt tetras to the program counter, and Arg tetras to the stack pointer. If returning from an application trap the status register is not popped from the stack.

TRAP – Trap

Description:

Execute trap. The data field is loaded into the specified target register, Rt. The trap number to execute comes from the contents of register Ra or an immediate value encoded in the instruction. The trap number must be between 1 and 511. Trap numbers below 64 are reserved for the system. Trap numbers 64 and above may be used by applications.

Traps below 64 will use the vector base register to lookup the location of the service routine. Traps above 64 will use the application control register to lookup the location of the service routine.

Instruction Format:

TRAP Rt, Ra, #Data

39 33	32	31 24	23	22	21 16	15	14	13 8	7 5	4 0
$Imm_{14..8}$	0	$Imm_{7..0}$	Va	Sa	Ra_6	0	0	Rt_6	0_3	0_5

TRAP Rt, #Vec, #Data

39 33	32	31 24	23 16	15	14	13 8	7 6	5	4 0
$Imm_{14..8}$	1	$Imm_{7..0}$	Vec_8	0	0	Rt_6	$\sim_2$	V_9	0_5

Clock Cycles: 1

Operation:

The program counter and the status register are pushed on an internal stack. Next the vector is fetched from the exception vector table and jumped to.

Memory Operations

AMADD - Addition

Description:

Atomically add source operand register Rb to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is contained in register Ra.

Supported Operand Sizes: .t, .o, .n

Instruction Formats: AMO

AMADD Rt, Rb, [Ra]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMADD Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
20_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMAND – Bitwise And

Description:

Bitwise ‘And’ source operand register Rb to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is contained in register Ra.

Supported Operand Sizes: .t, .o, .n

Instruction Formats: AMO

AMAND Rt, Rb, [Ra]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
8_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMAND Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
24_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMASL – Arithmetic Shift Left

Description:

Atomically shift left source operand from memory by Rb and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is contained in register Ra.

Supported Operand Sizes: .t, .o, .n

Instruction Formats: AMO

AMASL Rt, Rb, [Ra]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
6_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMASL Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
22_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMEOR – Bitwise Exclusive Or

Description:

Bitwise exclusive ‘Or’ source operand register Rb to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is contained in register Ra.

Supported Operand Sizes: .t, .o, .n

Instruction Formats: AMO

AMEOR Rt, Rb, [Ra]

39 35	34	33	32 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
10_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMEOR Rt, imm, [Ra]

39 35	34	33	32 31	30 23	22	21 16	15	14 9	8	7 5	4 0
26_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMLSR – Logical Shift Right

Description:

Atomically shift right source operand from memory by Rb and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is contained in register Ra.

Supported Operand Sizes: .t, .o, .n

Instruction Formats: AMO

AMLSR Rt, Rb, [Ra]

39 35	34	33	32 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
7_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMLSR Rt, imm, [Ra]

39 35	34	33	32 31	30 23	22	21 16	15	14 9	8	7 5	4 0
23_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMMIN - Minimum

Description:

If Rb is less than the value from memory, store Rb to memory. The original value of the memory cell is stored in register Rt. The memory address is contained in register Ra. Values are treated as signed two's complement integers. This operation is performed in an atomic fashion.

Supported Operand Sizes: .t, .o, .n

Instruction Formats: AMO

AMMIN Rt, Rb, [Ra]

39 35	34	33	32 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
2_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMMINU - Minimum

Description:

If Rb is less than the value from memory, store Rb to memory. The original value of the memory cell is stored in register Rt. The memory address is contained in register Ra. Values are treated as unsigned integers. This operation is performed in an atomic fashion.

Supported Operand Sizes: .t, .o, .n

Instruction Formats: AMO

AMMINU Rt, Rb, [Ra]

39 35	34	33	32 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
12_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMOR – Bitwise Or

Description:

Bitwise ‘Or’ source operand register Rb to value from memory and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is contained in register Ra.

Supported Operand Sizes: .t, .o, .n

Instruction Formats: AMO

AMOR Rt, Rb, [Ra]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
9_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMOR Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
25_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

CMPXCHG – Compare and Exchange

Description:

If the contents of the addressed memory cell is equal to the contents of Rb then a value is stored to memory from the source register Rc. The original contents of the memory cell are loaded into register Rt. The memory address is contained in register Ra. The memory address must be properly aligned. If the operation was successful then Rt and Rb will be the same value. The compare and swap operation is an atomic operation; no other access is allowed between the load and potential store operation.

Supported Operand Sizes: .t, .o, .n

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa
4	.c	24-bit
5	.p	40-bit
6	.n	96-bit
7		reserved

Instruction Formats: CMPXCHG

CMPXCHG Rt, Rb, Rc, [Ra]

39	38	37 32	31	30	29	28 24	22	21 16	15	14 9	8	7 5	4 0
~	Vc	Rc_6	Sc	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	25_5

Clock Cycles:

Notes:

FLOAD Rn,<ea>

Description:

Load register Rt from floating-point source. The source value is converted to the machine width; 96-bit triple precision.

Supported Operand Sizes: .b, .w, .t, .o, .p, .n

Sz_3	Ext.	Operand
0		reserved
1	.h	16-bit half
2	.s	32-bit single
3	.d	64-bit double
4		Reserved
5		Reserved
6	.t	96-bit triple
7		reserved

Instruction Formats: NDXL

FLOAD Rt, d(Rb,Rc*Sc) – indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
1	Vc	Rc_6	Sc	Vb	Sb	Rb_6	St	Rt_6	$D_{6..0}$	V	Sz_3	16_5

Clock Cycles:

Notes:

FSTORE Ra,<ea>

Description:

Store register Ra to destination. The register is converted from triple precision to the storage precision.

Supported Operand Sizes: .h, .s, .d, .t

Sz_3	Ext.	Operand
0		Reserved
1	.h	16-bit half
2	.s	32-bit single
3	.d	64-bit double
4		Reserved
5		reserved
6	.t	96-bit triple
7		reserved

Instruction Formats: NDXS

STORE Ra, d(Rb, Rc*Sc) – Indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
1	Vc	Rc_6	Sc	Vb	Sb	Rb_6	Sa	Ra_6	$D_{6..0}$	V	Sz_3	18_5

Clock Cycles:

Notes:

LOAD Rn,<ea>

Description:

Load register Rt from source. The source value is sign extended to the machine width. Loading register r54, the stack canary placeholder, will cause a check trap if the value loaded is not equal to the current value of the stack canary register.

Supported Operand Sizes: .b, .w, .t, .o, .p, .n

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa
4	.c	24-bit
5	.p	40-bit
6	.n	96-bit
7		group

Instruction Formats: NDXL

LOAD Rt, d(Rb,Rc*Sc) – indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
0	Vc	Rc_6	Sc	Vb	Sb	Rb_6	St	Rt_6	$D_{6..0}$	V	Sz_3	16_5

Clock Cycles:

Notes:

LOADG Gn,<ea>

Description:

Load group of five registers from source.

Gn	Registers
0	R0 to R4
1	R5 to R9
2	R10 to R14
3	R15 to R19
4	R20 to R24
5	R25 to R29
6	R30 to R34
7	R35 to R39

Gn	Registers
8	R40 to R44
9	R45 to R49
10	R50 to R54
11	R55 to R59
12	R60 to R64
13	ASP, SSP, HSP, MSP

Supported Operand Sizes: .b, .w, .l

Instruction Formats: NDXL

LOADG Gt, d(Rb,Rc*Sc) – indexed

39	38 33	32	31	30	29 24	23	22	2120	19 16	15 8	7 5	4 0
Vc	Rc_6	Sc	Vb	Sb	Rb_6	Vt	~	$\sim_2$	Gt_4	$D_{7..0}$	7_3	16_5

Clock Cycles:

Notes:

LOADZ Rn,<ea>

Description:

Load register Rt from source. The source value is zero extended to the machine width. Loading register r54, the stack canary placeholder, will cause a check trap if the value loaded is not equal to the current value of the stack canary register.

Supported Operand Sizes: .b, .w, .t, .o, .p, .n

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa
4	.c	24-bit
5	.p	40-bit
6	.n	96-bit
7		reserved

Instruction Formats: NDXL

LOAD Rt, d(Rb,Rc*Sc) – indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
~	Vc	Rc_6	Sc	Vb	Sb	Rb_6	St	Rt_6	$D_{6..0}$	V	Sz_3	17_5

Clock Cycles:

Notes:

STORE Ra,<ea>

Description:

Store register Ra to destination.

Supported Operand Sizes: .b, .w, .t, .o, .p, .n

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa
4	.c	24-bit
5	.p	40-bit
6	.n	96-bit
7		group

Instruction Formats: NDXS

STORE Ra, d(Rb, Rc*Sc) – Indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
0	Vc	Rc_6	Sc	Vb	Sb	Rb_6	Sa	Ra_6	$D_{6..0}$	V	Sz_3	18_5

Clock Cycles:

Notes:

STOREG Gt,<ea>

Description:

Store register group to destination. The destination is a 512 bit / 64 byte aligned region of memory.

Gn	Registers
0	R0 to R4
1	R5 to R9
2	R10 to R14
3	R15 to R19
4	R20 to R24
5	R25 to R29
6	R30 to R34
7	R35 to R39

Gn	Registers
8	R40 to R44
9	R45 to R49
10	R50 to R54
11	R55 to R59
12	R60 to R64
13	ASP, SSP, HSP, MSP

Supported Operand Sizes: .b, .w, .l

Instruction Formats: NDXS

STOREG Ga, d(Rb, Rc*Sc) – Indexed

39	38 33	32	31	30	29 24	23	22	2120	19 16	15 8	7 5	4 0
Vc	Rc_6	Sc	Vb	Sb	Rb_6	Va	~	$\sim_2$	Ga_4	$D_{7..0}$	7_3	18_5

Clock Cycles:

Notes:

Compare and Exchange

```
ATOM a0,"AAAAAA"
LOAD a0,[a3]
CMP t0,a0,a1
PEQ t0,"TTF"
STORE a2,[a3]
LDI a0,1
LDI a0,0
```

Load add and store:

```
ATOM "AAA"
LOAD a0,[a2]
ADD t0,a0,a1
STORE t0,[a2]
```

Load or and store

```
ATOM "AAA"
```

```
LOAD a0,[a2]
OR t0,a0,a1
STORE t0,[a2]
```

Load and complement and store

```
ATOM “AAA”
LOAD a0,[a2]
AND t0,a0,~a1
STORE t0,[a2]
```

Vector Specific Instructions

V2BITS

Description

Convert Boolean vector to bits. A bit specified by Rb or an immediate of each vector element is copied to the bit corresponding to the vector element in the target register. The target register is a scalar register. Usually, Rb would be zero so that the least significant bit of the vector is copied.

A typical use is in moving the result of a vector compare operation into a mask register.

Instruction Format: R2

V2BITS Rt, Ra, Rb – Register direct

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
40_6	0	0	0	0	Rb_6	Va	Sa	Ra_6	0	St	Rt_6	Sz_3	2_5

Clock Cycles: 1

V2BITS Rt, Ra, #bit – Register direct

39 34	33	32	31	30 24	23	22	21 16	15	14	13 8	7 5	4 0
40_6	1	0	0	Imm_7	Va	Sa	Ra_6	0	St	Rt_6	Sz_3	2_5

Clock Cycles: 1

Operation

For x = 0 to VL-1

Rt.bit[x] = Ra[x].bit[Rb]

Exceptions: none

Example:

```
cmp v1,v2,v3        ; compare vectors v2 and v3
v2bits m1,v1,#8     ; move NE status to bits in m1
vmask “11100000”
add v4,v5,v6        ; perform some masked vector operations
muls v7,v8,v9
add v7,v7,v4
```

Cryptographic Accelerator Instructions

AES64DS – Final Round Decryption

Description:

Perform the final round of decryption for the AES standard. Registers Rb, Ra represent the entire AES state.

Integer Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$50h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Operation:

Rt = Ra & Rb

Exceptions: none

AES64DSM – Middle Round Decryption

Description:

Perform a middle round of decryption for the AES standard. Registers Rb, Ra represent the entire AES state.

Integer Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$51h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Operation:

Rt = Ra & Rb

Exceptions: none

AES64ES – Final Round Encryption

Description:

Perform the final round of encryption for the AES standard. Registers Rb, Ra represent the entire AES state.

Integer Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$52h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Operation:

Rt = Ra & Rb

Exceptions: none

AES64ESM – Middle Round Encryption

Description:

Perform a middle round of encryption for the AES standard. Registers Rb, Ra represent the entire AES state.

Integer Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$53h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Operation:

Rt = Ra & Rb

Exceptions: none

SHA256SIG0

Description:

Implements the Sigma0 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Ra are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R2

SHA256SIG0 Rt, Ra – Register direct

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
56_6	~	0	0	0	0_6	Va	Sa	Ra_6	Vt	St	Rt_6	6_3	2_5

Clock Cycles: 1

Operation:

Rt = sign extend(ror32(Ra,7) ^ ror32(Ra,18) ^ (Ra_{32} >> 3))

Execution Units: ALU #0

Exceptions: none

SHA256SIG1

Description:

Implements the Sigma1 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Ra are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R2

SHA256SIG1 Rt, Ra – Register direct

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
57_6	~	0	0	0	0_6	Va	Sa	Ra_6	Vt	St	Rt_6	6_3	2_5

Clock Cycles: 1

Operation:

Rt = sign extend(ror32(Ra,17) ^ ror32(Ra,19) ^ (Ra_{32} >> 10))

Execution Units: ALU #0

Exceptions: none

SHA256SUM0

Description:

Implements the Sum0 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Ra are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R2

SHA256SUM0 Rt, Ra – Register direct

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
58_6	~	0	0	0	0_6	Va	Sa	Ra_6	Vt	St	Rt_6	6_3	2_5

Clock Cycles: 1

Operation:

Rt = sign extend(ror32(Ra,2) ^ ror32(Ra,13) ^ ror32(Ra, 22))

Execution Units: ALU #0

Exceptions: none

SHA256SUM1

Description:

Implements the Sum1 transformation function used in the SHA2-256 and SHA2-224 hash function. Only the low order 32 bits of Ra are operated on. The 32-bit result is sign extended to the machine width.

Instruction Format: R2

SHA256SUM1 Rt, Ra – Register direct

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
59_6	~	0	0	0	0_6	Va	Sa	Ra_6	Vt	St	Rt_6	6_3	2_5

Operation:

Rt = sign extend(ror32(Ra,6) ^ ror32(Ra,11) ^ ror32(Ra, 25))

Execution Units: ALU #0

Exceptions: none

SHA512SIG0

Description:

Implements the Sigma0 transformation function used in the SHA2-512 hash function.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$34h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

Clock Cycles: 1

Operation:

Rt = ror64(Ra,1) ^ ror64(Ra, 8) ^ (Ra >> 7)

Execution Units: ALU #0

Exceptions: none

SHA512SIG1

Description:

Implements the Sigma1 transformation function used in the SHA2-512 hash function.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$35h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

Clock Cycles: 1

Operation:

Rt = ror64(Ra,19) ^ ror64(Ra, 61) ^ (Ra >> 6)

Execution Units: ALU #0

Exceptions: none

SHA512SUM0

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$36h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

SHA512SUM1

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$37h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

SM3P0

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$38h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

SM3P1

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$39h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

SM4ED

Description:

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$56h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

SM4KS

Description:

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$57h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Modifiers

ATOM

Description:

Treat the following sequence of instructions as an “atom”. Rt specifies the register results are to be written to.

Disable interrupts for the following instructions.

	Mask Bit	
MASK Modifier Scope	0,1	Instruction zero
	2,3	Instruction one
	4,5	Instruction two
	6,7	Instruction three
	8,9	Instruction four
	10,11	Instruction five
	12,13	Instruction six
	14,15	Instruction seven

Mask Bit	Meaning
00	No action
01	Disable interrupts
10	Disable interrupts and lock bus
11	Reserved

Instruction Format:

39 34	33	32	31 24	23 16	15	14	13 8	7 5	4 0
35_6	~	1	$Imm_{15..8}$	$Imm_{7..0}$	Vt	St	Rt_6	Sz_3	2_5

Assembler Syntax:

Example:

```
ATOM “LLLLAA”
LOAD a0,[a3]
CMP t0,a0,a1
PEQ t0,”TTF”
STORE a2,[a3]
LDI a0,1
LDI a0,0
```

```
ATOM “LLLL”
LOAD a1,[a3]
ADD t0,a0,a1
```

```
MOV a0,a1
STORE t0,[a3]
```

CARRY

Description:

Apply the carry modifier to following instructions according to a bit mask. This modifier may be used to perform extended precision addition. It may also be used to retrieve the high order multiplier bits or the divide remainder. Note that carry input is not available for the first instruction under the modifier's shadow. Generating carry output for the eight instruction is discarded. Note that postfixes do not count as instructions.

	Mask Bit	
Carry Modifier Scope	0,1	Instruction zero
	2,3	Instruction one
	4,5	Instruction two
	6,7	Instruction three
	8,9	Instruction four
	10,11	Instruction five
	12,13	Instruction six
	14,15	Instruction seven

Mask Bit	Letter	Meaning
00	N	No carry in or out
01	I	Use carry in
10	O	Generate carry out
11	C	Use carry in and generate carry out

Instruction Format:

39 34	33	32	31 16	15	14	13 8	7 5	4 0
33_6	~	0	$Imm_{15..0}$	~	0	Rn_6	$\sim_3$	2_5

Assembler Syntax:

Specifying carry input / output capability for following instructions consists of a map using one of four characters: 'I' for input only, 'O' for output only, 'C' for both input and output and 'N' for neither input or output. A character is present in a string for each following instruction in sequence.

Example:

```
CARRY "OCCCCINN"   ; first generate carry out, second to fifth use carry in and out, sixth use carry in, seven and eight ignore carry.
ADD r6,r3,r7        ; 'O' gen carry
ADD r6,r6,#1234     ; 'C' carry in and carry out
ADD r6,r2,r1        ; 'C' carry in and carry out
ADD r6,r6,#456      ; 'C' carry in and carry out
ADD r7,r6,#456      ; 'C' carry in and carry out
ADD r8,r7,#987      ; 'I' carry in
MUL r8,r9,r10       ; 'N' no carry in or out
```

VMASK

Description:

Apply the vector masking to following instructions according to a bit mask. Note that postfixes do not count as instructions.

MASK Modifier Scope	Mask Bit	
	0 to 2	Instruction zero
	3 to 5	Instruction one
	6 to 8	Instruction two
	9 to 11	Instruction three
	12 to 14	Instruction four
	15 to 17	Instruction five
	18 to 20	Instruction six
	21 to 23	Instruction seven

Instruction Format:

39 34	33	32	31 8	7 5	4 0
34_6	~	1	$Imm_{23..0}$	$\sim_3$	2_5

Assembler Syntax:

Specifying the mask register for following instructions consists of a map using single digit numeric characters between '0' and '7'. A character is present in a string for each following instruction in sequence.

Example:

```
VMASK ”12345000”
ADD v6,v3,v7           ; vector mask reg #1
ADD v6,v6,#1234        ; vector mask reg #2
ADD v6,v2,v1           ; vector mask reg #3
ADD v6,v6,#456         ; vector mask reg #4
ADD v7,v6,#456         ; vector mask reg #5
ADD v8,v7,#987         ; vector mask reg #0
MUL v8,v9,v10          ; vector mask reg #0
```

PRED

Description:

Apply the predicate to following instructions according to a bit mask. The predicate may be applied to a maximum of eight instructions. Note that postfixes do not count as instructions.

	Mask Bit	
Pred Modifier Scope	0,1	Instruction zero
	2,3	Instruction one
	4,5	Instruction two
	6,7	Instruction three
	8,9	Instruction four
	10,11	Instruction five
	12,13	Instruction six
	14,15	Instruction seven

Mask Bit	Meaning
00	Always execute (ignore predicate)
01	Execute only if predicate is true
10	Execute only if predicate is false
11	Always execute (ignore predicate)

Instruction Format:

39 34	33	32	31 16	15 10	9 5	4 0
32_6	~	1	$Imm_{15..0}$	Rn_6	$Cond_5$	2_5

Assembler Syntax:

The predicate condition is part of the mnemonic. ‘PEQ’ predicates logic if the equals flag in the register containing flags is set. Other conditions work in a similar fashion. After the instruction mnemonic the register containing the predicate flags is specified. Next a character string containing ‘T’ for True, ‘F’ for false, or ‘I’ for ignore for the next eight instructions is present.

Example:

```
PEQ r2,"TTTFFFII"   ; next three execute if true, three after execute if false, two after always execute
MUL r3,r4,r5        ; executes if True
ADD r6,r3,r7        ; executes if True
ADD r6,r6,#1234     ; executes if True
DIV r3,r4,r5        ; executes if FALSE
ADD r6,r2,r1        ; executes if FALSE
ADD r6,r6,#456      ; executes if FALSE
MUL r8,r9,r10       ; always executes
```

ROUND

Description:

Set the rounding mode for following instructions according to a bit mask. Note that postfixes do not count as instructions.

	Mask Bit	
ROUND Modifier Scope	0 to 2	Instruction zero
	3 to 5	Instruction one
	6 to 8	Instruction two
	9 to 11	Instruction three
	12 to 14	Instruction four
	15 to 17	Instruction five
	18 to 20	Instruction six
	21 to 23	Instruction seven

Instruction Format:

39 34	33	32	31 8	7 5	4 0
36_6	~	1	$Imm_{23..0}$	$\sim_3$	2_5

Assembler Syntax:

Example: