THOR CORE2021 GUIDE

[Document subtitle]

ABSTRACT

Details for the Thor2021 processing core including programming model, memory management and instruction set architecture.

Robert Finch

[Course title]

Table of Contents

Overview

Thor is a powerful 64-bit superscalar processor that represents a generational refinement of processor architecture. The processor contains 64, 64 bit general purpose integer registers. Thor uses variable length instructions varying between two and eight bytes in length and handles 8, 16, 32, and 64 bit data within a 64 bit address space.

History

Thor2021 is a work in progress beginning in October 2021. Thor2021 originated from Thor which originated from RiSC-16 by Dr. Bruce Jacob. RiSC-16 evolved from the Little Computer (LC-896) developed by Peter Chen at the University of Michigan. See the comment in Thor2021.v. The author has tried to be innovative with this design borrowing ideas from many other processing cores.

Design Objectives

This processor is somewhat pedantic in nature and targeted towards high performance operation as a general-purpose processor. Following are some of the criteria that were used on which to base the design.

- Designed for Superscalar operation - the ability to execute more than one instruction at a time. To achieve high performance it is generally accepted that a processor must be able to execute more than a single instruction in any given clock cycle.
- Support for vector operations.
- Simplicity - architectural simplicity leads to a design that is easy to implement resulting in reliability and assured correctness along with easy implementation of supporting tools such as compilers. Simplicity also makes it easier to obtain high performance and results in lower overall cost.
- Extensibility - the design must be extensible so that features not present in the first release can easily be added at a later date.
- Low Cost

This design meets the above objectives in the following ways. The instruction set has been designed to minimize the interactions between instructions, allowing instructions to be executed as independent units for superscalar operation. There are a sufficient number of registers to allow the compiler to schedule parallel processing of code. A reasonably large general purpose register set is available making the design reasonably compatible with many existing compilers and assemblers. Where needed, additional specialized instructions have been added to the processor to support a sophisticated operating system and interrupt management.

Motivation

The author wanted an FPGA based processing core for experimental purposes.

Differences from the Original

The string instructions in the original Thor ISA are no longer present.

Stack operations have been removed.

Mnemonics changed for load and store instructions.

Byte, wyde and UTF21 search instructions have been added.

Support for decimal floating-point has been added.

Support for cryptographic accelerator functions has been added.

Case Comparison Hi-lites

Some of the more striking points of a handful of architectures are compared to what is available in Thor2021.

Case Comparison 6502

6502 vs Thor2021

Overview

This is a bit of an apples to oranges comparison as the two designs are for different environments. The 6502 was designed for a much smaller operating environment and is extremely frugal with transistor usage. The Thor2021 was designed as 64-bit processor used for experimentation in a much larger environment.

Instruction Format

The 6502 as a byte-oriented design has a compact variable instruction length encoding. Many instructions are encoded using an average of about two bytes.

While variable sized instructions offer great advantage for code density, they add complexity to the processing core. Thor2021 also uses a variable size instruction encoding. As such for a given single instruction it requires roughly twice the memory of a 6502. However, the instructions in the Thor2021 operate on 64-bit values, to perform the same operations in the 6502 would require many more bytes. Several instructions in the Thor2021 are more powerful than what can be found in the 6502.

Registers

The Thor2021 has many more registers than the 6502. It is a general-purpose register-oriented design while the 6502 is accumulator oriented. A register file of about 32 registers has been found to be a good match to many computing environments. This is somewhat of a historical determination. The Thor2021 has available many more transistors than were available to the 6502 design. The Thor2021 has many special purpose registers. The 6502 does not have any.

Instructions

The 6502 uses relative branches to allow a code dense instruction encoding. Thor2021 also uses relative branches to help reduce the instruction size. It has a larger branch displacement than the 6502 as that is what could be encoded easily.

The 6502 offers only basic instructions (ADD, SUB, CMP, AND, ORA, EOR, LDA, STA) as examples. There are no complex instructions in the 6502 ISA. All instructions execute within a handful of clock cycles. the Thor2021 has a ton of instructions compared to a 6502. It supports floating point and posit arithmetic.

The 6502 is an accumulator-based architecture that allows one memory-based operand for most instructions. Thor2021 is register based and the only instructions accessing memory are load and store type instructions.

Case Comparison ARM

Overview

The ARM architecture has become extremely popular.

Instruction Format

The ARM machine was originally a 32-bit fixed instruction format machine. It has had added onto it 16-instruction formats.

Registers

The program counter is referenced using one the registers codes available for general purpose registers.

> *The author is not fond of architectures that use a general-purpose register as the program counter. He believes a separate register is a better approach. Having the pc as part of the general register file is archaic.*

Case Comparison RISCV

RISCV vs Thor2021

Instruction Format

While variable sized instructions offer great advantage for code density, they add complexity to the processing core.

In RISCV support for 16-bit compressed instructions consumes two opcode bits, and opcode bits are valuable. The use of these two bits and the reduction of the opcode space for other instructions is an excellent trade-off. Compressed instructions can improve code density by about 25% or more and consequently make better use of the cache. There is only the occasional instruction that can not be encoded using two fewer encoding bits, so only a very small percentage would be gained back in code density by having two more bits available. Thor2021 uses a variable length instruction encoding which allow it to achieve code density similar to RISCV.

The JAL instruction in RISCV allows any register to be used to store the return address. In practice only one or two registers which are fixed by the ABI are used. This means that there are about four bits of opcode space wasted for unnecessary register specification. Making use of

these extra four bits is extremely valuable. The Thor2021 design only requires two bits to specify the return address register. The presence of four extra bits to specify the target address makes absolute addressing appealing for this design.

To build constants the LUI instruction is used. In RISCV the LUI instruction allows any register to be used as the target and has a 20-bit constant field because of encoding constraints. In practice it is possible to get by using only one or two registers to build constants with. Thor2021 has more direct support for constants larger than 32 bits. It makes use of ADDIS (add immediate shifted), ORIS, and ANDIS instructions to build 64-bit constants. These instructions support building 64-bit constants directly. RISCV does not really provide much for building constants over 32 bits.

Instructions

RISCV does not include indexed addressing modes in the standard implementation. Indexed addressing is accomplished when required using additional instructions and registers to calculate the effective address. Thor2021 directly supports indexed addressing with an optionally scaled index register. When indexed addressing is required Thor2021 is more code dense than RISCV. However indexed addressing is not used that often.

RISCV accesses memory and I/O exclusively using load and store instructions. Thor2021 has several additional instructions which access memory and I/O.

Register File

RISCV does almost everything using general-purpose registers. This paradigm increases the pressure on the register file. In the Thor2021 design there are more register files involved. Effectively, there are a few more additional registers which reduce the pressure on the general-purpose register file. There is a trend to place some global variables in the register file for performance reasons. These variables include operating vars for garbage collection, pointers to global and thread data and pointers for exception handling.

One reason to use more register files is that in a superscalar design it may allow more instructions to be committed at the same time. There is usually a limit on the number of write ports to the general register file. This limit affects how many instructions can be committed at once. By providing separate register files for some operations it effectively increases the number of write ports available making it possible to commit more instructions per cycle.

Return Address Registers

There is not a requirement for more than a couple of return address registers. The instruction set may be refined to allow only a single bit to specify the return address register.

Compare Results Registers

RISCV stores comparison results if needed in general-purpose registers. It has just a single instruction (SLT) dedicated to generating compare results. RISCV makes use of branches that compare-and-branch encoded in a single instruction. This is effective at removing the need for most compare operations. The intermediate result of the compare is hidden in the architecture; there is no need for visible compare results registers. There is still a need for the computed result of a compare operation. Sometimes software records the comparison result for later usage. For example, there may be a line of code: x = y > 10. Which will set x true if y is greater than 10.

Compares are tightly coupled to branch operations. Some architectures like RISCV compare and branch in a single instruction. Other architectures use a flags register or several flags registers. Yet other architectures simply use the general-purpose registers.

One reason to use a separate group of compare results registers is that in a superscalar design it may allow more instructions to be committed at the same time. There is usually a limit on the number of write ports to the general register file. This limit affects how many instructions can be committed at once. By providing separate register files for some operations it effectively increases the number of write ports available making it possible to commit more instructions per cycle.

Operating modes.

This design uses four operating modes. It has the RISCV operating modes. The author has seen a comment to the effect that debug on a RISCV processor really acts like an additional mode.

Memory Management

RISCV offers several memory management options including several different paging arrangements and a couple of optional base and bound registers.

Case Comparison MMIX

Instruction Format

MMIX comes across as more of a pedantic processor design. MMIX instructions are structure simply for the most part using a 32-bit format divided into four-byte regions. The author assumes this is primarily to enhance the readability of instructions. The constant field is often limited to eight bits. Thor2021 has fewer registers and that allows more constant bits to be encoded in the same size instruction.

Register File

MMIX has a 256-entry register file. It is not clear that this number of registers has any benefit over a 32-register design, but it makes the instruction format clear and easy to understand which may be a goal for a processor used for academic purposes.

Instructions

There are a lot of conditional move instructions in the MMIX ISA. Thor2021 currently supports only a single conditional move instruction.

Case Comparison PowerPC

Instruction Format

The PowerPC uses a fixed 32-bit instruction format.

Instructions

The PowerPC supports indexed addressing like the Thor2021 although index scaling is not present. The author has found indexed addressing makes up about 3% of instructions and scaled indexes a much smaller percentage.

Registers

The PowerPC has a dedicated link register and eight condition code registers. Thor2021 has with a pair of link registers dedicated in the GPR file. The PowerPC also has a loop count register used for counted loops. Thor2021 also has a loop count register.

Case Comparison x86

Registers

The x86 series has a register file that is accessible in subparts. Parts of a single register may be referred to instructions. For example, EAX is a 32-bit register that is also accessible as AL for byte operations. This has no-doubt complicated the x86 design. This contrasts with Thor2021 and many RISC designs where the registers are always manipulated as whole units.

Case Comparison SPARC

Registers

The SPARC machine uses register windowing, where a subset of registers is available from a much larger set that is “windowed”. In the SPARC the subset register window scrolls up and down automatically during subroutine calls and returns. The idea was to improve performance by not having to stack and unstack registers to memory during subroutine operations. However, with a good modern optimizing compiler the performance level of the SPARC is not much different than that of other architectures.

Nomenclature

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

Number of Bits		Instructions
8	byte	LDB, STB
16	wyde	LDW, STW
32	tetra	LDT, STT
64	octa	LDO, STO
128	hexi	LDH, STH

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.

Development Aspects

Device Target

The core has been developed with FPGA usage in mind. In particular it is expected that the register file is built out of block memories.

Implementation Language

The core is implemented in the System Verilog language primarily for its ability to process array objects. Much of the core is plain vanilla Verilog code.

Programming Model

General Registers

There are 64 general purpose registers. General purpose registers are 64 bits wide. The general register file is unified and may hold integer or floating-point values.

Register #0 is always zero.

r0	always zero
r1	return value / arg0
r2	return value / arg1
r3	temporary register caller save
r4	temporary register
r5	temporary register
r6	temporary register
r7	temporary register
r8	temporary register
r9	temporary register
r10	temporary register
r11	register var callee save
r12	register var
r13	register var
r14	register var
r15	register var
r16	register var
r17	register var
r18	register var
r19	
r20	
r21	
r22	
r24	Type number
r25	Class Pointer
r26	Base Pointer
r27	User Stack Pointer[1]
r28	Interrupt Stack Pointer
r29	Exception Stack Pointer
r30	Debug Stack Pointer
r31	Kernel task register
r32/F0	Floating point
…	
r63/F31	

LC	Loop Counter

C0	available
C1	return address
C2	milli-code return address
C3	available
C4	available
C5	available
C6	exceptioned IP
C7	Instruction pointer, read only

ZS	
DS	
ES	
FS	
GS	
HS	
SS	
CS	

DBAD0	Debug Address #0
DBAD1	Debug address #1
DBAD2	Debug address #2
DBAD3	Debug Address #3
DBCTRL	Debug Control
DBSTAT	Debug Status

1 this register is implied in the push and rts instructions, and updated by hardware

r27 is special in that it refers to one of r27, r28, r29, or r30 depending on the operating mode of the core. This allows the same code to be reused in different operating modes. For instance loading r27 while in debug mode will actually load r30 and all references to r27 will be rerouted to r30 in debug mode.

Register Tags

For the bypassing network, commonly used registers have a register tag associated with them. The register tag for the general registers varies from 1 to 63 corresponding to registers 1 to 63. Vector registers use tags 64 to 127. Other registers use additional tags as noted in the text.

Stack and Frame Pointers

Although the stack and frame pointer registers may be used with any instruction the core has special hardware to detect stack bounds violations by either the stack pointer or frame pointer. The stack and frame pointer registers should be kept aligned on octa-byte boundaries. That is, they should be a multiple of eight, which has the least significant three bits as zero. There is currently no hardware in the core to enforce alignment.

> *The author considered having the stack pointer as an independent register but that would require replicating a number of instructions (add, sub, and, or, etc.) just for the stack pointer. The author feels it is better to keep the stack pointer general-purpose in nature so that it may leverage the usage of the existing instruction set. This design is primarily a load / store architecture.*

Loop Count

The loop count register is used with branch instructions to form counted loops. It may be automatically decremented and tested when the branch instruction is executing.

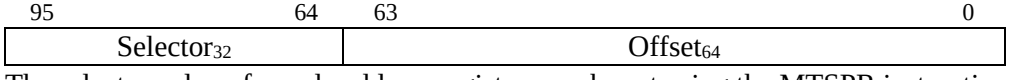
Code Address Registers

Thor2021 has eight code address registers C0 to C7. These are also referred to as branch registers in other architectures. C7 refers to the current instruction pointer.

Code Address Register Format

A code address register is composed of a 64-bit offset field and a 32-bit selector field. It is necessary to store both the selector and offset in a linkage register so that a far return may be performed.

A branch instruction will set both the selector and offset values for the instruction pointer. The new selector value for the IP will come from one of the other code address registers as specified in the instruction.



The selector value of a code address register may be set using the MTSPR instruction and selecting the code address selector as the target.

Reg #		Usage
0	available for use	zero by convention
1	Subroutine return address	dedicated for subroutine linkage
2	Milli-code return address	dedicated for subroutine linkage
3	available for use	
4	available for use	
5	available for use	
6	Exception Instruction Pointer	dedicated for exception processing
7	Instruction Pointer	dedicated to instruction addressing

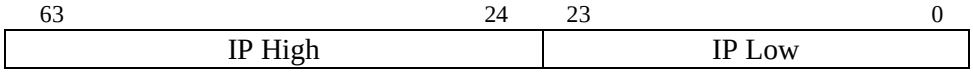
The presence of multiple code address registers allows multi-level return addresses to be used for performance. Leaf routines may use C1 as the return address. Next to leaf routines may use C2, etc. So that memory operations are avoided when implementing subroutine call and return.

The instruction pointer register is read-only. The instruction pointer cannot be modified by moving a value to this register.

Instruction Pointer

The instruction pointer, IP, points to the currently executing instruction. The lower 24-bits of the instruction pointer increment as instructions are processed. Branch instructions normally manipulate only the low order 24 bits of the instruction pointer. The entire pointer may be set using a branch-to-register instruction.

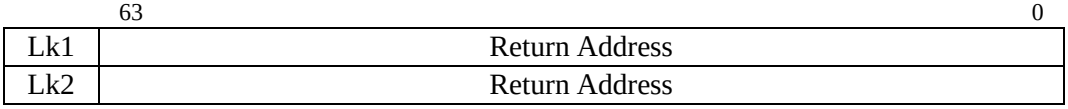
> *To conserve hardware and improve performance of the counter only the low order 24-bits increment. It is extremely rare to have 16MB or more of code without resets of the entire instruction pointer due to subroutine calls.*



Link Registers

Related to the instruction pointer are subroutine linkage registers. The architecture has two link registers for storing subroutine return addresses.

> *While many architectures have only a single link register, it is sometimes useful to have a second link register, for instance to implement milli-code routines. Some architectures allow any general- purpose register to be used for subroutine linkage. Often the ABI specifies that a specific register is used for this purpose. The author feels that supporting any GPR as a link register wastes instruction encoding bits that are better used for other purposes.*



Selector Registers

Selector registers are a piece of the segmented memory management. They are a short form for segment descriptors which they represent. There are nine selector registers in the architecture. Several are dedicated to specific uses.

Reg	Tag	Usage
ZS	144	
DS	145	data segment
ES	146	
FS	147	
GS	148	
HS	149	
SS	150	stack segment
CS	151	code segment
LDT	-	refers to address and size of local descriptor table

General Purpose Vector (v0 to v63) / Registers

v0 always has the value zero.

Register	Description / Suggested Usage	Saver
v0	always reads as zero (hardware)	
v1-v63		

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).

Register	Tag	Usage
m0	160	contains all ones by convention
m1	161	
m2	162	
m3	163	
m4	164	
m5	165	
m6	166	
m7	167	

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 #160.

15 … 8	7 … 0
0	$Elements_{7..0}$

Summary of Special Purpose Registers

[U/S/H/M]_IE (0x?004)

This register contains interrupt enable bits. The register is present at all operating levels. Only enable bits at the current operating level or lower are visible and may be set or cleared. Other bits will read as zero and ignore writes. Only the lower four bits of this register are implemented. The bits have individual bit set / clear capability using the CSRRS, CSRRC instructions.

63 … 4	3	2	1	0
~	mie	hie	sie	uie

[U/S/H/M]_CAUSE (CSR- 0x?006)

This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code to determine what to do. Only the low order 16 bits are implemented. The high order bits read as zero and are not updateable.

[U/S/H/M]_SCRATCH – CSR 0x?041

This is a scratchpad register. Useful when processing exceptions. There is a separate scratch register for each operating mode.

[U/S/H/M]_TIME (0x?FE0)

The TIME register corresponds to the wall clock real time. This register can be used to compute the current time based on a known reference point. The register value will typically be a fixed number of seconds offset from the real wall clock time. The lower 32 bits of the register are driven by the tm_clk_i clock time base input which is independent of the cpu clock. The tm_clk_i input is a fixed frequency used for timing that cannot be less than 10MHz. The low order 32 bits represent the fraction of one second. The upper 32 bits represent seconds passed. For example, if the tm_clk_i frequency is 100MHz the low order 32 bits should count from 0 to 99,999,999 then cycle back to 0 again. When the low order 32 bits cycle back to 0 again, the upper 32 bits of the register is incremented. The upper 32 bits of the register represent the number of seconds passed since an arbitrary point in the past.

Note that this register has a fixed time basis, unlike the TICK register whose frequency may vary with the cpu clock. The cpu clock input may vary in frequency to allow for performance and power adjustments.

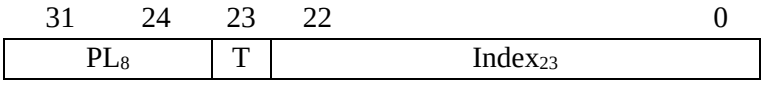
U_FSTAT - CSR 0x0014 Floating Point Status and Control Register

The floating-point status and control register may be read using the CSR instruction. Unlike other CSR's the control register has its own dedicated instructions for update. See the section on floating point instructions for more information.

Bit		Symbol	Description
63:53			reserved
52		inexact	inexact
51		dbz	divide by zero
50		under	underflow
49		over	overflow
48		invop	invalid operation
47		~	reserved
46:44	**RM**	rm	rounding mode
43	**E5**	inexe	- inexact exception enable
42	**E4**	dbzxe	- divide by zero exception enable
41	**E3**	underxe	- underflow exception enable
40	**E2**	overxe	- overflow exception enable
39	**E1**	invopxe	- invalid operation exception enable
38	**NS**	ns	- non standard floating point indicator
Result Status			
32		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
31	**RA**	rawayz	rounded away from zero (fraction incremented)
30	**SC**	C	denormalized, negative zero, or quiet NaN
29	**SL**	neg <	the result is negative (and not zero)
28	**SG**	pos >	the result is positive (and not zero)
27	**SE**	zero =	the result is zero (negative or positive)
26	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
21 to 25			reserved
20	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
19	**X5**	inerx	- inexact result exception occurred (sticky)
18	**X4**	dbzx	- divide by zero exception occurred
17	**X3**	underx	- underflow exception occurred
16	**X2**	overx	- overflow exception occurred
15	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
14	**GX**	gx	- global exception indicator – set if any enabled exception has happened
13	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
8 to 12			reserved
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

S_PTA – CSR 0x1003

This register contains the selector for the PTA descriptor describing the highest-level page directory for memory management. The PTA descriptor contains the paging table depth and the size of the pages mapped. Register tag #152.



PTA Descriptor

The PTA descriptor establishes the location and size of the root page table in memory. The base address must be 4kB aligned.

n+3	ACR_{20}	$\sim_{44}$				
n+2	$Limit_{63..0}$					
n+1	$\sim_{64}$					
n	$Base_{63..12}$		~	TD_3	S_1	$\sim_7$

TD			S	
0	1 level lookup		0	map 4kB pages
1	2 level lookup		1	map 1MB pages
2	3 level lookup			
3	4 level lookup			
4 to 7	reserved			

S_ASID – CSR 0x101F

This register contains the address space identifier (ASID) or memory map index (MMI). The ASID is used in this design to select (index into) a memory map in the paging tables. Only the low order eight bits of the register are implemented.

S_KEYS – CSR 0x1020 to 0x1027

These eight registers contain the collection of keys associated with the process for the memory lot system. Each key is twenty-one bits in size. All eight registers are searched in parallel for keys matching the one associated with the memory page. Keyed memory enhances the security and reliability of the system.

			20 0
1020			key0
1021			key1
…			…
1027			key7

C0,C1,…C7 (CREGS) – M_CSR 0x3100 to 0x310F

This set of registers allows access to the code address register array. Since code address registers are larger than 64-bits they are split across two register locations for each code address register.

Reg #	Name	Alternate Name	Reg Tag
0x3100	C0L		128
0x3101	C0H		129
…			…
0x310C	C6L	EIPL	140
0x310D	C6H	EIPH	141
0x310E	C7L	IPL	142
0x310F	C7H	IPH	143

ZS,DS,ES,FS,GS,HS,SS,CS (SREGS) – M_CSR 0x3120 to 0x3127

These registers reflect the values of the segment selectors currently active in the core. Writing to a selector register triggers a load of the corresponding descriptor cache. The CS selector is read-only.

Reg #	Name	Reg Tag
0x3120	ZS	144
0x3121	DS	145
0x3122	ES	146
0x3123	FS	147
0x3124	GS	148
0x310D	HS	149
0x310E	SS	150
0x310F	CS	151

m0 to m7 – U_CSR 0x0130 to 0x0137

M_CR0 (CSR 0x3000) Control Register Zero

This register contains miscellaneous control bits including a bit to enable protected mode.

Bit		Description
0	Pe	Protected Mode Enable: 1 = enabled, 0 = disabled
8 to 13		
16		
30	DCE	data cache enable: 1=enabled, 0 = disabled
32	BPE	branch predictor enable: 1=enabled, 0=disabled
34	WBM	write buffer merging enable: 1 = enabled, 0 = disabled
35	SPLE	speculative load enable (1 = enable, 0 = disable) (0 default)
36		
63	D	debug mode status. this bit is set during an interrupt routine if the processor was in debug mode when the interrupt occurred.

This register supports bit set / clear CSR instructions.

DCE

Disabling the data cache is useful for some codes with large data sets to prevent cache loading of values that are used infrequently. Disabling the data cache may reduce security risks for some kinds of attacks. The instruction cache may not be disabled. Enabling / disabling the data cache is also available via the CACHE instruction.

BPE

Disabling branch prediction will significantly affect the cores performance but may be useful for debugging. Disabling branch prediction causes all branches to be predicted as not-taken. No entries will be updated in the branch history table if the branch predictor is disabled.

WBM bit

Merging of values stored to memory may be disabled by setting this bit. On reset write buffer merging is disabled because it is likely desirable to setup I/O devices. Many I/O devices require updates to individual bytes by separate store instructions. (Write buffer merging is not currently implemented).

SPLE

Enabling speculative loads give the processor better performance at an increased security risk to meltdown attacks.

M_HARTID (CSR 0x3001)

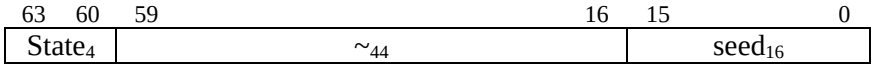
This register contains a number that is externally supplied on the hartid_i input bus to represent the hardware thread id or the core number.

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.

M_SEED (CSR 0x3003)

This register contains a random seed value based on an external entropy collector. The most significant bit of the state is a busy bit.



State$_4$ Bit	
0	dead
1	test
2	valid, the seed value is valid
3	Busy, the collector is busy collecting a new seed value

M_BADADDR (CSR 0x3007)

This register contains the effective address for a load / store operation that caused a memory management exception or a bus error. Note that the address of the instruction causing the exception is available in the EIP register.

M_BAD_INSTR (CSR 0x300B)

This register contains a copy of the exceptioned instruction.

M_DBADx (CSR 0x3018 to 0x301B) Debug Address Register

These registers contain addresses of instruction or data breakpoints. The registers may also be used as trace triggering address registers.

63	0
Address $_{63..0}$	

M_DBCR (CSR 0x301C) Debug Control Register

This register contains bits controlling the circumstances under which a debug interrupt will occur.

bits			
3 to 0	Enables a specific debug address register to do address matching. If the corresponding bit in this register is set and the address (instruction or data) matches the address in the debug address register then a debug interrupt will be taken.		
17, 16	This pair of bits determine what should match the debug address register zero in order for a debug interrupt to occur.		
	17:16		
	00	match the instruction address	
	01	match a data store address	
	10	reserved	
	11	match a data load or store address	
19, 18	This pair of bits determine how many of the address bits need to match in order to be considered a match to the debug address register. These bits are ignored when matching instruction addresses, which are always half-word aligned.		
	19:18		Size
	00	all bits must match	byte
	01	all but the least significant bit should match	char
	10	all but the two LSB's should match	tetra
	11	all but the three LSB's should match	octa
23 to 20	Same as 16 to 19 except for debug address register one.		
27 to 24	Same as 16 to 19 except for debug address register two.		
31 to 28	Same as 16 to 19 except for debug address register three.		
32 to 35	Trace enable on address register		
36	Enable branch compression for trace.		
55 to 62	These bits are a history stack for single stepping mode. An exception will automatically disable single stepping mode and record the single step mode state on stack. Returning from an exception pops the single step mode state from the stack.		
63	This bit enables SSM (single stepping mode)		

M_DBSR (CSR 0x301D) - Debug Status Register

This register contains bits indicating which addresses matched. These bits are set when an address match occurs and must be reset by software.

bit	
0	matched address register zero
1	matched address register one

2	matched address register two
3	matched address register three
63 to 4	not used, reserved

M_TVEC – CSR 0x3030 to 0x3037

These registers contain the address of the exception handling routine for a given operating level. TVEC[3] (0x3036) is used directly by hardware to form an address of the debug routine. The lower eight bits of TVEC[3] are not used. The lower bits of the exception address are determined from the operating level. TVEC[0] to TVEC[2] are used by the REX instruction. The low half of the register contains the offset of the exception processing routine. The high half of the register contains the selector value of the exception processing routine.

A sync instruction should be used after modifying one of these registers to ensure the update is valid before continuing program execution.

Reg #	
0x3030	TVEC[0] low
0x3031	TVEC[0] high
…	
0x3036	TVEC[3] low
0x3037	TVEC[3] high

M_PM_STACK – CSR 0x3040

This register contains an eight-entry operating mode and interrupt mask stack. When an exception or interrupt occurs, this register is shifted to the left by eight bits and the low order bits are set according to the exception mode, when an RTE instruction is executed this register is shifted to the right by eight bits. On RTE the last stack entry is set to $3F masking all interrupts on stack underflow. The low order eight bits represent the current operating mode and interrupt mask.

63 … 8	7 6	5 4	3 1	0
<seven more groups>	~2	OM	IPL	IM

OM = operating mode, 0 to 4
IPL = interrupt priority level
IM = interrupt mask

M_SCRATCH – CSR 0x3041

This is a scratchpad register. Useful when processing exceptions.

D_VSTEP – CSR 0x4046

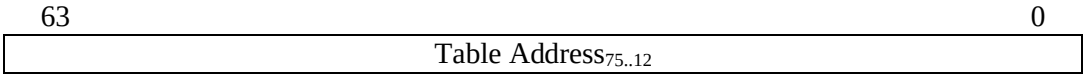
This register holds the current vector step number. It may need to be saved and restored during exception processing to ensure vector operations work as expected.

D_VTMP – CSR 0x4047

This register holds state for an internal temporary register used during vector processing. This register may need to be saved and restored during exception processing.

M_GDTB – CSR 0x3050

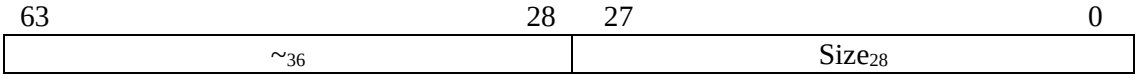
The GDTB register holds the base location of the global descriptor table. The descriptor table must be 4kB-byte aligned.



Note that the global descriptor table must be located in the low 76-bits of the physical address space. The address of the GDT is a physical address.

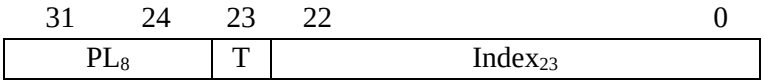
M_GDTL – CSR 0x3051

The GDTB register holds the size in bytes of the global descriptor table. The descriptor table must be 4kB-byte aligned. The global descriptor table has a maximum of 8388608 entries.



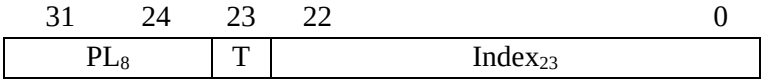
M_LDT – CSR 0x3052

The LDT register holds the selector for the local descriptor table. Register tag #153.



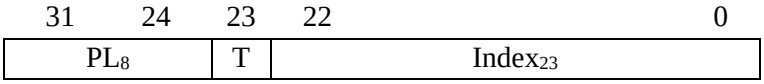
M_KYT – CSR 0x3053

The KYT register holds the selector for the memory key table. Register tag #154.



M_TCB - CSR 0x3054

This register holds the selector for the currently active task control block. Register tag #155.



Hardware Queues

There are sixteen hardware FIFO queues. Queue #15 is used to implement instruction tracing. The queues are accessible with the PUSHQ, POPQ, PEEKQ, and STATQ system instructions.

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.

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.

Polling for Interrupts

To support code that needs to run with interrupts disabled an interrupt polling instruction (PFI) is provided in the instruction set. For instance, the system could be running a high priority task with interrupts disabled. There may be sections of code where it is possible to process an interrupt however. In some code environments, it is not enough to disable and enable interrupts around critical code. The code must be effectively run with interrupt disabled all the time. This makes it necessary to poll for interrupts in software. For instance, stack prologue code may cause false pointer matches for the garbage collector because stack space is allocated before the contents are defined. If the GC scan occurs on this allocated but undefined area of memory, there could be false matches.

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 current register set, operating mode and interrupt enable bits are pushed onto an internal stack when an exception occurs. This stack is only eight entries deep as that is the maximum amount of nesting that can occur. Further nesting of exceptions can be achieved by saving the state contained in the exception registers.

Exception Vectoring

Exceptions are handled through a vector table. The vector table has four entries, one for each operating level the core may be running at. The location of the vector table is determined by TVEC[3]. If the core is operating at mode three for instance and an interrupt occurs vector table address number three is used for the interrupt handler. Note that the interrupt automatically switches the core to operating mode three. An exception handler at the machine level may redirect exceptions to a lower-level handler identified in one of the vector registers. More specific exception information is supplied in the cause register.

Operating Level	Address (If TVEC[3] contains $F…FC0000)	
0	$F…FC0000	Handler for operating level zero
1	$F…FC0020	
2	$F…FC0040	
3	$F…FC0060	

Reset

The core begins executing instructions at address $F…FC0100. All registers are in an undefined state. Register set #0 is selected.

Precision

Exceptions in Thor2021 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 reorder 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.

Exception Cause Codes

The following table outlines the cause code for a given purpose. These codes are specific to Thor2021. Under the HW column an ‘x’ indicates that the exception is internally generated by the processor; the cause code is hard-wired to that use. An ‘e’ indicates an externally generated interrupt, the usage may vary depending on the system.

Cause Code		HW	Description	
0			no exception	
1	IBE	x	instruction bus error	
2	EXF	x	Executable fault	
4	TLB	x	tlb miss	
			FMTK Scheduler	
128		e		
129	KRST	e	Keyboard reset interrupt	
130	MSI	e	Millisecond Interrupt	
131	TICK	e		
156	KBD	e	Keyboard interrupt	
157	GCS	e	Garbage collect stop	
158	GC	e	Garbage collect	
159	TSI	e	FMTK Time Slice Interrupt	
3			Control-C pressed	
20			Control-T pressed	

26			Control-Z pressed	
32	SSM	x	single step	
33	DBG	x	debug exception	
34	TGT	x	call target exception	
35	MEM	x	memory fault	
36	IADR	x	bad instruction address	
37	UNIMP	x	unimplemented instruction	
38	FLT	x	floating point exception	
39	CHK	x	bounds check exception	
40	DBZ	x	divide by zero	
41	OFL	x	overflow	
47				
48	ALN	x	data alignment	
49	KEY	x	memory key fault	
50	DWF	x	Data write fault	
51	DRF	x	data read fault	
52	SGB	x	segment bounds violation	
53	PRIV	x	privilege level violation	
54	CMT	x	commit timeout	
55	BT	x	branch target	
56	STK	x	stack fault	
57	CPF	x	code page fault	
58	DPF	x	data page fault	
60	DBE	x	data bus error	
61	PMA	x	physical memory attributes check fail	
62	NMI	x	Non-maskable interrupt	
225	FPX_IOP	x	Floating point invalid operation	
226	FPX_DBZ	x	Floating point divide by zero	
227	FPX_OVER	x	floating point overflow	
228	FPX_UNDER	x	floating point underflow	
229	FPX_INEXACT	x	floating point inexact	
231	FPX_SWT	x	floating point software triggered	
239			Software exception handling	
240	SYS		Call operating system (FMTK)	
241			FMTK Schedule interrupt	
242	TMR	x	system timer interrupt	
243	GCI	x	garbage collect interrupt	
253	RST	x	reset	
254	NMI	x	non-maskable interrupt	
255	PFI		reserved for poll-for-interrupt instruction	

DBG

A debug exception occurs if there is a match between a data or instruction address and an address in one of the debug address registers.

IADR

This exception is currently not implemented but reserved for the purpose of identifying bad instruction addresses. If the two least significant bits of the instruction address are non-zero then this exception will occur.

UNIMP

This exception occurs if an instruction is encountered that is not supported by the processor. It may also occur if there is an attempt to use an instruction in a mode that does not support it.

OFL

If an arithmetic operation overflows (multiply, add, or shift) and the overflow exception is enabled in the arithmetic exception enable register then an OFL exception will be triggered.

KEY

This fault will occur if an attempt is made to access memory for which the app does not have the key.

FLT

A floating-point exception is triggered if an exceptional condition occurs in the floating-point unit and the exception is enabled. Please see the section on floating-point for more details.

DRF, DWF, EXF

Data read fault, data write fault, and execute fault are exceptions that are returned by the memory management unit when an attempt is made to access memory for which the corresponding access type is not allowed. For instance, if the memory page is marked as non-executable an attempt is made to load the instruction cache from the page then an execute fault EXF exception will occur.

CPF, DPF

The code page fault and data page fault exceptions are activated by the mmu if the page is not present in memory. Access may be allowed but simply unavailable. These faults are not currently implemented.

PRIV

Some instructions and CSR registers are legal to use only at a higher operating level. If an attempt is made to use the privileged instruction by a lower operating level, then a privilege violation exception may occur. For instance, attempting to use RTI instruction from user operating level.

STK

If the value loaded into one of the stack pointer registers (the stack pointer sp or frame pointer fp) is outside of the bounds defined by the stack bounds registers, then a stack fault exception will be triggered.

DBE

A timeout signal is typically wired to the err_i input of the core and if the data memory does not respond with an ack_i signal fast enough an error will be triggered. This will happen most often when the core is attempting to access an unimplemented memory area for which no ack signal is generated. When the err_i input is activated during a data fetch, an exception is flagged in a result register for the instruction. The core will process the exception when the instruction commits. If the instruction does not commit (it could be a speculated load instruction) then the exception will not be processed.

PMA

The addressed memory did not pass the physical memory attributes testing. For example a write operation attempted to a ROM address space.

IBE

A timeout signal is typically wired to the err_i input of the core and if the instruction memory does not respond with an ack_i signal fast enough and error will be triggered. This will happen most often when the core is attempting to access an unimplemented memory area for which no ack signal is generated. When the err_i input is activated during an instruction fetch, a breakpoint instruction is loaded into the cache at the address of the error.

NMI

Non-maskable interrupt.

BT

The core will generate the BT (branch target) exception if a branch instruction points back to itself. Branch instructions in this sense include jump (JMP) and call (CALL) instructions.

Segmentation

Overview

Segmentation is a low overhead means of memory protection and virtualization. Providing separate protected address spaces for different applications is the job of the operating system. Ideally segmentation hardware should not be visible to the application. The application should appear as though it has a flat memory model. The core contains eight segment registers. The segmentation system is managed via a combination of hardware and software. Up to 256 privilege levels are available.

Privilege levels

Memory access is available according to privilege levels. The segmentation system allows up to 256 privilege levels.

Usage

The segment register to use during address formation for data addresses is identified by a field in the instruction. This field is set to default values by the assembler. For code addresses segment register #7 (the CS) is always used.

- If segmentation is not desired then segmentation can effectively be ignored by setting all the segment registers to zero. The processor can also be built without segmentation by commenting out the 'SEGMENTATION' definition.

Software Support

Segmentation is software supported. A software implementation allows a high degree of flexibility when implementing the segmentation model. Loading a value into a selector register causes a software segmentation exception to occur. The exception routine then loads the segment base, limit and access rights from a table in memory. It's up to the system level software to determine if protection rules are violated.

Segment registers may only be transferred to or from one of the general-purpose registers. The mtspr and mfspr instructions can be used to perform the move. A segment register may also be loaded using the LDIS instruction. After loading a segment register the instruction stream should be synchronized with a memory barrier (MEMSB) to ensure the segment value can be ready for a following memory operation.

There are two vectors in the vector table reserved for implementing far subroutine call and return instructions.

Address Formation:

Non-segmented address bits 0 to 11 pass through the segmentation module unchanged. Address bits 63 to 12 are added to the contents of the segment register to form the final segmented address. Note that there is no shift associated with the segment addition. Future implementations of the processor may include additional low order address bits in the segment register to allow a finer grain for memory page / paragraph size.

Address[63:12]	Address[11:0]
+	+
Segment register value[63:12]	000_{12}
=	
Segmented address[63:0]	

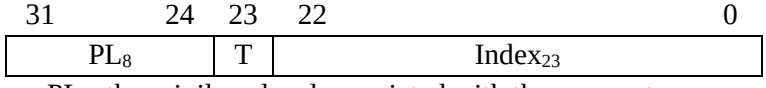
Selecting a segment register

A specific segment register for a memory operation may be selected using a segment prefix in assembler code. Segment prefixes apply to data addresses only. Code addresses always use segment register #7 – the code segment. The segment prefix indicator is encoded by a three-bit field in the instruction.

Selectors

The core uses selectors as a more compact way to represent segment registers. Rather than pass the entire segment descriptor to routines (256 bits) and have each routine check for privilege violations, the core uses 32-bit selectors. Privilege violations are checked for at the time the segment register components (base, limit and access rights) are loaded into the descriptor cache. The selector includes a field identifying the privilege level, and a second field identifying which segment descriptor the selector is associated with. The selector format is shown below.

Selector Format:



PL_8: the privilege level associated with the segment

Index$_{23}$: the index into the descriptor table

T: 0 = global, 1 = local descriptor table

Selector Registers

There are eighteen selector registers.

#	Reg	Usage
0	ZS	
1	DS	data selector
2	ES	
3	FS	
4	GS	
5	HS	
6	SS	stack selector
7	CS	code selector
8	PMA0	physical memory attributes
9	PMA1	
10	PMA2	
11	PMA3	
12	PMA4	
13	PMA5	
14	PMA6	
15	PMA7	
16	LDT	local descriptor table selector
17	KYT	memory key table selector

Descriptor Cache

If every memory access had to incur another memory access to load descriptor information processing speed would be adversely affected. To avoid additional memory access descriptors selected by selectors are cached in a descriptor cache for the selector register. The descriptor cache is only loaded when the value in the selector register is updated. Moving a value to a selector register with the MTSEL instruction causes the descriptor cache to be loaded from memory.

Non-Segmented Code Area

The address range defined as 64'hFxxxxxxxxxxxxxx (the top nibble is 'F') is a non-segmented code area. This area allows the operating system to work without paying attention to the code segment. Interrupt and exception vectors should vector into the non-segmented code area. The only way to change the code segment is by transferring to the operating system via a sys call instruction.

Changing the Code Segment

The only way to change the code segment is by transferring to the operating system via a sys call instruction. The operating system, while operating in the non-segmented code area, can alter the code segment without causing a transfer of control. The operating system establishes the code segment for a task while running in the non-segmented code area. To support far subroutine calls and returns there are vectors in the vector table that allow implementation of a far call or return.

The Descriptor Table

The descriptor table is a software managed table that contains information on the location and size for segments in the form of memory descriptors. Each descriptor is 32 bytes in size. Memory descriptor entries in the table have the following format:

	255 236	235 192	191 128	127 64	63 0
w0	ACR_{20}	$\sim_{44}$	$Limit_{64}$	$\sim_{64}$	$Base_{64}$
w1	ACR_{20}	$\sim_{44}$	$Limit_{64}$	$\sim_{64}$	$Base_{64}$
…					

The descriptor table may contain other types of descriptors beyond basic memory descriptors, such as call gates.

The base address of, and the number of entries in the descriptor table is contained in the LDT or GDT special purpose registers. The descriptor table may be updated with regular load and store instructions when the processor is at privilege level zero.

32-bit selectors are used to index into the table to determine the characteristics of the segment.

Memory Descriptors

Memory descriptors describe the location and size of memory segments. They have the following format:

n+3	ACR_{20}	$\sim_{44}$
n+2	$Limit_{63..0}$	
n+1	$\sim_{64}$	
n	$Base_{63..0}$	

The Access Rights Field (ACR_{16}) – Memory Descriptor

19	18	17	16	15	14	13	12	11 4	3	2	1	0

P	Sys	Stk	A	C	R	W	X	DPL_8	Con	U2	U1	U0

P: 1 = segment present, 0 = segment not present
Sys: 0 = system descriptor, 1 = memory descriptor
Stk: 1 = stack segment
A: 1= accessed
C: 1 = cacheable (ignored for executable segments which are always cached)
R: 1 = readable
W: 1 = writeable
X: 1 = executable, 0 = data
DPL_8 = descriptor privilege level
Con: 1 = conforming code segment
U2: available for OS use
U1: available for OS use
U0: available for OS use

Typical Values for ACR

8D000 – executable, readable code segment, privilege level zero
8E000 – read/writeable data segment, privilege level zero
AE000 – read / writeable stack segment, privilege level zero

Stack Segment Descriptors

Stack segment descriptors describe the location and limits of stack segments. They have the following format:

n+3	ACR_{20}	$Depth_{44}$
n+2	$Upper\ Limit_{63..0}$	
n+1	$\sim_{64}$	
n	$Base_{63..0}$	

A stack segment descriptor is almost the same as a memory segment descriptor except that it has a stack depth associated with it. Bit 17 of the ACR for the data descriptor is set. The lower limit of the stack segment is the upper limit minus the stack depth. If either bounds are exceeded a stack fault occurs rather than a bounds violation. This provides the capacity to expand the stack. One limitation of this mechanism is that the stack is limited to 44 address bits (16TB). Note that the stack is always word aligned so the upper and lower limits represent word boundaries.

System Segment Descriptors

System descriptors are identified by having bit 18 of the access rights set to one. There are potentially sixteen different system descriptor types.

The Access Rights Field (ACR_{20}) – System Descriptor

19	18	17	16	15	14	13	12	11	4	3	2	1	0

P	1	~	A	$Type_4$	DPL_8	~	U2	U1	U0

$Type_4$	Gate	
0	unused	
1	PTA descriptor	specifies location and size of root page table
2	LDT descriptor	specifies location and size of local descriptor table
3	KYT descriptor	specifies location and size of memory key table
4	Call gate	
5	Task Gate	
6	Interrupt Gate	
7	Trap gate	

PTA Descriptor

The PTA descriptor establishes the location and size of the root page table in memory. The base address must be 4kB aligned.

n+3	ACR_{20}	$\sim_{44}$				
n+2	$Limit_{63..0}$					
n+1	$\sim_{64}$					
n	$Base_{63..12}$		~	TD_3	S_1	$\sim_7$

LDT Descriptor

The LDT descriptor establishes the location and size of the local descriptor table in memory.

n+3	ACR_{20}	$\sim_{44}$
n+2	$\sim_{36}$	$Size_{27..0}$
n+1	$\sim_{64}$	
n	$Base_{63..0}$	

KYT Descriptor

The KYT descriptor establishes the location and size of the memory key table in memory.

n+3	ACR_{20}	$\sim_{44}$
n+2	$Limit_{63..0}$	
n+1	$\sim_{64}$	
n	$Base_{63..0}$	

Call Gate Descriptor

n+3	ACR_{20}	$\sim_{44}$	
n+2		N_5	$Selector_{31..0}$
n+1	$\sim_{64}$		
n	$Offset_{63..0}$		

Segment Load Exception

Moving a value to a selector register (a move to SPR #32 to 38,40) triggers a segment load exception to allow the segment descriptor to be loaded from one of the descriptor tables. This exception is triggered for a LDIS or MTSPR instruction. There is a separate exception vector (vectors #256 to 264) to handle each segment register. The selector value being loaded into the segment register is reflected in the ARG1 special purpose register.

Segment Bounds Exception

If an address is greater than or equal to the limit specified in the segment limit register then a segment limit exception occurs. This applies for all segments including code and data segments.

Segment Usage Conventions

Segment register #7 is the code segment (CS) register. All program counter addresses are formed with the code segment register unless the upper nibble of the address is ‘F’ in which case the code segment is ignored.

Segment register #6 is the stack segment (SS) register by convention. Future versions of the core may use this register implicitly for stack accesses. The assembler automatically selects the stack

segment when one of the stack pointer registers is specified in the instruction. Segment register #1 is the data segment (DS) by convention. The data segment is selected as the segment register for memory operations when the stack segment is not selected.

Power-up State

On reset the value in the segment registers are undefined. Note that the processor begins executing instructions out of the non-segmented code area as the reset address is 96'hFF000007_FFFFFFFFFFC0100. One of the first tasks of the boot program would be to initialize the segment registers to known values. The segment register must be setup to perform data accesses properly.

Segment Registers

Num		Long name	Comment
0	ZS	zero (NULL) segment	by convention contains zero
1	DS	data segment	by convention – default for loads/stores
2	ES	extra segment	by convention
3	FS		
4	GS		
5	HS		
6	SS	Stack segment	default for stack load/stores
7	CS	Code segment	always used for code addressing

TLB – Translation Lookaside Buffer

Overview

The page map is limited in the translations it can perform because of its size. The solution to allowing more memory to be mapped is to use main memory to store the translations tables, then cache address translations in a translation look-aside buffer or TLB. This is sometimes also called an address translation cache ATC. The TLB offers a means of address virtualization and memory protection. A TLB works by caching address mappings between a real physical address and a virtual address used by software. The TLB deals with memory organized as pages. Typically, software manages a paging table whose entries are loaded into the TLB as translations are required.

Size / Organization

The TLB has 1024 entries per set. The size was chosen as it is the size of one block ram for 32-bit data in the FPGA. This is quite a large TLB. Many systems use smaller TLBs. There is not really a need for such a large one, however it is available.

The TLB is organized as a four-way set associative cache.

What is Translated

The TLB processes all user mode addresses including both instruction and data addresses. It is known as a *unified* TLB. Addresses in other modes of operation are not translated. Additionally, virtual addresses with the top forty bits set are not translated to allow access to the BIOS and system rom.

Page Size

Because the TLB caches address translations it can get away with a much smaller page size than the page map can for a larger memory system. 4kB is a common size for many systems. In this case the TLB uses 4kB pages to match the size of pages for keyed memory and segmentation. For a 512MB system (the size of the memory in the test system) there are 131,072 4kB pages.

Management

The TLB unit is a software managed TLB. When a translation miss occurs, an exception is generated to allow software to update the TLB. It is left up to software to decide how to update the TLB. There may be a set of hierarchal page tables in memory, or there could be a hash table used to store translations.

The TLB is updated using the TLBRW instruction which both reads and writes the TLB. More descriptive text is present at the TLBRW instruction description.

Flushing the TLB

The TLB maintains the address space (ASID) associated with a virtual address. This allows the TLB translations to be used without having to flush old translations from the TLB during a task switch.

Global Bit

In addition to the ASID the TLB entries contain a bit that indicates that the translation is a global translation and should be present in every address space.

PAM – Page Allocation Map

Overview

Memory is organized into 131,072 4kB pages.

The PAM is a software structure made up of 131,072 bit-pairs stored in memory. There is a bit pair for each possible physical memory page. The PAM is used by software to manage the allocation of physical pages of memory.

Memory Usage

Total memory used by the PAM is 32kB.

Organization

The PAM is organized as a string of bit-pairs, one pair for each physical memory page. Bit pairs are used rather than single bits to mark allocated pages as it is convenient to also mark runs of pages. Marking runs of pages using bit-pairs makes it possible to free the pages of a previous allocation.

Bit-Pair Value	Meaning
0	Page of memory is free, available for use.
1	reserved
2	Page is allocated, end of run of pages
3	Page is allocated

PMA - Physical Memory Attributes Checker

Overview

The physical memory attributes checker is a hardware module that ensures that memory is being accessed correctly according to its physical attributes.

Physical memory attributes are stored in an eight-entry table. This table includes the address range the attributes apply to and the attributes themselves. Address ranges are resolved only to bit four of the address. Meaning the granularity of the check is 16 bytes.

Most of the entries in the table are hard-coded and configured when the system is built.

Physical memory attributes checking is applied in all operating modes.

Register Description

Regno	Bits		
00	64	LB0	lower bound - address bits 4 to 67 of the physical address range
08	64	UB0	upper bound - address bits 4 to 67 of the physical address range
10	16	AT0	memory attributes
18	~	~	reserved
…	…	…	6 more register sets
E0	64	LB7	lower bound - address bits 4 to 67 of the physical address range
E8	64	UB7	upper bound - address bits 4 to 67 of the physical address range
F0	16	AT7	memory attributes
F8	~	~	reserved

Attributes

Bitno		
0	X	may contain executable code
1	W	may be written to
2	R	may be read
3	C	may be cached
4-6	G	granularity

G	
0	byte accessible
1	wyde accessible
2	tetra accessible
3	octa accessible

		4 to 7	reserved	
7	~	reserved		
8-15	T	device type (rom, dram, eeprom, I/O, etc)		

Key Cache

Overview

Associated with each page of memory is a memory key. To access a page of memory the memory key must match with one of the keys in the applications keyset. The keyset is maintained in the keys CSRs. The key size of 20 bits is a minimum size recommended for security purposes.

The key associated with each memory page is stored in a table in main memory. Each key occupies a tetra-byte of memory to keep caching simple. So that two memory accesses are not required to access a page of memory this table of keys is cached. When a page of memory is accessed the key cache is accessed in parallel.

The key cache is a direct mapped cache organized as 256 lines of 16 keys. Key values are stored in LUT rams. 256 address tags are stored in LUT ram.

Card Memory

Overview

Also present in the memory system is Card memory. The card memory is a telescopic memory which reflects with increasing detail where in the memory system a pointer write has occurred. This is for the benefit of garbage collection systems. Card memory is updated using a write barrier when a pointer value is stored to memory.

Organization

Memory is divided into 256-byte card memory pages. Each card has a single byte recording whether a pointer store has taken place in the corresponding memory area. To cover a 512MB memory system 2MB card memory is required at the outermost layer. The outer most 2MB card memory layer is itself divided into 4096 256-byte card pages. Note that each byte represents the pointer store status for a 256B region. The 4096B memory is further resolved to single octa indicating if any pointer store has taken place. Thus, for a 512MB memory system a three-level card memory is used.

Layer	Resolving Power	
0	2 MB	256B regions
1	4 kB	128kB regions
2	8 B	64 MB regions

There is only a single card memory in the system, used by all tasks.

Location

Card memory must be based at physical address zero, extending up to the amount of card memory required. This is so that the address calculation of the memory update may be done with a simple right-shift operation.

Operation

As a program progresses it writes pointer values to memory using the write barrier. Storing a pointer triggers an update to all the layers of card memory corresponding to the main memory location written. A byte is set in each layer of the card memory system corresponding to the memory location of the pointer store.

The garbage collection system can very quickly determine where pointer stores have occurred and skip over memory that has not been modified.

Sample Write Barrier

```
; Milli-code routine for garbage collect write barrier.
; Usable with up to 64-bit memory systems.
; Three level card memory
;
GCWriteBarrier:
        STO     a0,[a1]         ; store the value to memory at a1
        SRL     a1,a1,#8        ; compute card address
        STB     x0,[a1]         ; clear bit in card memory
        SRL     a1,a1,#8        ; repeat for each table level
        STB     x0,[a1]
        SRL     a1,a1,#8
        STB     x0,[a1]
        ;… more stores as needed
        JMP     ra1
```

System Memory Map

There are several components to the system which use tables in memory. These tables are statically allocated at the time the system is built. The table sizes depend on the size of main memory. The card memory table must be located at address zero. So, it is probably best to group the tables together at the low end of memory.

Address	Usage	
$00000000 to $001FFFFF	Card Memory (2 MB)	
$00210000 to $0022FFFF	PAM (128kB 2 copies)	

$00280000 to $0029FFFF	Key memory (128 kB)	

Debugging Unit

Overview

The Thor2021 has several debug features including debug exceptions on address matches and instruction tracing. Instruction trace trigger registers are shared with the debug address registers. Which function is triggered on an address match is controlled in the debug control register.

Instruction Tracing

Instruction tracing is enabled by setting the trace enable bit (bit 32 to 35) for the corresponding debug address match register. Tracing will begin when an address match occurs and continue until the trace buffer is full. The trace queue is 8kB in size allowing thousands of instructions to be traced.

Trace Queue Entry Format

The trace queue stores both complete instruction pointer addresses and branch taken-not-taken (TNT) history. The low order two bits of the trace entry indicate the type of record stored by the entry. There are currently two record types. Record type zero is an instruction pointer address. Record type one is a history record for branches.

111			2	1	0
				Rectype$_2$	
~$_{14}$	Selector Value$_{32}$	Instruction Pointer bits 0 to 63		00	

111		9	8	2	1	0
Branch Taken-Not-Taken History$_{103}$			count$_7$		01	

Up to 103 bits of branch TNT history may be stored in a single record. The number of bits stored is recorded in bits 2 to 8 of the record. After four full branch TNT history records, the trace will record the current instruction address in whole.

Trace Readback

A trace of instructions executed may be read back from the trace queue using the PEEKQ and POPQ instructions. The processor trace queue is accessible as queue number 15. Queue 15 contains the raw history record. Software should get the status using the STATQ instruction to see if data is available, then pop queue 15 to get the data record.

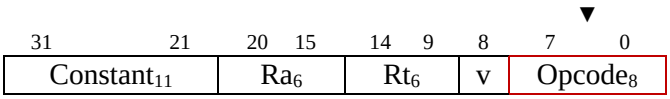
Instruction Set Description

Overview

Like the original Thor core, the instruction set is variable length. Instructions vary from two to eight bytes in length. Commonly used instructions often have short forms. While adding complexity to the processor, variable length instructions make better use of the cache.

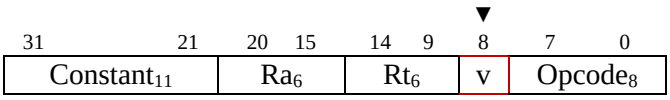
Root Opcode

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 eight bits of the instruction. The instruction length is determined entirely by the value of the root opcode.



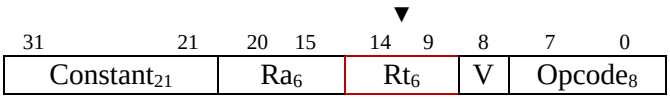
Vector Instruction Indicator

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 Thor2021 encodes this information in bit 8 of all instructions. If bit 8 is a '1' then the instruction is a vector instruction. See the sample instruction below.



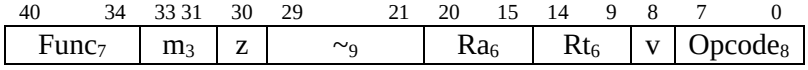
Target Register Spec

Most instructions have a target register. The register spec for the target register is usually in the same position, bits 9 to 14 of an instruction. If the instruction is a vector instruction, then the target register will be a vector register, otherwise it is a scalar register.

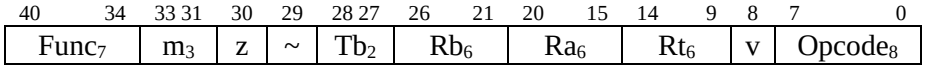


Register Formats

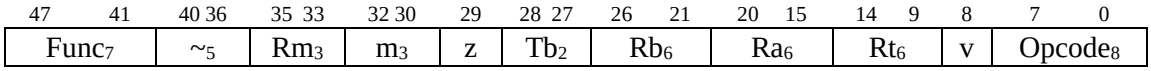
R1 (one source register)



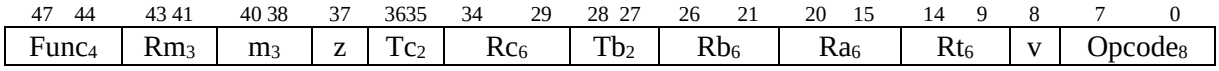
R2 (two source register)



R2L (two source register)



R3 (three source register)



Clock Cycles

Clock cycles given for instructions are approximate and represent a relative relationship between instructions. An instruction with a given clock cycle count of two will take approximately twice as long to execute as an instruction with a given clock cycle count of one. Generally, a clock cycle count of one means the instruction requires only a single clock to execute. However, the effective execution time of the instruction may be less if multiple execution units can execute the instruction or if execution of the instruction may be overlapped with execution of other instructions.

Execution Units

Most instructions execute on a particular type of execution unit. For instance, the ADD instruction is executed on an ALU while the LDB instruction is executed on a memory unit. The execution unit for the instruction is listed in the instruction's description. The execution of some instructions is supported on only a single execution unit even if multiple execution units of the same type are available. For instance, infrequently used instructions like bitfield manipulations or sub-word shifts and rotates are supported on only a single ALU. Instructions that execute on two different execution units may execute at the same time. This is important for instruction scheduling.

Arithmetic / Logical / Shift

ABS – Absolute Value

Description:

This instruction takes the absolute value of a register and places the result in a target register. Both the source and target registers are treated as integer values.

Integer Instruction Format: R1

40 34	33 31	30	29 21	20 15	14 9	8	7 0
$06h_7$	m_3	z	$\sim_9$	Ra_6	Rt_6	v	$01h_8$

v: 0 = scalar, 1 = vector op

Operation:

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

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Rt[x] = Ra[x] < 0 ? -Ra[x] : Ra[x]
    else if (z) Rt[x] = 0
    else Rt[x] = Rt[x]
```

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

ADD[O] - Register-Register

Description:

Add two registers and place the sum in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The mask register is ignored for scalar instructions. All registers are integer registers. If the 'O' bit of the instruction is set then signed overflow is tested.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	m_3	z	O	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 1

Execution Units: All Integer ALU's

Operation:

Rt = Ra + Rb

Exceptions: overflow, if enabled

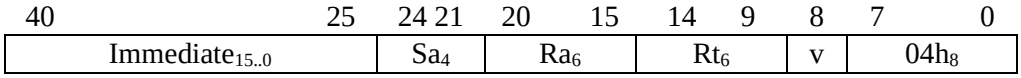
Notes:

ADDI - Add Immediate

Description:

Add a register and immediate value and place the sum in the target register. The immediate value is shifted to the left by a multiple of 16 bits before the addition takes place. The immediate is sign extended to the machine width and padded with zeros on the right. This instruction may be used to build a large constant.

Instruction Format: RI



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + (immediate << (Sa * 16))

Exceptions:

Notes:

AND – Bitwise And

Description:

Perform a bitwise ‘and’ operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$08h_7$	m_3	z	~	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

ANDC – Bitwise ‘And’ with Complement

Description:

Perform a bitwise ‘and’ with complement operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$0Bh_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Operation:

Rt = Ra & ~Rb

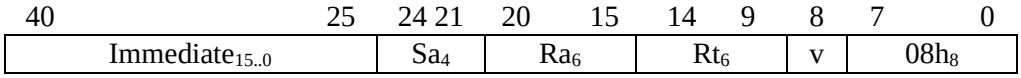
Exceptions: none

ANDI – Bitwise ‘And’ Immediate

Description:

Bitwise ‘and’ a register and immediate value and place the result in the target register. The immediate value is shifted to the left by a multiple of 16 bits before the addition takes place. The immediate is one extended to the machine width and padded with ones on the right. This instruction may be used to ‘and’ a large constant.

Instruction Format: RIS



Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra & (immediate << (Sa * 16))

Exceptions:

Notes:

ANDM – Bitwise ‘And’ with Mask

Description:

Generates a mask consisting of ‘1’s from the mask begin, Mb_6, to the mask end Me_6, inclusive. Perform a bitwise ‘and’ operation between register Ra and the mask and store the result in register Rt. The immediate constant is one extended before use.

Instruction Format: RM

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$8h_4$	m_3	z	Me_6	Mb_6	Ra_6	Rt_6	v	AAh_7

Operation

Rt = Ra & Immediate

Vector Operation

for x = 0 to VL-1

if (Vm0[x]) Vt[x] = Va[x] & Immediate

else Vt[x] = Vt[x]

Exceptions: none

BCDADD – BCD Add

Description:

Adds two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is an eight-bit BCD number.

Instruction Format:

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$00h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$F5h_8$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra + Rb

Exceptions: none

BCDMUL – BCD Multiply

Description:

Multiply two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is a sixteen-bit BCD number.

Instruction Format:

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$02h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$F5h_8$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra * Rb

Exceptions: none

BCDSUB – BCD Subtract

Description:

Subtract two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is an eight-bit BCD number.

Instruction Format:

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$01h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$F5h_8$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra - Rb

Exceptions: none

BFALIGN – Bitfield Align

Description:

Align the contents of register Ra with a bitfield specified between mask begin and mask end.

Integer Instruction Format: RM

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$0h_4$	m_3	z	Me_6	Mb_6	Ra_6	Rt_6	v	AAh_7

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

Operation

Rt = (Ra << Mb) & Immediate Mask

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = (Va[x] << Mb) & Immediate Mask
    else Vt[x] = Vt[x]
```

Exceptions: none

BFCHG – Bitfield Change

Description:

Flip the bits of a bitfield specified between mask begin and mask end.

Integer Instruction Format: RM

Generates a mask consisting of '1's between the mask begin, Rc, and mask end, Me_6, inclusive. Bitwise exclusive 'or' the mask with the value in register Ra and store the result in register Rt. This instruction is useful for flipping bit fields.

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
Ah_4	m_3	z	Me_6	Mb_6	Ra_6	Rt_6	v	AAh_7

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

Operation

Rt = Ra ^ Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] ^ Immediate
    else Vt[x] = Vt[x]
```

Exceptions: none

BFCLR – Bitfield Clear

Description:

Clear the bits of a bitfield specified between mask begin and mask end.

Integer Instruction Format: RM

Generates a mask consisting of '1's between the mask begin, Rc, and mask end, Me_6, inclusive. Bitwise and with the complement of the mask with the value in register Ra and store the result in register Rt.

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
Bh_4	m_3	z	Me_6	Mb_6	Ra_6	Rt_6	v	AAh_7

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

Operation

Rt = Ra & ~Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] & ~Immediate
    else Vt[x] = Vt[x]
```

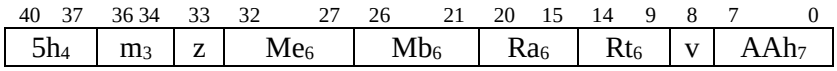
Exceptions: none

BFEXT – Bitfield Extract

Description:

Extract the bits of a bitfield specified beginning at mask begin and ending at mask end and sign extend and right align the result in the target register.

Integer Instruction Format: RM



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

Operation

Rt = sign extend ((Ra & Immediate) >> Mb)

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = sign extend ((Va[x] & Immediate) >> Mb)
    else Vt[x] = Vt[x]
```

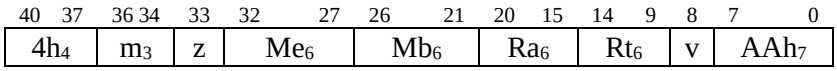
Exceptions: none

BFEXTU – Bitfield Extract Unsigned

Description:

Extract the bits of a bitfield specified beginning at mask begin and ending at mask end and right align the result in the target register.

Integer Instruction Format: RM



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

Operation

Rt = (Ra & Immediate) ror by Mb

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = (Va[x] & Immediate) ror by Mb
    else Vt[x] = Vt[x]
```

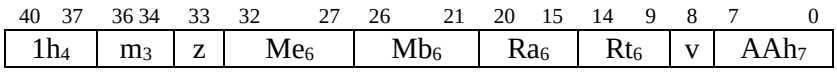
Exceptions: none

BFFFO –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.

Instruction Format: BF



Clock Cycles:

Execution Units: Integer

Exceptions: none

BFSET – Bitfield Set

Description:

Set the bits of a bitfield specified between mask begin and mask end.

Integer Instruction Format: RM

Generates a mask consisting of '1's between the mask begin, Mb_6, and mask end, Me_6, inclusive. Bitwise 'or' the mask with the value in register Ra and store the result in register Rt.

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$9h_4$	m_3	z	Me_6	Mb_6	Ra_6	Rt_6	v	AAh_7

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

Operation

Rt = Ra | Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] | Immediate
    else Vt[x] = Vt[x]
```

Exceptions: none

BMAP – Byte Map

Description:

Bytes are mapped from the 16-byte source Rb, Ra into bytes in the target register. This instruction may be used to permute the bytes in register pair Rb, Ra and store the result in Rt. This instruction may also pack bytes, wydes or tetras. The map is determined by the low order 32-bits of register Rc.

Integer Instruction Format: R3

40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$4Ch_8$

Rc value:

3128	2724	2320	1916	1512	11 8	7 4	3 0	
$B7_4$	$B6_4$	$B5_4$	$B4_4$	$B3_4$	$B2_4$	$B1_4$	$B0_4$	<= Target byte = Bn

Operation:

Vector Operation

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

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

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$Func_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	02_8

Fn_7	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]) | (Ra[i][1]&Rb[1][j]) | … | (Ra[i][7]&Rb[7][j])
```

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: none

Notes:

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

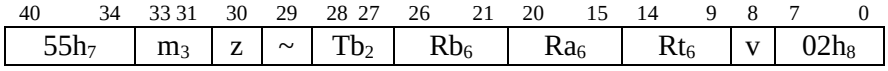
BYTNDX – Byte Index

Description:

This instruction searches Ra, which is treated as an array of eight bytes, for a byte value specified by Rb 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: RI



Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Rb in Ra)

Exceptions: none

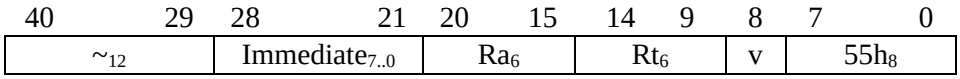
BYTNDXI – Byte Index

Description:

This instruction searches Ra, which is treated as an array of eight bytes, for a byte value specified by 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: RI



Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Imm_8 in Ra)

Exceptions: none

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 Rc then an exception will occur.

Instruction Format: R3

40 38	37	36 35	34 29	28 27	26 21	20 15	14 12	11 9	8	7 0
m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	$\sim_3$	cn_3	v	$4Dh_8$

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)

Clock Cycles: 1

Exceptions: bounds check

Notes:

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

CHKI – 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 Format: CHKI

40 29	28 27	26 21	20 15	14 12	11 9	8	7 0
$Immediate_{12}$	Tb_2	Rb_6	Ra_6	Imm_3	cn_3	v	$45h_8$

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

Exceptions: bounds check

Notes:

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

A seven-bit immediate value may be specified by Tb_2 and Rb.

CLMUL – Carry-less Multiply

Description:

Compute the low order product bits of a carry-less multiply. Both operands must be in registers.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$2Eh_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

4 clock cycles

Exceptions: none

Execution Units: 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

CLMULH – Carry-less Multiply High

Description:

Compute the high order product bits of a carry-less multiply. Both operands must be in registers.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$2Fh_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

4 clock cycles

Exceptions: none

Execution Units: 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

CMOVNZ – Conditional Move

Description:

CMOVNZ moves a value from Rb or Rc depending on the value in Ra. If Ra is non-zero then Rb is moved to Rt, otherwise Rc is moved to Rt.

For the vector version of the instruction Ra is a scalar. Each bit of Ra is used corresponding to the vector element to determine if the move takes place. This instruction may be used to convert a set of bits in Ra into a Boolean vector.

Instruction Format: R3

40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$54h_8$

Vector Operation

For x = 0 to VL-1

if (Vm[x]) Vt[x] = Ra.bit[x] ? Rb : Rc

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

Exceptions: none

Execution Units: integer ALU

CMP – Compare

Description

Compare two registers and return the relationship between them. Both values are treated as signed numbers.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$2Ah_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

1 clock cycle

Operation:

Rt = Ra < Rb ? –1 : Ra = Rb ? 0 : 1

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] < Vb[x] ? –1 : Va[x]=Vb[x] ? 0 : 1
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

CMPI – Compare Immediate

Description

Compare a register and an immediate value and return the relationship between them. The immediate value is shifted to the left by a multiple of 16 bits before the compare takes place. The immediate is sign extended to the machine width and padded with zeros on the right.

Both values are treated as signed numbers.

Instruction Format: RI

40 25	24 21	20 15	14 9	8	7 0
$Immediate_{15..0}$	Sa_4	Ra_6	Rt_6	v	$50h_8$

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

Operation:

Rt = Ra < Imm ? –1 : Ra = Imm ? 0 : 1

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] < Imm ? –1 : Va[x]=Imm ? 0 : 1
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

CMPU – Compare Unsigned

Description

Compare two registers and return the relationship between them. Both values are treated as unsigned numbers.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$2Bh_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

1 clock cycle

Operation:

Rt = Ra < Rb ? –1 : Ra = Rb ? 0 : 1

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] < Vb[x] ? –1 : Va[x]=Vb[x] ? 0 : 1
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

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

Instruction Format: R1

40 34	33 31	30	29 21	20 15	14 9	8	7 0
$02h_7$	m_3	z	$\sim_9$	Ra_6	Rt_6	v	$01h_8$

Execution Units: integer ALU

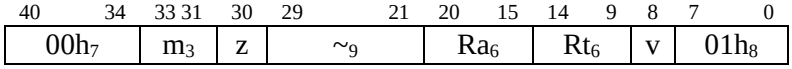
Exceptions: none

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



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. This is an alternate mnemonic for the BFCHG function.

Instruction Format: RM

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
Ah_4	m_3	z	63_6	0_6	Ra_6	Rt_6	v	AAh_7

1 clock cycle

Operation

Rt = ~Ra

Vector Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = ~Va[x]
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

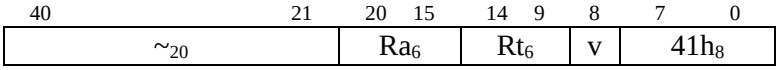
Exceptions: none

CPUID – CPU Identification

Description:

This instruction returns general information about the core. Register Ra is used as a table index to determine which row of information to return.

Instruction Format:



Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Info[Ra]

Exceptions: none

Index	bits	Information Returned
0	63 to 0	The processor core identification number. This field is determined from an external input. It would be hard wired to the number of the core in a multi-core system.
2	63 to 0	Manufacturer name first eight chars “Finitron”
3	63 to 0	Manufacturer name last eight characters
4	63 to 0	CPU class “64BitSS”
5	63 to 0	CPU class
6	63 to 0	CPU Name “Thor2021”
7	63 to 0	CPU Name
8	63 to 0	Model Number “M1”
9	63 to 0	Serial Number “1234”
10	63 to 0	Features bitmap
11	31 to 0	Instruction Cache Size (32kB)
11	63 to 32	Data cache size (16kB)

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

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$18h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Supported Formats: .o

Clock Cycles: 1

Execution Units: Integer

Operation:

Rt = Abs(Ra - Rb)

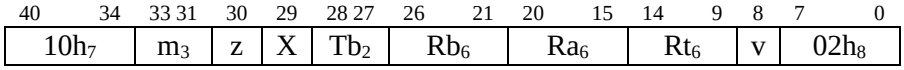
Exceptions: none

DIV[X] – Division

Description:

Divide two operand values and place the result in the target register. Both operands are in registers. Both operands are treated as signed values. This instruction may cause a divide by zero or overflow exception if enabled. The exception is enabled by the ‘X’ bit of the instruction.

Instruction Format: R2



Execution Units: ALU

Clock Cycles: 20

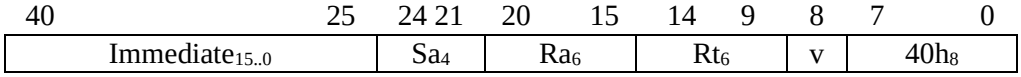
Exceptions: DBZ, OFL, if enabled

DIVI – Divide by Immediate

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 is an immediate value. Both operands are treated as signed values.

Instruction Format: RI



Execution Units: ALU

Clock Cycles: 20

Exceptions: none

Notes:

DIVU – Divide Unsigned

Description:

Divide two operand values and place the result in the target register. Both operands are in registers. Both operands are treated as unsigned values.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$11h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Execution Units: ALU

Clock Cycles: 20

Exceptions: none

DIVUI – Divide Unsigned by Immediate

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 is an immediate value. Both operands are treated as unsigned values.

Instruction Format: RI

40 25	24 21	20 15	14 9	8	7 0
$Immediate_{15..0}$	Sa_4	Ra_6	Rt_6	v	$4Fh_8$

Execution Units: ALU

Clock Cycles: 20

Exceptions: none

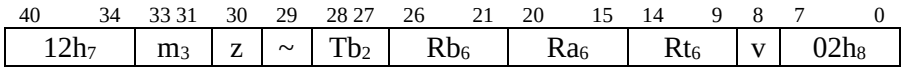
Notes:

DIVSU – Divide Signed by Unsigned

Description:

Divide two operand values and place the result in the target register. Both operands are in registers. The register operand Ra is a signed value, Rb is an unsigned value.

Instruction Format: R2



Execution Units: ALU

Clock Cycles: 20

Exceptions: none

ENOR – Bitwise Exclusive Nor

Description:

Perform a bitwise ‘nor’ operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$02h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Exceptions: none

EOR – Bitwise Exclusive Or

Description:

Perform a bitwise ‘or’ operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$0Ah_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Exceptions: none

EORI – Bitwise Exclusive 'Or' Immediate

Description:

Perform a bitwise exclusive 'or' operation between operands. The immediate value is shifted to the left by a multiple of 16 bits before the addition takes place. The immediate is zero extended to the machine width and padded with zeros on the right. This instruction may be used to form a large constant.

Instruction Format: RI

40 25	24 21	20 15	14 9	8	7 0
$Immediate_{15..0}$	Sa_4	Ra_6	Rt_6	v	$0Ah_8$

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

Operation

Rt = Ra ^ Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] ^ Immediate
    else Vt[x] = Vt[x]
```

Exceptions: none

LDI – Load Immediate

Description:

This is an alternate mnemonic for the ADDI instruction where register Ra is zero. Add register zero and a sign extended immediate value and place the sum in the target register. For the vector instruction both Ra and Rt are vector registers. If the Ra register field is zero then the value zero is used for Ra.

Instruction Format: RI

40 25	24 21	20 15	14 9	8	7 0
$Immediate_{15..0}$	Sa_4	0_6	Rt_6	v	$04h_8$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = immediate

Exceptions:

Notes:

MAX – Maximum Value

Description:

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

Instruction Format:

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$29h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
IF Ra < Rb
        Rt = Rb
else
        Rt = Ra
```

MIN – Minimum Value

Description:

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

Instruction Format:

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$28h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
IF Ra < Rb
        Rt = Ra
else
        Rt = Rb
```

MKBOOL – Make Boolean

Description:

This instruction is an alternate mnemonic for the CMOVNZ instruction. This instruction places a zero in the target register if the source register is zero, otherwise one is placed in the target register. This instruction reduces the source operand to a Boolean value.

Integer Instruction Format: R3

40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
m_3	z	2_2	0_6	2_2	1_6	Ra_6	Rt_6	v	$54h_8$

Operation:

Rt = Ra

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

MOV – Move Register-Register

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves one general purpose register to another. This instruction uses one less register port than using the OR instruction to move between registers.

Instruction Format:

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$5h_4$	m_3	z	63_6	0_6	Ra_6	Rt_6	v	AAh_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVSXB – Move Byte, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a byte from one general purpose register to another. This instruction uses one less register port than using the OR instruction to move between registers.

Instruction Format:

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$5h_4$	m_3	z	7_6	0_6	Ra_6	Rt_6	v	AAh_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVSXT – Move Tetra, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a tetra from one general purpose register to another. This instruction uses one less register port than using the OR instruction to move between registers.

Instruction Format:

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$5h_4$	m_3	z	31_6	0_6	Ra_6	Rt_6	v	AAh_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVSXW – Move Wyde, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a wyde from one general purpose register to another. This instruction uses one less register port than using the OR instruction to move between registers.

Instruction Format:

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$5h_4$	m_3	z	15_6	0_6	Ra_6	Rt_6	v	AAh_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVZXB – Move Byte, Zero Extend

Description:

This is an alternate mnemonic for the BFEXTU instruction. This instruction moves a byte from one general purpose register to another. This instruction uses one less register port than using the OR instruction to move between registers.

Instruction Format:

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$4h_4$	m_3	z	7_6	0_6	Ra_6	Rt_6	v	AAh_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVZXT – Move Tetra, Zero Extend

Description:

This is an alternate mnemonic for the BFEXTU instruction. This instruction moves a tetra-byte from one general purpose register to another. This instruction uses one less register port than using the OR instruction to move between registers.

Instruction Format:

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$4h_4$	m_3	z	31_6	0_6	Ra_6	Rt_6	v	AAh_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVZXW – Move Wyde, Zero Extend

Description:

This is an alternate mnemonic for the BFEXTU instruction. This instruction moves a wyde from one general purpose register to another. This instruction uses one less register port than using the OR instruction to move between registers.

Instruction Format:

40 37	36 34	33	32 27	26 21	20 15	14 9	8	7 0
$4h_4$	m_3	z	7_6	0_6	Ra_6	Rt_6	v	AAh_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

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: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$06h_7$	m_3	z	O	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

4 clock cycles

Exceptions: none

Execution Units: ALU

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]
```

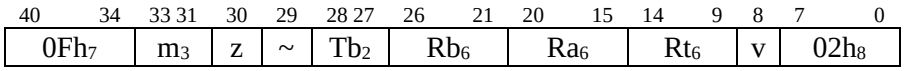
Exceptions: overflow, if enabled

MULH – Multiply High

Description:

Compute the high order product of two values. Both operands must be in registers. Both the operands are treated as signed values, the result is a signed result.

Integer Instruction Format: R2



4 clock cycles

Exceptions: none

Execution Units: ALU

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]
```

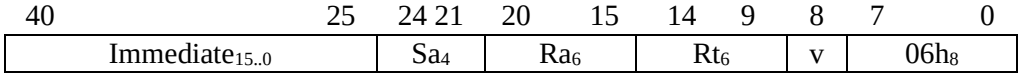
Exceptions: none

MULI – Multiply Immediate

Description:

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

Integer Instruction Format: RI



4 clock cycles

Execution Units: ALU

Operation:

Rt = Ra * Immediate

Vector Operation

```
for x = 0 to VL - 1
    if (Vm0[x]) Vt[x] = Va[x] * Vb[x]
    else Vt[x] = Vt[x]
```

Exceptions: none

MULF – Fast Unsigned Multiply

Description:

Multiply two values located in registers Ra and Rb. 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: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$15h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Execution Units: ALU

Clock Cycles: 1

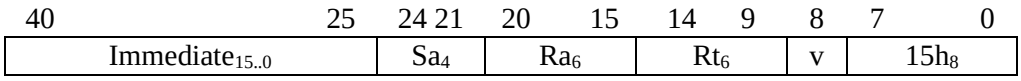
Exceptions: none

MULFI – Fast Unsigned Multiply Immediate

Description:

Multiply two values. The first operand is in register Ra. The second operand is 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



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

Execution Units: ALU

Clock Cycles: 1

Exceptions: none

MUX – Multiplex

Description:

If a bit in Ra is set then the bit of the target register is set to the corresponding bit in Rb, otherwise the bit in the target register is set to the corresponding bit in Rc.

Instruction Format:

40 38	37	3635	34 29	28 27	26 21	20 15	14 11	8	7 0
m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$64h_8$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

For n = 0 to 63
    If $Ra_{[n]}$ is set then
        $Rt_{[n]} = Rb_{[n]}$
    else
        $Rt_{[n]} = Rc_{[n]}$

Exceptions: none

NAND – Bitwise Nand

Description:

Perform a bitwise ‘nand’ operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$00h_7$	m_3	z	~	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

NEG - Negate

Description:

This is an alternate mnemonic for the SUB instruction where Ra is zero.

This instruction takes the negative of a value contained in a register Rb.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	m_3	z	~	Tb_2	Rb_6	0_6	Rt_6	v	$02h_8$

Scalar Operation

Rt = - Rb

Vector Operation

for x = 0 to VL - 1

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

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

Notes

Exceptions:

NOR – Bitwise Nor

Description:

Perform a bitwise ‘nor’ operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$01h_7$	m_3	z	~	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

NOT – Logical Not

Description:

This instruction is an alternate mnemonic for the CMOVNZ instruction. This instruction places a one in the target register if the source register is zero, otherwise zero is placed in the target register. This instruction reduces the source operand to a Boolean value.

Integer Instruction Format: R3

40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
m_3	z	2_2	1_6	2_2	0_6	Ra_6	Rt_6	v	$54h_8$

Operation:

Rt = !Ra

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

OR – Bitwise Or

Description:

Perform a bitwise ‘or’ operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$09h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Execution Units: All Integer ALU’s

Exceptions: none

ORC – Bitwise ‘Or’ with Complement

Description:

Perform a bitwise ‘or’ with complement operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$03h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Operation:

Rt = Ra | ~Rb

Execution Units: All Integer ALU’s

Exceptions: none

ORI – Bitwise ‘Or’ Immediate

Description:

Perform a bitwise or operation between operands. The immediate value is shifted to the left by a multiple of 16 bits before the operation takes place. The immediate is zero extended to the machine width and padded with zeros on the right. This instruction may be used to ‘or’ a large constant or build a 64-bit constant in a register.

Instruction Format: RI

40 25	24 21	20 15	14 9	8	7 0
$Immediate_{15..0}$	Sa_4	Ra_6	Rt_6	v	$09h_8$

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

Operation

Rt = Ra | Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] | Immediate
    else Vt[x] = Vt[x]
```

Execution Units: All Integer ALU’s

Exceptions: none

ORN – Bitwise ‘Or’ with Complement

Description:

This is an alternate mnemonic for the ORC instruction. Perform a bitwise ‘or’ with complement operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$03h_7$	m_3	z	~	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

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. Sc may vary from zero to fifteen.

Instruction Format: R3

40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$43h_8$

Operation:

$Rt = Abs(Ra - Rb) >> Rc_{[3:0]}$

Clock Cycles: 1

Execution Units: Integer

Exceptions:

None

REVBIT – Reverse Bit Order

Description:

This instruction reverses the order of bits in Ra and stores the result in Rt. Bits may be reversed in individual bytes, wydes, tetras or octas.

Integer Instruction Format: R1

40 36	35 34	33 31	30	29 21	20 15	14 9	8	7 0
$0Ah_5$	Sz_2	m_3	z	$\sim_9$	Ra_6	Rt_6	v	$01h_8$

v: 0 = scalar, 1 = vector op

Sz_2	Ext.	Meaning
0	.BP	reverse order within bytes (byte parallel)
1	.WP	reverse order within wydes (wyde parallel)
2	.TP	reverse order within tetras (tetra parallel)
3	.OP	reverse order within octas (octa parallel)

Operation:

Vector Operation

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

ROL – Rotate Left

Description:

Left rotate an operand value by an operand value and place the result in the target register. 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

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$43h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ROR – Rotate Right

Description:

Right rotate an operand value by an operand value and place the result in the target register. 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

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$44h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

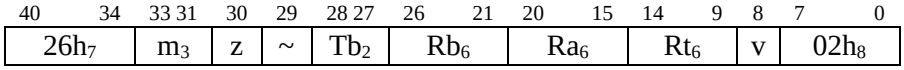
Example:

SEQ – Set if Equal

Description:

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

Instruction Format: R2

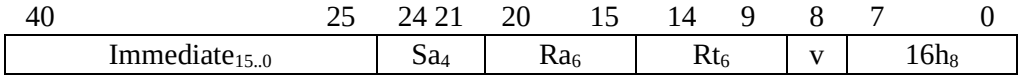


SEQI – Set if Equal Immediate

Description:

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

Instruction Format: RI



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

SGE / SLE – 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.

This instruction may also be used to test less than or equal to by swapping the operands.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$21h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

SGEU / SLEU – 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 unsigned values.

This instruction may also be used to test less than or equal to by swapping the operands.

Instruction Format: R2

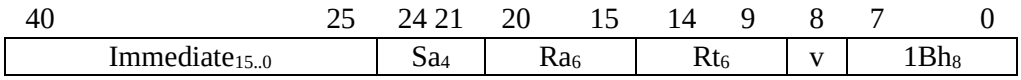
40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$23h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

SGTI – Set if Greater Than Immediate

Description:

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

Instruction Format: RI



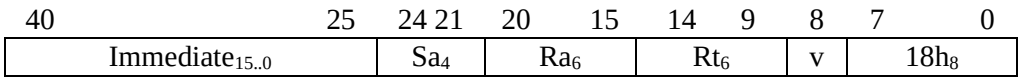
1 clock cycle / N clock cycles (N = vector length)

SLEI – Set if Less Than or Equal Immediate

Description:

This instruction is an alternate mnemonic for the SLTI instruction where the constant has been adjusted by one. For instance, SLEI $t0,$a0,#4 is the same as SLTI $t0,$a0,#5. The assembler will adjust the constant and use the SLTI instruction.

Instruction Format: RI



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

SLL[O] –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. Overflow may be detected if enabled and the sign of the result is different than the sign of Ra.

Instruction Formats: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$40h_7$	m_3	z	O	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Operation Size: .o

Execution Units: integer ALU

Exceptions: overflow, if enabled

Example:

SLT / SGT – 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 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.

Instruction Format: R2

40 – 34	33 – 31	30	29	28 – 27	26 – 21	20 – 15	14 – 9	8	7 – 0
$20h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

SLTI – Set if Less Than Immediate

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, which is an immediate constant then the target register is set to a one, otherwise the target register is set to a zero.

Instruction Format: RI

40 – 25	24 – 21	20 – 15	14 – 9	8	7 – 0
$Immediate_{15..0}$	Sa_4	Ra_6	Rt_6	v	$18h_8$

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

SLTU / SGTU – Set if Less Than Unsigned

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than a second operand in 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.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$22h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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) then the target register is set to a one, otherwise the target register is set to a zero.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$27h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

SNEI – Set if Not Equal Immediate

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, which is an immediate constant then the target register is set to a one, otherwise the target register is set to a zero.

Instruction Format: RI

40 25	24 21	20 15	14 9	8	7 0
$Immediate_{15..0}$	Sa_4	Ra_6	Rt_6	v	$17h_8$

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

SRA –Shift Right Arithmetic

Description:

Right shift an operand value by an operand value and place the result in the target register. The most significant bit is preserved. 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

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$42h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$41h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

SUB – Subtract

Description:

Subtract Rb from Ra and place the difference in the target register Rt. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The mask register is ignored for scalar instructions.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra - Rb

Exceptions: none

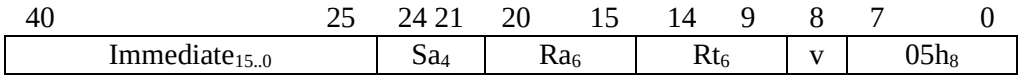
Notes:

SUBFI – Subtract from Immediate

Description:

Subtract a register from a sign extended immediate value and place the difference in the target register. For the vector instruction both Ra and Rt are vector registers.

Instruction Format: RI



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Immediate - Ra

Exceptions:

Notes:

UTF21NDXI – UTF21 Index

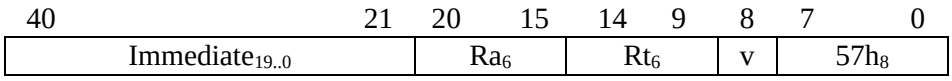
Description:

This instruction searches Ra, which is treated as an array of three UTF21 character, for a value specified by an immediate value and places the index of the value 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 character. The index result may vary from -1 to +2. The index of the first found character is returned (closest to zero).

If a vector UTF21NDX 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 +191 in the scalar register. Thus, UTF21NDX may be used to determine the length of a null termination string in the vector register.

Only the first 2^{20} UTF21 characters may be searched for by the immediate form of the instruction.

Instruction Format: RI



Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Imm_{21} in Ra)

Exceptions: none

UTF21NDX – UTF21 Index

Description:

This instruction searches Ra, which is treated as an array of three UTF21 characters, for a 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 character. The index result may vary from -1 to +2. The index of the first found character is returned (closest to zero).

If a vector UTF21NDX 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 +191 in the scalar register. Thus, UTF21NDX may be used to determine the length of a null termination string in the vector register.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$1Ch_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Rb in Ra)

Exceptions: none

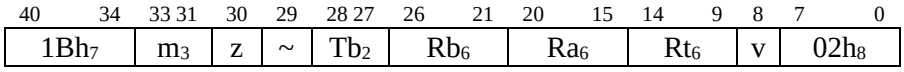
WYDENDX – WYDE Index

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).

If a vector WYDENDX 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 +255 in the scalar register. Thus, WYDENDX may be used to determine the length of a null termination string in the vector register.

Instruction Format: R2



Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Rb in Ra)

Exceptions: none

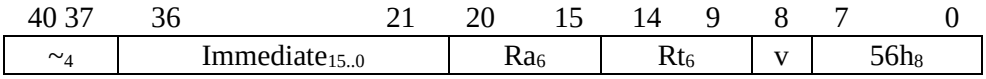
WYDENDXI – Wyde Index

Description:

This instruction searches Ra, which is treated as an array of four wydes, for a value specified by an immediate value and places the index of the value 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).

If a vector WYDENDX 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 +255 in the scalar register. Thus, WYDENDX may be used to determine the length of a null termination string in the vector register.

Instruction Format: RI



Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Imm_8 in Ra)

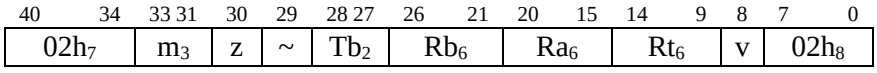
Exceptions: none

XNOR – Bitwise Exclusive Nor

Description:

Perform a bitwise ‘nor’ operation between operands. This is an alternate mnemonic for the ENOR instruction.

Integer Instruction Format: R2



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

Exceptions: none

XOR – Bitwise Exclusive Or

Description:

Perform a bitwise ‘or’ operation between operands.

Integer Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$0Ah_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Execution Units: All Integer ALU’s

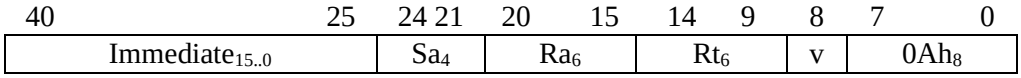
Exceptions: none

XORI – Bitwise Exclusive ‘Or’ Immediate

Description:

Perform a bitwise exclusive ‘or’ operation between operands. The immediate value is shifted to the left by a multiple of 16 bits before the addition takes place. The immediate is zero extended to the machine width and padded with zeros on the right. This instruction may be used to exclusive ‘or’ a large constant.

Instruction Format: RI



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

Operation

Rt = Ra ^ Immediate

Vector Operation

for x = 0 to VL-1

if (Vm0[x]) Vt[x] = Va[x] ^ Immediate

else Vt[x] = Vt[x]

Execution Units: All Integer ALU’s

Exceptions: none

Floating-Point Instructions

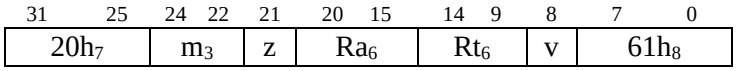
Overview

FABS – Absolute Value

Description:

This instruction takes the absolute value of a register and places the result in a target register. The values are treated as double precision floating-point values. The sign bit of the number is cleared, no rounding of the number takes place.

Integer Instruction Format: R1



v: 0 = scalar, 1 = vector op

Operation:

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

Vector Operation

```
for x = 0 to VL - 1
        if (Vm[x]) Rt[x] = Ra[x] < 0 ? -Ra[x] : Ra[x]
        else if (z) Rt[x] = 0
        else Rt[x] = Rt[x]
```

Execution Units: F

Clock Cycles: 1

Exceptions: none

Notes:

FADD – Add Register-Register

Description:

Add two registers and place the sum in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The mask register is ignored for scalar instructions. The values are treated as double precision floating-point values.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

Instruction Format: R2L

47 41	40 36	35 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	$\sim_5$	Rm_3	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$72h_8$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Rb

Exceptions:

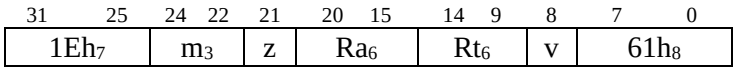
Notes:

FCLASS – Classify Value

Description:

FCLASS classifies the value in register Ra and returns the information as a bit vector in the integer register Rt.

Integer Instruction Format: R1



v: 0 = scalar, 1 = vector op

Bit in Rt	Meaning
0	1 = negative infinity
1	1 = negative number
2	1 = negative subnormal number
3	1 = negative zero
4	1 = positive zero
5	1 = positive subnormal number
6	1 = positive number
7	1 = positive infinity
8	1 = signalling nan
9	1 = quiet nan
10 to 62	not used
63	1 = negative, 0 = positive number

FCMP – Compare

Description

Compare two registers and return the relationship between them.

Integer Instruction Format: R2

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.

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$10h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

1 clock cycle

Operation:

Rt = Ra < Rb ? –1 : Ra = Rb ? 0 : 1

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] < Vb[x] ? –1 : Va[x]=Vb[x] ? 0 : 1
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

FCMPB – Compare

Description

Compare two registers and return the relationship between them.

Integer Instruction Format: R2

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.

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$15h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

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

Operation:

Vector Operation

FCX – Clear Floating-Point Exceptions

Description:

This instruction clears floating point exceptions. The Exceptions to clear are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$11h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$61h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	global invalid operation clears the following: - division of infinities - zero divided by zero - subtraction of infinities - infinity times zero - NaN comparison - division by zero
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	summary exception

FDIV – Divide Register-Register

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. The values are treated as double precision floating-point values.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$09h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

Instruction Format: R2L

47 41	40 36	35 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$09h_7$	$\sim_5$	Rm_3	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$72h_8$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra / Rb

Exceptions:

Notes:

FDX – Disable Floating Point Exceptions

Description:

This instruction disables floating point exceptions. The Exceptions disabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$13h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$61h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Disabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FEX – Enable Floating Point Exceptions

Description:

This instruction enables floating point exceptions. The Exceptions enabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$12h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$61h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FFINITE – Number is Finite

Description:

Test the value in Ra to see if it's a finite number and return Z=1 or Z = 0 in register Rt.

Integer Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$0Fh_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

v: 0 = scalar, 1 = vector op

Clock Cycles: 1

Execution Units: Floating Point

Example:

```
finite    $cr1,$f7
```

FMA – Floating Point Multiply Add

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$0h_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$63h_8$

Operation:

Rt = Ra * Rb + Rc

Clock Cycles: 30

Execution Units: All Floating Point

FNMA – Floating Point Negate Multiply Add

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$2h_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$63h_8$

Operation:

Rt = -Ra * Rb + Rc

Clock Cycles: 30

Execution Units: All Floating Point

FNMS – Floating Point Negate Multiply Subtract

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$3h_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$63h_8$

Operation:

Rt = -Ra * Rb - Rc

Clock Cycles: 30

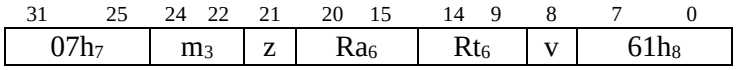
Execution Units: All Floating Point

FMAN – Mantissa of Number

Description:

This instruction provides the mantissa of a double precision floating point number contained in a general-purpose register as a 52-bit zero extended result. The hidden bit of the floating-point number remains hidden.

Integer Instruction Format: R1



v: 0

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

FMS – Floating Point Multiply Subtract

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$1h_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$63h_8$

Operation:

Rt = Ra * Rb - Rc

Clock Cycles: 30

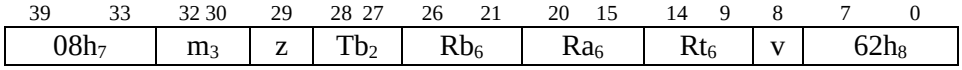
Execution Units: All Floating Point

FMUL – Floating point multiplication

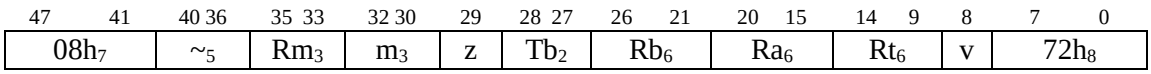
Description:

Multiply two double precision floating point numbers in registers Ra and Rb and place the result into target register Rt.

Instruction Format: R2



Instruction Format: R2L



Clock Cycles: 25

Execution Units: All Floating Point

FNEG – Negate Register

Description:

This instruction negates a double precision floating point number contained in a general-purpose register. The sign bit of the number is inverted. No rounding takes place.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$22h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

v: 0 = scalar, 1 = vector op

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = -Ra

FRM – Set Floating Point Rounding Mode

Description:

This instruction sets the rounding mode bits in the floating-point control register (FPSCR). The rounding mode bits are set to the bitwise ‘or’ of an immediate field in the instruction and the contents of register Ra. Either Ra or the immediate field should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$14h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$61h_8$

Execution Units: All Floating Point

Operation:

FPSCR.RM = Ra | Immediate

FRSQRTE – Float Reciprocal Square Root Estimate

Description:

Estimate the reciprocal of the square root of the number in register Ra and place the result into target register Rt.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$01h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

Clock Cycles: 5

Execution Units: Floating Point

Notes:

The estimate is only accurate to about 3%. The estimate is performed in single precision (32-bit) floating point, then converted to a 64-bit format. That means that input values must in the range of a 32-bit floating point number. Values outside of this range will return infinity or zero as a result.

Taking the reciprocal square root of a negative number results in a Nan output.

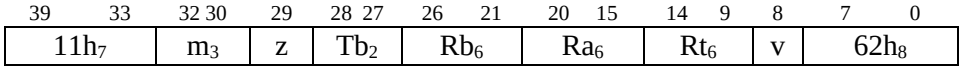
FSEQ - Float Set if Equal

Description:

The register compare instruction compares two registers as floating-point values for equality and sets the compare results register as a result. Note that negative and positive zero are considered equal.

Instruction Format: R2

Both values are treated as double precision (64-bit) floating point numbers. The result is returned as an integer value of 1 or 0.



1 clock cycle

Clock Cycles: 1

Execution Units: Floating Point

Operation:

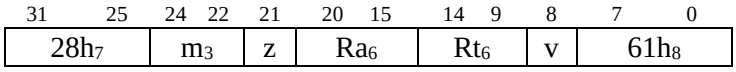
```
if Ra == Rb
        Rt= 1
else
        Rt= 0
```

FSIGMOID – Sigmoid Approximate

Description:

This function uses a 1024 entry 32-bit precision lookup table with linear interpolation to approximate the logistic sigmoid function in the range -8.0 to +8.0. Outside of this range 0.0 or +1.0 is returned. The sigmoid output is between 0.0 and +1.0. The value of the sigmoid for register Ra is returned in register Rt as a 64-bit double precision floating-point value.

Instruction Format: R1



Clock Cycles: 5

Execution Units: Floating Point

FSIGN – Sign of Number

Description:

This instruction provides the sign of a double precision floating point number contained in a general-purpose register as a floating-point double result. The result is +1.0 if the number is positive, 0.0 if the number is zero, and -1.0 if the number is negative.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$06h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = sign of (Ra)

FSLT - Float Set if Less Than

Description:

The register compare instruction compares two registers as floating-point values for less than and sets the target register as a result.

Instruction Format: R2

Both values are treated as double precision (64-bit) floating point numbers. The result is returned as an integer value of 1 or 0.

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$12h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

1 clock cycle

Clock Cycles: 1

Execution Units: Floating Point

Operation:

if Ra < Rb

Rt = 1

else

Rt = 0

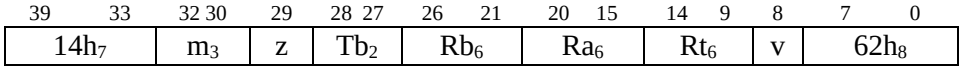
FSNE - Float Set if Not Equal

Description:

The register compare instruction compares two registers as floating-point values for inequality and sets the compare results register as a result. Note that negative and positive zero are considered equal.

Instruction Format: R2

Both values are treated as double precision (64-bit) floating point numbers. The result is returned as an integer value of 1 or 0.



1 clock cycle

Clock Cycles: 1

Execution Units: Floating Point

Operation:

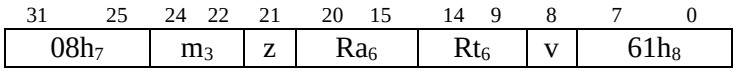
```
if Ra == Rb
        Rt= 1
else
        Rt= 0
```

FSQRT – Floating point square root

Description:

Take the square root of the floating-point number in register Ra and place the result into target register Rt. The sign bit (bit 63) of the register is set to zero. This instruction can generate NaNs.

Instruction Format: R1, R1L



Operation:

Rt = sqrt (Ra)

Clock Cycles: 64 (est).

Execution Units: Floating Point

FSTAT – Get Floating Point Status and Control

Description:

The floating-point status and control register may be read using the FSTAT instruction. The format of the FPSCR register is outlined on the next page.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$0Ch_7$	$\sim_3$	~	$\sim_6$	Rt_6	0	$61h_8$

Execution Units: All Floating Point

Operation:

Rt = FPSCR

Floating Point Status And Control Register Format:

Bit		Symbol	Description
31:29	**RM**	rm	rounding mode (unimplemented)
28	**E5**	inexe	- inexact exception enable
27	**E4**	dbzxe	- divide by zero exception enable
26	**E3**	underxe	- underflow exception enable
25	**E2**	overxe	- overflow exception enable
24	**E1**	invopxe	- invalid operation exception enable
23	**NS**	ns	- non standard floating point indicator
Result Status			
22		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
21	**RA**	rawayz	rounded away from zero (fraction incremented)
20	**SC**	C	denormalized, negative zero, or quiet NaN
19	**SL**	neg <	the result is negative (and not zero)
18	**SG**	pos >	the result is positive (and not zero)
17	**SE**	zero =	the result is zero (negative or positive)
16	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
15	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
14	**X5**	inerx	- inexact result exception occurred (sticky)
13	**X4**	dbzx	- divide by zero exception occurred
12	**X3**	underx	- underflow exception occurred
11	**X2**	overx	- overflow exception occurred
10	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
9	**GX**	gx	- global exception indicator – set if any enabled exception has happened
8	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

Greyed out items are not implemented.

FSUB – Subtract Register-Register

Description:

Subtract two registers and place the difference in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The mask register is ignored for scalar instructions. The values are treated as double precision floating-point values.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

Instruction Format: R2L

47 41	40 36	35 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	$\sim_5$	Rm_3	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$72h_8$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

FTOI – Float to Integer

Description:

This instruction converts a floating-point double value to an integer value.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$02h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

Clock Cycles: 2

Execution Units: All Floating Point

FTRUNC – Truncate Value

Description:

The FTRUNC instruction truncates off the fractional portion of the number leaving only a whole value. For instance, ftrunc(1.5) equals 1.0. Ftrunc does not change the representation of the number. To convert a value to an integer in a fixed-point representation see the FTOI instruction.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$15h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

Clock Cycles: 1

Execution Units: Floating Point

FTX – Trigger Floating Point Exceptions

Description:

This instruction triggers floating point exceptions. The Exceptions to trigger are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$10h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$61h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

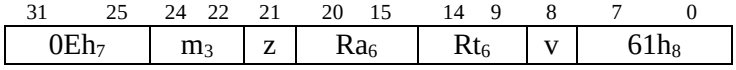
Bit	Exception Enabled
0	global invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

ISNAN – Is Not a Number

Description:

Test the value in Ra to see if it's a nan (not a number) and return true Z=1 or false Z=0 in register Rt.

Instruction Format: R1



Clock Cycles: 1

Execution Units: Floating Point

Example:

```
isnan   $cr1,$f7
```

ITOF – Integer to Float

Description:

This instruction converts an integer value to a double precision floating point representation.

Instruction Format: F1, F1L

31 25	24 22	21	20 15	14 9	8	7 0
$03h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

Clock Cycles: 2

Execution Units: All Floating Point

Decimal Floating-Point Instructions

DFABS – Absolute Value

Description:

This instruction takes the absolute value of a register and places the result in a target register. The values are treated as double precision decimal floating-point values. The sign bit of the number is cleared, no rounding of the number takes place.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$20h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

v: 0 = scalar, 1 = vector op

Operation:

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

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Rt[x] = Ra[x] < 0 ? -Ra[x] : Ra[x]
    else if (z) Rt[x] = 0
    else Rt[x] = Rt[x]
```

Execution Units: F

Clock Cycles: 1

Exceptions: none

Notes:

DFADD – Add Register-Register

Description:

Add two registers and place the sum in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The mask register is ignored for scalar instructions. The values are treated as double precision decimal floating-point values.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Instruction Format: R2L

47 41	40 36	35 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	$\sim_5$	Rm_3	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$76h_8$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

DFCMP – Compare

Description

Compare two registers and return the relationship between them.

Integer Instruction Format: R2

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.

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$10h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

1 clock cycle

Operation:

Rt = Ra < Rb ? –1 : Ra = Rb ? 0 : 1

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] < Vb[x] ? –1 : Va[x]=Vb[x] ? 0 : 1
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

DFCMPB – Compare

Description

Compare two registers and return the relationship between them.

Integer Instruction Format: R2

Both values are treated as double precision (64-bit) decimal 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.

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$15h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

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

Operation:

Vector Operation

DFCX – Clear Floating-Point Exceptions

Description:

This instruction clears floating point exceptions. The Exceptions to clear are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$11h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$65h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	global invalid operation clears the following: - division of infinities - zero divided by zero - subtraction of infinities - infinity times zero - NaN comparison - division by zero
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	summary exception

DFDIV – Divide Register-Register

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. The values are treated as double precision decimal floating-point values.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$09h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Instruction Format: R2L

47 41	40 36	35 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$09h_7$	$\sim_5$	Rm_3	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$76h_8$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra / Rb

Exceptions:

Notes:

DFDX – Disable Floating Point Exceptions

Description:

This instruction disables floating point exceptions. The Exceptions disabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$13h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$65h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Disabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

DFEX – Enable Floating Point Exceptions

Description:

This instruction enables floating point exceptions. The Exceptions enabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$12h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$65h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

DFMA – Floating Point Multiply Add

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$00h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$67h_8$

Instruction Format: R3L

55 49	48 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$00h_7$	$\sim_5$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$77h_8$

Operation:

Rt = Ra * Rb + Rc

Clock Cycles: 30

Execution Units: All Floating Point

DFNMA – Floating Point Negate Multiply Add

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$02h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$67h_8$

Instruction Format: R3L

55 49	48 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$02h_7$	$\sim_5$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$77h_8$

Operation:

Rt = -Ra * Rb + Rc

Clock Cycles: 30

Execution Units: All Floating Point

DFNMS – Floating Point Negate Multiply Subtract

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$02h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$67h_8$

Instruction Format: R3L

55 49	48 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$02h_7$	$\sim_5$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$77h_8$

Operation:

Rt = -Ra * Rb - Rc

Clock Cycles: 30

Execution Units: All Floating Point

DFMAN – Mantissa of Number

Description:

This instruction provides the mantissa of a double precision floating point number contained in a general-purpose register as a 52-bit zero extended result. The hidden bit of the floating-point number remains hidden.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$07h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

v: 0

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

DFMS – Floating Point Multiply Subtract

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$01h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$67h_8$

Instruction Format: R3L

55 49	48 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$01h_7$	$\sim_5$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$77h_8$

Operation:

Rt = Ra * Rb - Rc

Clock Cycles: 30

Execution Units: All Floating Point

DFMUL – Floating point multiplication

Description:

Multiply two double precision floating point numbers in registers Ra and Rb and place the result into target register Rt.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$08h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Instruction Format: R2L

47 41	40 36	35 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$08h_7$	$\sim_5$	Rm_3	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$76h_8$

Clock Cycles: 25

Execution Units: All Floating Point

DFNEG – Negate Register

Description:

This instruction negates a double precision floating point number contained in a general-purpose register. The sign bit of the number is inverted. No rounding takes place.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$22h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

v: 0 = scalar, 1 = vector op

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = -Ra

DFRM – Set Floating Point Rounding Mode

Description:

This instruction sets the rounding mode bits in the floating-point control register (FPSCR). The rounding mode bits are set to the bitwise ‘or’ of an immediate field in the instruction and the contents of register Ra. Either Ra or the immediate field should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$14h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$65h_8$

Execution Units: All Floating Point

Operation:

FPSCR.RM = Ra | Immediate

DFSIGN – Sign of Number

Description:

This instruction provides the sign of a double precision floating point number contained in a general-purpose register as a floating-point double result. The result is +1.0 if the number is positive, 0.0 if the number is zero, and -1.0 if the number is negative.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$06h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = sign of (Ra)

DFSTAT – Get Floating Point Status and Control

Description:

The floating-point status and control register may be read using the FSTAT instruction. The format of the FPSCR register is outlined on the next page.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$0Ch_7$	$\sim_3$	~	$\sim_6$	Rt_6	0	$65h_8$

Execution Units: All Floating Point

Operation:

Rt = FPSCR

Floating Point Status And Control Register Format:

Bit		Symbol	Description
31:29	**RM**	rm	rounding mode (unimplemented)
28	**E5**	inexe	- inexact exception enable
27	**E4**	dbzxe	- divide by zero exception enable
26	**E3**	underxe	- underflow exception enable
25	**E2**	overxe	- overflow exception enable
24	**E1**	invopxe	- invalid operation exception enable
23	**NS**	ns	- non standard floating point indicator
Result Status			
22		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
21	**RA**	rawayz	rounded away from zero (fraction incremented)
20	**SC**	C	denormalized, negative zero, or quiet NaN
19	**SL**	neg <	the result is negative (and not zero)
18	**SG**	pos >	the result is positive (and not zero)
17	**SE**	zero =	the result is zero (negative or positive)
16	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
15	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
14	**X5**	inerx	- inexact result exception occurred (sticky)
13	**X4**	dbzx	- divide by zero exception occurred
12	**X3**	underx	- underflow exception occurred
11	**X2**	overx	- overflow exception occurred
10	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
9	**GX**	gx	- global exception indicator – set if any enabled exception has happened
8	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

Greyed out items are not implemented.

DFSUB – Subtract Register-Register

Description:

Subtract two registers and place the difference in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The mask register is ignored for scalar instructions. The values are treated as double precision floating-point values.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Instruction Format: R2L

47 41	40 36	35 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	$\sim_5$	Rm_3	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$76h_8$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

DFTOI – Float to Integer

Description:

This instruction converts a floating-point double value to an integer value.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$02h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

Clock Cycles: 2

Execution Units: All Floating Point

DFTX – Trigger Floating Point Exceptions

Description:

This instruction triggers floating point exceptions. The Exceptions to trigger are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$10h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$65h_8$

Execution Units: All Floating Point

Operation:

Exceptions:

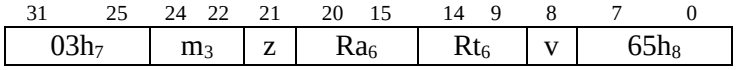
Bit	Exception Enabled
0	global invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

ITODF – Integer to Float

Description:

This instruction converts an integer value to a double precision decimal floating point representation.

Instruction Format: F1, F1L



Clock Cycles: 2

Execution Units: All Floating Point

Load / Store Instructions

Overview

Addressing Modes

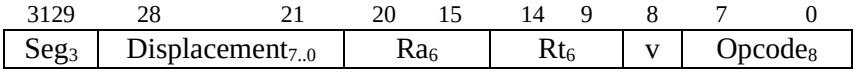
Load and store instructions have two addressing modes, register indirect with displacement and scaled indexed addressing. There are two formats for register indirect with displacement addressing, differing in the number of bits used to represent a displacement. Store operations have two fewer bits reserved for encoding displacements in instructions.

Load and store instructions specify a segment register to use with the instruction. The assembler will provide a default value for this field that is suitable in most cases. The default can be overridden using a segment prefix indicator.

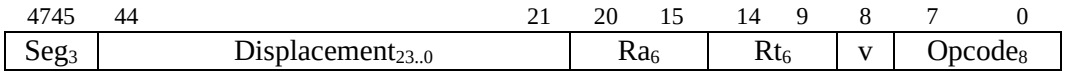
Load Formats

Register Indirect with Displacement Format

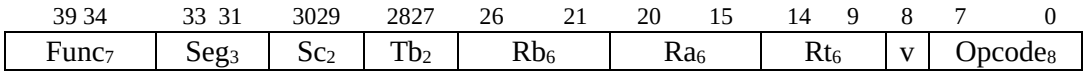
For register indirect with displacement addressing the load or store address is the sum of a register Ra and a displacement constant found in the instruction.



Register Indirect with Long Displacement Format



Scaled Indexed Format



Store Formats

Stores have two fewer displacement bits than loads. The scaled indexed format is wider than that of the load.

Register Indirect with Displacement Format

For register indirect with displacement addressing the load or store address is the sum of a register Ra and a displacement constant found in the instruction.

3129	2827	26 21	20 15	14 9	8	7 0
Sg_3	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$Opcode_8$

Register Indirect with Long Displacement Format

47 45	44 29	2827	26 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{21..6}$	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$Opcode_8$

Scaled Indexed Format

47 42	41 39	3837	3635	34 29	2827	26 21	20 15	14 9	8	7 0
$00h_6$	Seg_3	Sc_2	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	$\sim_6$	v	$C0h_8$

CACHE – Cache Command

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 Format:

3129	28 21	20 15	1412	119	8	7 0
Sg_3	$Displacement_{7..0}$	Ra_6	DC_3	IC_3	v	$9Fh_8$

Commands:

IC_3	Mne.	Operation
0	NOP	no operation
3	invline	invalidate line associated with given address
4	invall	invalidate the entire cache (address is ignored)
5 to 7		reserved

DC_3	Mne.	Operation
0	NOP	no operation
1	enable	enable cache (instruction cache is always enabled)
2	disable	not valid for the instruction cache
3	invline	invalidate line associated with given address
4	invall	invalidate the entire cache (address is ignored)
5 to 7		reserved

Clock Cycles: 3

Execution Units: All ALU's / Memory

Operation:

Exceptions: DBG

CACHEL – 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

47 45	44 21	20 15	14 12	11 9	8	7 0
Sg_3	$Displacement_{23..0}$	Ra_6	DC_3	IC_3	v	DFh_8

Commands:

IC_3	Mne.	Operation
0	NOP	no operation
3	invline	invalidate line associated with given address
4	invall	invalidate the entire cache (address is ignored)
5 to 7		reserved

DC_3	Mne.	Operation
0	NOP	no operation
1	enable	enable cache (instruction cache is always enabled)
2	disable	not valid for the instruction cache
3	invline	invalidate line associated with given address
4	invall	invalidate the entire cache (address is ignored)
5 to 7		reserved

Notes:

CACHEX – Cache Command

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.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 12	11 9	8	7 0
$0Ah_7$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	DC_3	IC_3	v	$B0h_8$

Commands:

IC_3	Mne.	Operation
0	NOP	no operation
3	invline	invalidate line associated with given address
4	invall	invalidate the entire cache (address is ignored)
5 to 7		reserved

DC_3	Mne.	Operation
0	NOP	no operation
1	enable	enable cache (instruction cache is always enabled)
2	disable	not valid for the instruction cache
3	invline	invalidate line associated with given address
4	invall	invalidate the entire cache (address is ignored)
5 to 7		reserved

Clock Cycles:

Execution Units: All ALU's / Memory

Operation:

Exceptions: DBG

LDB – Load Byte

Description:

An eight-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. For vector instructions Ra is a scalar register.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

40 37	3634	33 21	20 15	14 9	8	7 0
$Disp_4$	Sg_3	$Displacement_{12..0}$	Ra_6	Rt_6	0	$80h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_8$[Ra+displacement])

Vector Operation:

```
y = 0
for n = 0 to VL – 1
        if (Vm[n])
                Vb[n] = memory8[Ra+displacement + y * Rb]
        else if (z & ~C)
                Vb[n] = 0
        else
                Vb[n] = Vb[n]
        if (C) y = y + Vm[x]
        else y = y + 1
```

Exceptions: DBE, DBG, LMT, TLB

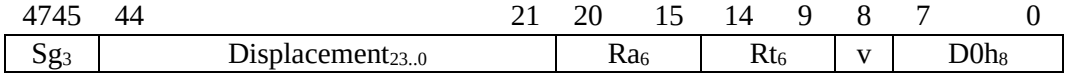
LDBL – Load Byte, Long Address

Description:

An eight-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. For vector instructions Ra is a scalar register.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:



Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_8$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDBU – Load Byte, Unsigned

Description:

An eight-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. For vector instructions Ra is a scalar register.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

3129	28 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{7..0}$	Ra_6	Rt_6	v	$81h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend ($memory_8$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDBUL – Load Byte Unsigned, Long Address

Description:

An eight-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

47	45	44	21	20	15	14	9	8	7	0
Sg_3		$Displacement_{23..0}$		Ra_6		Rt_6		v	$D1h_8$	

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend ($memory_8$[Ra+offset])

Exceptions: DBE, DBG, LMT, TLB

LDBUX – Load Byte Unsigned Indexed

Description:

An eight-bit value is loaded from memory zero extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

4039	38 36	35	34 32	31 29	2827	26 21	20 15	14 9	8	7 0
~$_2$	m$_3$	z	Seg$_3$	Sc$_3$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	B1h$_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend (memory$_8$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDBX – Load Byte Indexed

Description:

An eight-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb. For vector instructions Ra is a scalar register.

Instruction Format:

40	39	38 36	35	34 32	31 29	2827	26 21	20 15	14 9	8	7 0
~	C	m_3	z	Seg_3	Sc_3	Tb_2	Rb_6	Ra_6	Rt_6	v	$B0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_8$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDO – Load Octa

Description:

A sixty-four-bit value is loaded from memory then placed in the target register. The memory address is the sum of the sign extended displacement and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

3129	28 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{7..0}$	Ra_6	Rt_6	v	$86h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{64}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

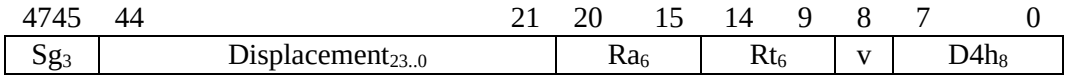
LDOL – Load Octa, Long Address

Description:

A sixty-four-bit value is loaded from memory then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:



Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{64}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDOX – Load Octa Indexed

Description:

A sixty-four-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$06h_7$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rt_6	v	$B0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{64}$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDT – Load Tetra

Description:

A thirty-two-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

3129	28 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{7..0}$	Ra_6	Rt_6	v	$84h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{32}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDTL – Load Tetra, Long Address

Description:

A thirty-two-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

47 45	44 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{23..0}$	Ra_6	Rt_6	v	$D4h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{32}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDTU – Load Tetra Unsigned

Description:

A thirty-two-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

3129	28 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{7..0}$	Ra_6	Rt_6	v	$84h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend ($memory_{32}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDTUL – Load Tetra Unsigned, Long Address

Description:

A thirty-two-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

47 45	44 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{23..0}$	Ra_6	Rt_6	v	$D5h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend ($memory_{32}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDTUX – Load Tetra Unsigned Indexed

Description:

A thirty-two-bit value is loaded from memory zero extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$05h_7$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rt_6	v	$B0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend ($memory_{32}$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDTX – Load Tetra Indexed

Description:

A thirty-two-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$04h_7$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rt_6	v	$B0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{32}$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDW – Load Wyde

Description:

A sixteen-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

3129	28 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{7..0}$	Ra_6	Rt_6	v	$82h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{16}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDWL – Load Wyde, Long Address

Description:

A sixteen-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

4745	44	21	20 15	14 9	8	7 0
Sg_3	$Displacement_{23..0}$		Ra_6	Rt_6	v	$D2h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{16}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDWU – Load Wyde Unsigned

Description:

A sixteen-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

3129	28 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{7..0}$	Ra_6	Rt_6	v	$83h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend ($memory_{16}$[Ra+offset])

Exceptions: DBE, DBG, LMT, TLB

LDWUL – Load Wyde Unsigned, Long Address

Description:

A sixteen-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

47..45	44..21	20..15	14..9	8	7..0
Sg_3	$Displacement_{23..0}$	Ra_6	Rt_6	v	$D3h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend ($memory_{16}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDWX – Load Wyde Indexed

Description:

A sixteen-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$02h_7$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rt_6	v	$B0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{16}$[Ra+Rb])

Exceptions: DBE, DBG, LMT, TLB

LDWUX – Load Wyde Unsigned Indexed

Description:

A sixteen-bit value is loaded from memory zero extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$03h_7$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rt_6	v	$B0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend ($memory_{16}$[Ra+Rb])

Exceptions: DBE, DBG, LMT, TLB

LIN2PHL – Convert Linear to Physical Address Low

Description:

The linear address in register pair Rb, Ra is converted to the low order 64-bits of the physical address.

Instruction Format:

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$36h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 3

Execution Units: Memory

Operation:

Rt = Lin2PhysL({Rb,Ra})

Exceptions: none

LIN2PHH – Convert Linear to Physical Address High

Description:

The linear address in register pair Rb, Ra is converted to the high order 64-bits of the physical address.

Instruction Format:

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$37h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 3

Execution Units: Memory

Operation:

Rt = Lin2PhysH({Rb,Ra})

Exceptions: none

LLAH – Load Linear Address High

Description:

The high order 64-bits of the linear memory address is calculated and stored in the target register.

Instruction Format:

3129	28 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{7..0}$	Ra_6	Rt_6	v	$8Ah_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = Ra+displacement

Exceptions: DBE, DBG, LMT, TLB

LLAHL – Load Linear Address High Long

Description:

The high order 64-bits of the linear memory address is calculated and stored in the target register.

Instruction Format:

4745	44 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{23..0}$	Ra_6	Rt_6	v	DAh_8

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = Ra+displacement

Exceptions: DBE, DBG, LMT, TLB

LLAHX – Load Linear Address High Indexed

Description:

The high order 64-bits of the linear memory address is calculated and stored in the target register.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$09h_7$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rt_6	v	$B0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = Ra+Rb*Sc

Exceptions: DBE, DBG, LMT, TLB

LLAL – Load Linear Address Low

Description:

The low order 64-bits of the linear memory address is calculated and stored in the target register.

Instruction Format:

3129	28 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{7..0}$	Ra_6	Rt_6	v	$89h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = Ra+displacement

Exceptions: DBE, DBG, LMT, TLB

LLALL – Load Linear Address Low Long

Description:

The low order 64-bits of the linear memory address is calculated and stored in the target register.

Instruction Format:

4745	44 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{23..0}$	Ra_6	Rt_6	v	$D9h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = Ra+displacement

Exceptions: DBE, DBG, LMT, TLB

LLALX – Load Linear Address Low Indexed

Description:

The low order 64-bits of the linear memory address is calculated and stored in the target register.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$08h_7$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rt_6	v	$B0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = Ra+Rb*Sc

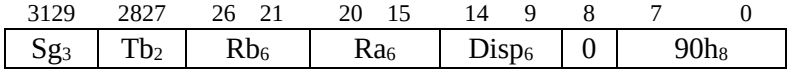
Exceptions: DBE, DBG, LMT, TLB

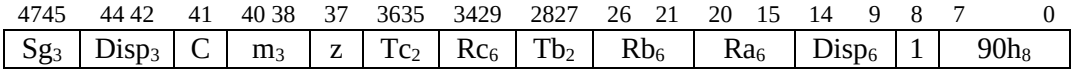
STB – Store Byte

Description:

An eight-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:





Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_8[Ra+displacement] = Rb_{[7..0]}$

Vector Operation:

```
y = 0
for n = 0 to VL – 1
    if (Vm[n])
        memory8[Ra+displacement + y * Rc] = Vb[n][7..0]
    else if (z & ~C)
        memory8[Ra+displacement + y * Rc] = 0[7..0]
    if (C) y = y + Vm[x]
    else y = y + 1
```

Exceptions: DBE, DBG, TLB, LMT

STBL – Store Byte, Long Addressing

Description:

An eight-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 45	44 29	2827	26 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{21..6}$	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$E0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_8[Ra+offset] = Rb_{[7..0]}$

Exceptions: DBE, DBG, TLB, LMT

STBX – Store Byte Indexed

Description:

An eight-bit value is stored to memory from register Rs. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

4039	38 36	35	34 32	31 29	2827	26 21	20 15	14 9	8	7 0
~$_2$	m$_3$	z	Seg$_3$	Sc$_3$	Tb$_2$	Rb$_6$	Ra$_6$	Rs$_6$	v	C0h$_8$

Clock Cycles: 10 (one memory access)

Execution Units: All AGU's / Memory

Operation:

memory$_8$[Ra+Rb*Sc] = Rs$_{[7:0]}$

Exceptions: DBE, DBG, LMT, TLB

STO – Store Octa

Description:

A sixty-four-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

3129	2827	26 21	20 15	14 9	8	7 0
Sg_3	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$93h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{64}$[Ra+displacement] = Rb

Exceptions: DBE, DBG, TLB, LMT

STOC – Store Octa, Clear Reservation

Description:

A sixty-four-bit value from the source register Rb is stored to memory if a reservation exists at the target address. The reservation at the target address is cleared. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

31..29	28..27	26..21	20..15	14..9	8	7..0
Sg_3	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$94h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{64}$[Ra+displacement] = Rb

Exceptions: DBE, DBG, TLB, LMT

STOCL – Store Octa, Clear Reservation, Long Addressing

Description:

A sixty-four-bit value from the source register Rb is stored to memory if a reservation exists at the target address. The reservation at the target address is cleared. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47..45	44..29	28..27	26..21	20..15	14..9	8	7..0
Sg_3	$Displacement_{21..6}$	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$E4h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{64}$[Ra+offset] = Rb

Exceptions: DBE, DBG, TLB, LMT

STOCX – Store Octa, Clear Reservation Indexed

Description:

A sixty-four-bit value from the source register Rs is stored to memory if a reservation exists at the target address. The reservation at the target address is cleared. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

39 34	33 31	30 29	28 27	26 21	20 15	14 9	8	7 0
$04h_6$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rs_6	v	$C0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{64}$[Ra+Rc*Sc] = Rb

Exceptions: DBE, DBG, LMT, TLB

STOL – Store Octa, Long Addressing

Description:

A sixty-four-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 45	44 29	2827	26 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{21..6}$	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$E3h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{64}$[Ra+offset] = Rb

Exceptions: DBE, DBG, TLB, LMT

STOX – Store Octa Indexed

Description:

A sixty-four-bit value is stored to memory from register Rs. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$03h_6$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rs_6	v	$C0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{64}$[Ra+Rb*Sc] = Rs

Exceptions: DBE, DBG, LMT, TLB

STT – Store Tetra

Description:

A thirty-two-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

3129	2827	26 21	20 15	14 9	8	7 0
Sg_3	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$92h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{32}[Ra+displacement] = Rb_{[31..0]}$

Exceptions: DBE, DBG, TLB, LMT

STTL – Store Tetra, Long Addressing

Description:

A thiry-two-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 45	44 29	2827	26 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{21..6}$	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$E2h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{32}[Ra+offset] = Rb_{[31..0]}$

Exceptions: DBE, DBG, TLB, LMT

STTX – Store Tetra Indexed

Description:

A thirty-two-bit value is stored to memory from register Rs. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$02h_6$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rs_6	v	$C0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{32}[Ra+Rb*Sc] = Rs_{[31:0]}$

Exceptions: DBE, DBG, LMT, TLB

STW – Store Wyde

Description:

A sixteen-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

31..29	28..27	26..21	20..15	14..9	8	7..0
Sg_3	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$91h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{16}[Ra+displacement] = Rb_{[15..0]}$

Exceptions: DBE, DBG, TLB, LMT

STWL – Store Wyde, Long Addressing

Description:

A sixteen-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 45	44 29	2827	26 21	20 15	14 9	8	7 0
Sg_3	$Displacement_{21..6}$	Tb_2	Rb_6	Ra_6	$Disp_6$	v	$E1h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{16}[Ra+offset] = Rb_{[15..0]}$

Exceptions: DBE, DBG, TLB, LMT

STWX – Store Wyde Indexed

Description:

A sixteen-bit value is stored to memory from register Rs. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

39 34	33 31	3029	2827	26 21	20 15	14 9	8	7 0
$01h_6$	Seg_3	Sc_2	Tb_2	Rb_6	Ra_6	Rs_6	v	$C0h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{16}[Ra+Rb*Sc] = Rs_{[15:0]}$

Exceptions: DBE, DBG, LMT, TLB

Branch / Flow Control Instructions

Overview

Mnemonics

There are two sets of mnemonics for branch instructions. Branch instructions that are IP relative, specify Ca = 7 in the branch instruction and are referred to with a 'B' as in BEQ. Using mnemonics that begin with 'B' imply the code address register is the instruction pointer, C7. Branch instructions that are relative to other code address registers are referred to as jump instructions and begin with a 'J' in the mnemonic.

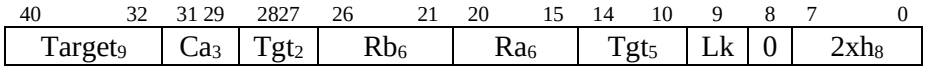
Instructions that decrement the loop counter begin with a 'D' in the mnemonic as in DBEQ standing for decrement and branch if equal.

Conditions

Conditional branches branch to the target address only if the condition is true. The condition is determined by the comparison of two general-purpose registers.

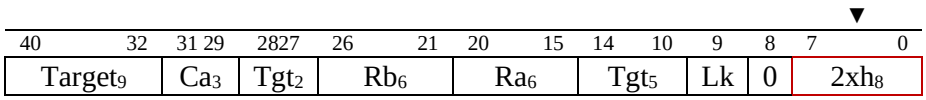
> *The original Thor machine used instruction predicates to implement conditional branching. Another instruction was required to set the predicate before branching. Combining compare and branch in a single instruction may reduce the dynamic instruction count. An issue with comparing and branching in a single instruction is that it may lead to a wider instruction format.*

Conditional Branch Format



Branch Conditions

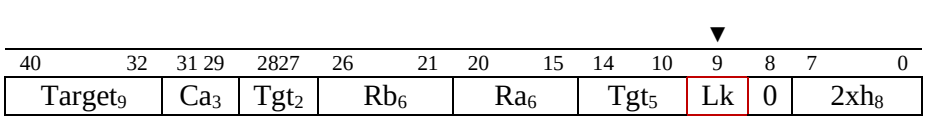
The branch opcode determines the condition under which the branch will execute.



2x	Comparison Test
28h	signed less than
29h	signed greater or equal
2Ah	signed less than or equal
2Bh	signed greater than
2Ch	unsigned less than
2Dh	unsigned greater or equal
2Eh	unsigned less than or equal
2Fh	unsigned greater than
26h	equal
27h	not equal
24h	bit clear
25h	bit set
38h	decrement, LC != 0 and signed less than
39h	decrement, LC != 0 and signed greater or equal
3Ah	decrement, LC != 0 and signed less than or equal
3Bh	decrement, LC != 0 and signed greater than
3Ch	decrement, LC != 0 and unsigned less than
3Dh	decrement, LC != 0 and unsigned greater or equal
3Eh	decrement, LC != 0 and unsigned less than or equal
3Fh	decrement, LC != 0 and unsigned greater than
36h	decrement, LC != 0 and equal
37h	decrement, LC != 0 and not equal
34h	decrement, LC != 0 and bit clear
35h	decrement, LC != 0 and bit set

Linkage

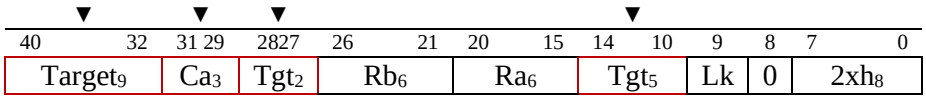
Branches may specify a linkage register which is updated with the address of the next instruction. This allows subroutines to be called. There are two link registers in the architecture.



Lk	Meaning
0	do not store return address
1	use Lk1

Branch Target

For conditional branches, the target address is formed as the sum of a code address register and a 16-bit constant specified in the instruction. Branches may be IP relative with a range of ±128kB. To perform a far branch operation, specify a code address register containing the target selector value as part of the target address.

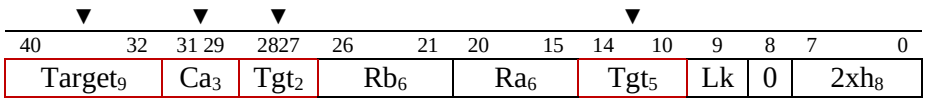


Near or Far Branching

In a segmented system a question that comes to mind is how to perform a far jump or call versus performing a near one. For Thor2021 all branches are effectively far branches as the target selector is always loaded into the IP register. The return address selector is always stored in a link register. However, if addresses are formed using IP relative addressing, $Ca_3 = 7$, then the selector value will remain unchanged making the branch effectively a near branch. If the Ca_3 field is specified as a zero, then the IP selector value will remain unchanged. A non-zero Ca_3 register is needed to perform a far branch.

Branch to Register

The branch to register instruction allows a conditional return from subroutine to be used or a branch to a value in a register. Branching to a value in a register allows all bits of the instruction pointer to be set. Since addresses are formed as the sum of a code address register and a constant in the instruction, branching to a register is inherent in the instruction. The target constant may be set to zero.

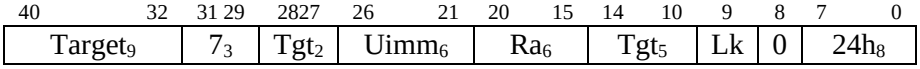


BBC – Branch if Bit Clear

Description:

This instruction branches to the target address if the $uimm_6$ bit of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B



Operation:

Lk = next IP
If (Ra.bit[$uimm_6$] == 0)
IP = IP + Constant

Execution Units: Branch

Exceptions: none

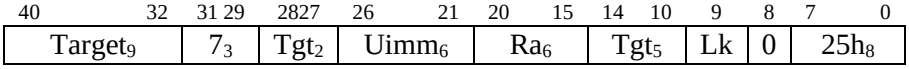
Notes:

BBS – Branch if Bit Set

Description:

This instruction branches to the target address if a bit of Ra is set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B



40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	7_3	Tgt_2	$Uimm_6$	Ra_6	Tgt_5	Lk	0	$25h_8$

Operation:

```
Lk = next IP
If (Ra.bit[Rb] == 1)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

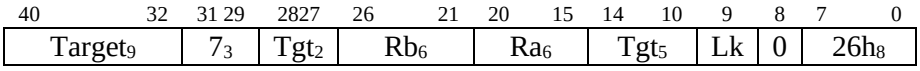
Notes:

BEQ – Branch if Equal

Description:

This instruction branches to the target address if the contents of the Ra equals the contents of Rb, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B



Operation:

Lk = next IP
If (Ra==Rb)
IP = IP + Constant

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 the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: B

40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	7_3	Tgt_2	Rb_6	Ra_6	Tgt_5	Lk	0	$29h_8$

Operation:

Lk = next IP
If (Ra>=Rb)
    IP = IP + Constant

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 the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: B

40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	7_3	Tgt_2	Rb_6	Ra_6	Tgt_5	Lk	0	$2Dh_8$

Operation:

Lk = next IP
If (Ra>=Rb)
    IP = IP + Constant

Execution Units: Branch

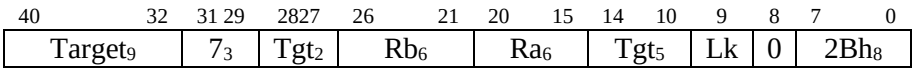
Exceptions: none

BGT – Branch if Greater Than

Description:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: B



Operation:

Lk = next IP
If (Ra > Rb)
IP = IP + Constant

Execution Units: Branch

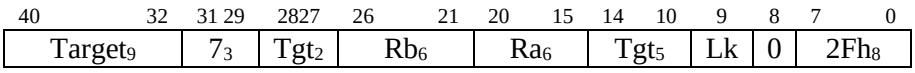
Exceptions: none

BGTU – Branch if Greater Than Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: B



Operation:

Lk = next IP
If (Ra > Rb)
IP = IP + Constant

Execution Units: Branch

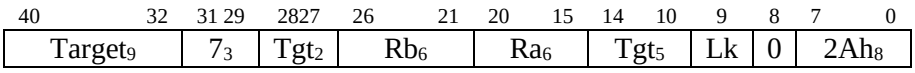
Exceptions: none

BLE – Branch if Less Than or Equal

Description:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: B



Operation:

Lk = next IP
If (Ra <= Rb)
IP = IP + Constant

Execution Units: Branch

Exceptions: none

BLEU – Branch if Less Than or Equal Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: B

40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	7_3	Tgt_2	Rb_6	Ra_6	Tgt_5	Lk	0	$2Eh_8$

Operation:

Lk = next IP
If (Ra <= Rb)
    IP = IP + Constant

Execution Units: Branch

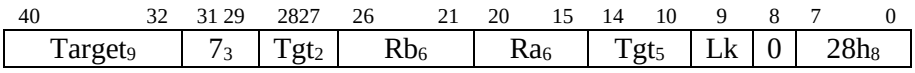
Exceptions: none

BLT – Branch if Less Than

Description:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: B



Operation:

Lk = next IP
If (Ra > Rb)
IP = IP + Constant

Execution Units: Branch

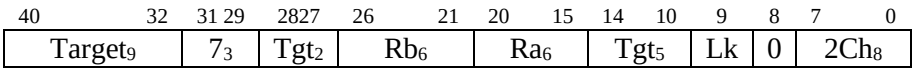
Exceptions: none

BLTU – Branch if Less Than Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: B



Operation:

Lk = next IP
If (Ra > Rb)
IP = IP + Constant

Execution Units: Branch

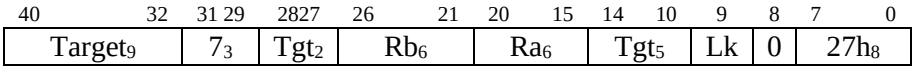
Exceptions: none

BNE – Branch if Not Equal

Description:

This instruction branches to the target address if the contents of the Ra is not equal to the contents of Rb, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B



Operation:

Lk = next IP
If (Ra != Rb)
IP = IP + Constant

Execution Units: Branch

Exceptions: none

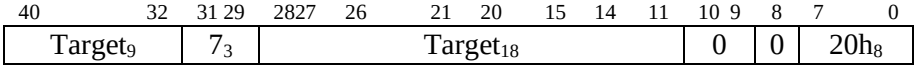
Notes:

BRA – Branch Always

Description:

This instruction always branches to the target address.

Formats Supported: B



Operation:

IP = IP + Constant

Execution Units: Branch

Exceptions: none

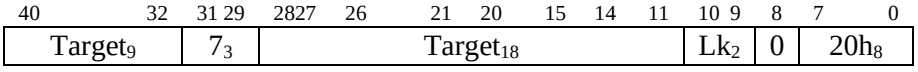
Notes:

BSR – Branch to Subroutine

Description:

This instruction always jumps to the target address. The address of the next instruction is stored in a link register.

Formats Supported: B



Operation:

Lk = next IP
IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

DBBC – Decrement and Branch if Bit Clear

Description:

This instruction branches to the target address if the $uimm_6$ bit of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B

40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	7_3	Tgt_2	$Uimm_6$	Ra_6	Tgt_5	Lk	0	$34h_8$

Operation:

```
Lk = next IP
If (Ra.bit[uimm6] == 0 and LC != 0)
        IP = IP + Constant
LC = LC - 1
```

Execution Units: Branch

Exceptions: none

Notes:

DBBS – Decrement and Branch if Bit Set

Description:

This instruction branches to the target address if a bit of Ra is set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B

40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	7_3	Tgt_2	$Uimm_6$	Ra_6	Tgt_5	Lk	0	$35h_8$

Operation:

```
Lk = next IP
If (Ra.bit[Rb] == 1 and LC != 0)
        IP = IP + Constant
LC = LC - 1
```

Execution Units: Branch

Exceptions: none

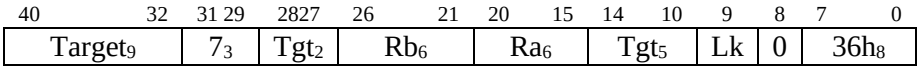
Notes:

DBEQ – Decrement and Branch if Equal

Description:

This instruction branches to the target address if the contents of the Ra equals the contents of Rb, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B



Operation:

Lk = next IP
If (Ra==Rb and LC != 0)
IP = IP + Constant
LC = LC – 1

Execution Units: Branch

Exceptions: none

Notes:

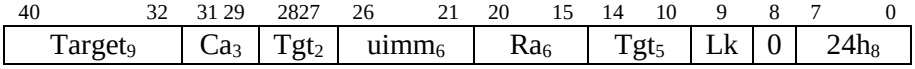
For a floating-point comparison positive and negative zero are considered equal.

JBC – Jump if Bit Clear

Description:

This instruction branches to the target address if a bit of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: J



Operation:

```
Lk = next IP
If (Ra.bit[Rb] == 0)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

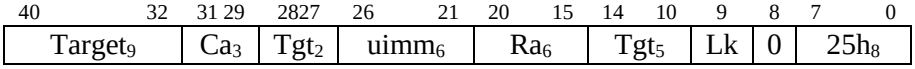
Notes:

JBS – Jump if Bit Set

Description:

This instruction branches to the target address if a bit of Ra is set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: J



Operation:

```
Lk = next IP
If (Ra.bit[Rb] == 1)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

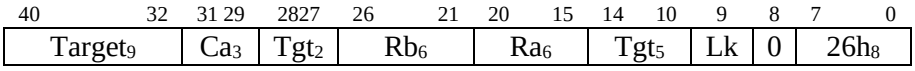
Notes:

JEQ – Jump if Equal

Description:

This instruction branches to the target address if the contents of the Ra equals the contents of Rb, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: J



Operation:

Lk = next IP
If (Ra==Rb)
    IP = Ca + Constant

Execution Units: Branch

Exceptions: none

Notes:

JGE – Jump if Greater Than or Equal

Description:

This instruction branches to the target address if the contents of the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: J

40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	Ca_3	Tgt_2	Rb_6	Ra_6	Tgt_5	Lk	0	$29h_8$

Operation:

Lk = next IP
If (Ra>=Rb)
    IP = Ca + Constant

Execution Units: Branch

Exceptions: none

JGEU – Jump if Greater Than or Equal Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: J

40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	Ca_3	Tgt_2	Rb_6	Ra_6	Tgt_5	Lk	0	$2Dh_8$

Operation:

Lk = next IP
If (Ra >= Rb)
    IP = Ca + Constant

Execution Units: Branch

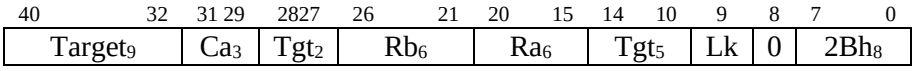
Exceptions: none

JGT – Jump if Greater Than

Description:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: J



Operation:

Lk = next IP
If (Ra > Rb)
IP = Ca + Constant

Execution Units: Branch

Exceptions: none

JGTU – Jump if Greater Than Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: J

40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	Ca_3	Tgt_2	Rb_6	Ra_6	Tgt_5	Lk	0	$2Fh_8$

Operation:

Lk = next IP
If (Ra > Rb)
    IP = Ca + Constant

Execution Units: Branch

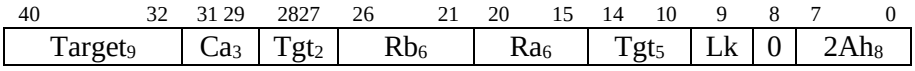
Exceptions: none

JLE – Jump if Less Than or Equal

Description:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: J



Operation:

Lk = next IP
If (Ra <= Rb)
IP = Ca + Constant

Execution Units: Branch

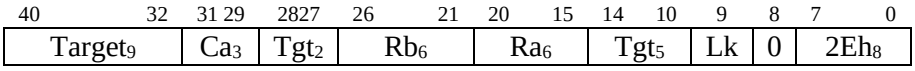
Exceptions: none

JLEU – Jump if Less Than or Equal Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: J



Operation:

Lk = next IP
If (Ra <= Rb)
IP = Ca + Constant

Execution Units: Branch

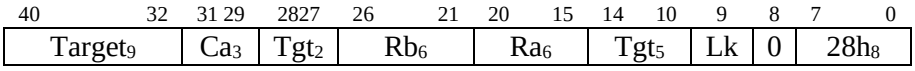
Exceptions: none

JLT – Jump if Less Than

Description:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: J



Operation:

Lk = next IP
If (Ra > Rb)
IP = Ca + Constant

Execution Units: Branch

Exceptions: none

JLTU – Jump if Less Than Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: J

40 32	31 29	2827	26 21	20 15	14 10	9	8	7 0
$Target_9$	Ca_3	Tgt_2	Rb_6	Ra_6	Tgt_5	Lk	0	$2Ch_8$

Operation:

Lk = next IP
If (Ra > Rb)
    IP = Ca + Constant

Execution Units: Branch

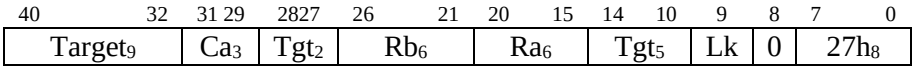
Exceptions: none

JNE – Jump if Not Equal

Description:

This instruction branches to the target address if the contents of the Ra is not equal to the contents of Rb, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: J



Operation:

Lk = next IP
If (Ra != Rb)
IP = Ca + Constant

Execution Units: Branch

Exceptions: none

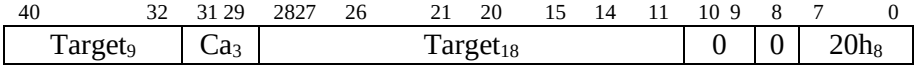
Notes:

JMP – Jump

Description:

This instruction always jumps to the target address.

Instruction Format: JMP



Operation:

Lk = next IP
IP = Ca + sign extend (Constant$_{28}$)

Execution Units: Branch

Exceptions: none

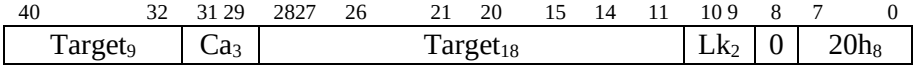
Notes:

JSR – Jump to Subroutine

Description:

This instruction always jumps to the target address. The address of the next instruction is stored in a link register. The target address range is ±256MB.

Formats Supported: JMP



Operation:

Lk = next IP
IP = Ca + Constant

Execution Units: Branch

Exceptions: none

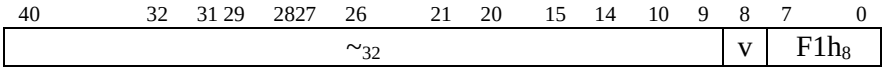
Notes:

NOP – No Operation

Description:

This instruction does not do anything.

Integer Instruction Format:



Operation: none

Vector Operation

Execution Units: I

Clock Cycles: 1

Exceptions: none

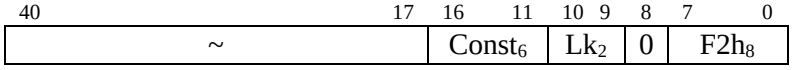
Notes:

RTS – Return from Subroutine

Description:

This instruction returns from a subroutine by transferring program execution to the address calculated as the sum of a link register and a constant.

Formats Supported: RTS



Flags Affected: none

Operation:

Execution Units: Branch

Exceptions: none

Notes:

A branch instruction may also be used to return from a subroutine

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

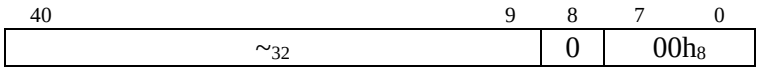
System Instructions

BRK – Break

Description:

This instruction initiates the processor debug routine. The processor enters debug mode. The cause code register is set to indicate execution of a BRK instruction. Interrupts are disabled. The instruction pointer is reset to the contents of tvec[3] 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, C6.

Instruction Format: BRK



Operation:

PMSTACK = (PMSTACK << 4) | 6
CAUSE = FLT_BRK
C[6] = IP
IP = tvec[3]

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

40	39 24	2321	20 15	14 9	8	7 0
~	$Regno_{16}$	O_3	Ra_6	Rt_6	0	$0Fh_8$

O_3		Operation
0	CSRRD	Only read the CSR, no update takes place, Ra should be x0.
1	CSRRW	Read/Write to CSR
2	CSRRS	Read/Set CSR bits
3	CSRRC	Read/Clear CSR bits
4 to 7		Reserved

CSRRS and CSRRC operations are only valid on registers that support the capability.

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

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

Clock Cycles: 1

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

DI – Disable Interrupts

Description:

This instruction disables interrupts for a short period of time. The Rb field specifies the number of following instructions for which interrupts are disabled. Interrupts may be disabled for the execution of a maximum of seven instructions.

Instruction Format: OSR2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$16h_7$	m_3	z	~	Tb_2	Rb_6	$\sim_6$	$\sim_6$	v	$07h_8$

Operation:

Execution Units: ALU

Clock Cycles:

Exceptions: none

Notes:

INT – Generate Interrupt

Description:

Generate interrupt. This instruction invokes the system exception handler. The return address is stored in the EIP register (code address register #6).

The return address stored is the address of the interrupt instruction, not the address of the next instruction. To call system routines use the SYS instruction.

The level of the interrupt is checked and if the interrupt level in the instruction is less than or equal to the current interrupt level then the instruction will be ignored.

Instruction Format:

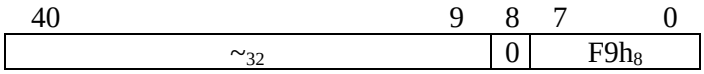
40 20	19 17	16 9	8	7 0
$\sim_{21}$	Lvl_3	$Cause_8$	0	$A6h_8$

MEMDB – Memory Data Barrier

Description:

All memory accesses before the MEMDB command are completed before any memory accesses after the data barrier are started.

Instruction Format:



Clock Cycles: 1

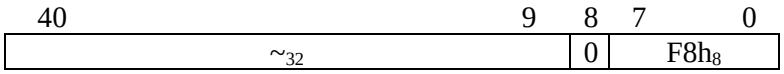
Execution Units: Memory

MEMSB – Memory Synchronization Barrier

Description:

All instructions before the MEMSB command are completed before any memory access is started.

Instruction Format:



Clock Cycles: 1

Execution Units: Memory

MFSEL – Move from Selector Register

Description:

This instruction moves a selector register indirectly specified by Rb or directly if Rb is a constant, to register Rt.

Instruction Format: OSR2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$28h_7$	m_3	z	~	Tb_2	Rb_6	$\sim_6$	Rt_6	v	$07h_8$

Operation:

Rt = Sel[Rb]

Examples:

```
LDI $t1,#7
MFSEL $t0,[$t1]          ; get CS
STO $t0,CSSave
```

Execution Units: ALU

Clock Cycles:

Exceptions: none

MTSEL – Move to Selector Register

Description:

This instruction moves register Ra to the selector register identified indirectly by the contents of Rb or directly if Rb is a constant. This instruction will load the associated descriptor cache from either the GDT or LDT tables.

It is not possible to move a value directly to the CS register as that would cause an unexpected change of program flow. Instead, the CS register must be loaded via a branch instruction.

Instruction Format: OSR2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$29h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	$\sim_6$	v	$07h_8$

Operation:

B[Rb] = Ra

Selector Registers:

#	Reg	Usage
0	ZS	
1	DS	data selector
2	ES	
3	FS	
4	GS	
5	HS	
6	SS	stack selector
7	CS	code selector
8	PMA0	physical memory attributes
9	PMA1	
10	PMA2	
11	PMA3	
12	PMA4	
13	PMA5	
14	PMA6	
15	PMA7	
16	PTA	page table address
17	LDT	local descriptor table selector
18	KYT	memory key table selector

Examples:

```
MTSEL DS,$a0
LDI $a1,#3
MTSEL [$a1],$a0         ; move indirect $a0 to b[$a1]
```

Execution Units: ALU

Clock Cycles:

Exceptions: none

Notes:

PEEKQ – Peek at Queue / Stack

Description:

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

Instruction Format: OSR2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$0Ah_7$	m_3	z	~	Tb_2	Rb_6	$\sim_6$	Rt_6	v	$07h_8$

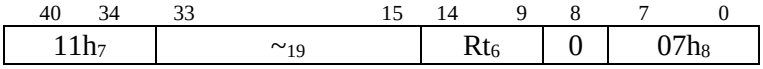
Exceptions: none

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: OSR2



Clock Cycles: 1 (if no exception present)

Operation:

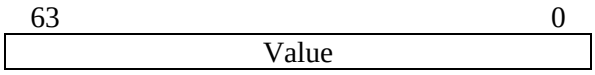
```
if (irq <> 0)
        Rt[7:0] = cause code
        Rt[11:8] = irq level
        PMSTACK = (PMSTACK << 4) | 6
        CAUSE = Const8
        C6 = IP
        IP = tvec[3]
```

Execution Units: Branch

POPQ – Pop from Queue / Stack

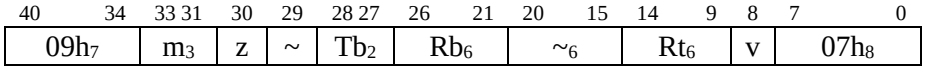
Description:

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



Value: the value that was pushed to the queue

Instruction Format: OSR2



Exceptions: none

Notes:

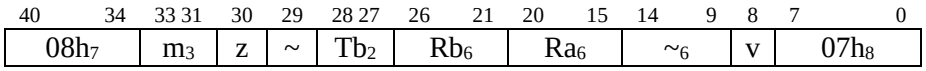
Queue #15 is the instruction trace que

PUSHQ – 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: OSR2



Instruction Format: PUSHQ

Exceptions: none

REX – Redirect Exception

Description:

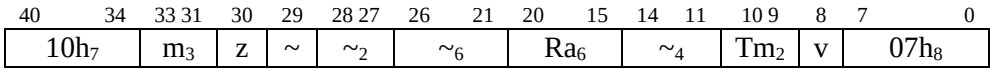
This instruction redirects an exception from an operating mode to a lower operating mode. This instruction if successful jumps to the target exception handler and does not return. If this instruction fails execution will continue with the next instruction.

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



Tm_2	
0	redirect to user mode
1	redirect to supervisor mode
2	redirect to hypervisor mode
3	reserved

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 machine mode the machine 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 (C6). 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 $constant_{12}$ field specifies a return offset in terms of bytes.

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

Instruction Format: OSR2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$13h_7$	m_3	z	~	Tb_2	Rb_6	$Constant_{12}$		0	$07h_8$

Flags Affected: none

Operation:

PMSTACK = PMSTACK >> 4
Semaphore[0] = 0
Semaphore[Rb] = 0
IP = C6 + Constant

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

SEI – Set Interrupt Level

Description:

The interrupt mask is set, disabling lower level maskable interrupts. The current interrupt mask level is stored in the target register Rt. The new interrupt mask level is set to the contents of Rb. This instruction is available only in machine mode.

Instruction Format:

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$\sim_7$	m_3	z	~	Tb_2	Rb_6	$\sim_6$	Rt_6	0	FBh_8

Clock Cycles: 1

Operation:

$im = Rb_{[2:0]}$

Exceptions: none

STATQ – Get Status of Queue / Stack

Description:

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

63	62	61 48	47 0
Qe	Dv	Data Count	Extended Data (XD)

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

XD: The high order 48 bits of the data stored by the queue if the queue is wider than 64 bits.

Instruction Format: OSR2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$0Bh_7$	m_3	z	~	Tb_2	Rb_6	$\sim_6$	Rt_6	v	$07h_8$

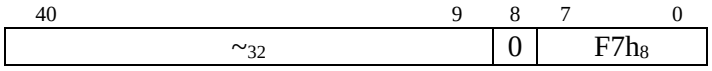
Exceptions: none

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:

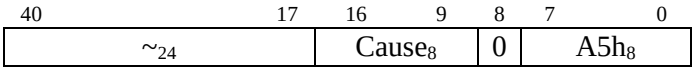


SYS – Call system routine

Description:

This instruction invokes the system exception handler. The return address is stored in the EIP register (code address register #6). This instruction causes the core to switch to machine mode.

Instruction Format:



Operation:

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. A random way may be selected for write by setting the 'r' bit in the Ra value.

The entry number for Ra comes from virtual address bits 12 to 21.

Page numbers are in terms of a 4kB page size.

Instruction Format: OSR2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$1Eh_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$07h_8$

Clock Cycles: 5

Execution Units: Memory

Ra Value Format

63	62 16	15	14 12	11 10	9 0
w	~	r	~	way	entry no

Rb/Rt Value Format

63 56	55	54	53	52 48	47 26	25 22	21 0
ASID	G	D	A	UCRWX	VPN	$~_4$	PPN

Bits		Meaning	
0 to 21	PPN	Physical page number (bits 22 to 43)	
22 to 25	~	reserved (expansion of physical page number)	
26 to 49	VPN	Virtual page number high address order bits 22 to 43	
48	X	1 = page is executable	These three combined indicate page present (P) 0 = not present
49	W	1 = page is writeable	
50	R	1 = page is readable	
51	C	1 = page is cachable	
52	U	reserved for system usage	
53	A	Accessed, set if translation was used	
54	D	Dirty, set if a write occurred to the page	
55	G	Global, global translation indicator	
56 to 63	ASID	ASID address space identifier	

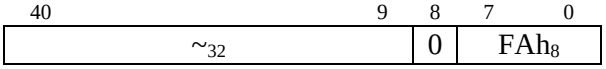
Exceptions: none

WFI – Wait for Interrupt

Description:

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

Instruction Format: SYS



Clock Cycles: 1 (if no exception present)

Execution Units: Branch

Vector Specific Instructions

V2BITS

Description

Convert Boolean vector to bits. A bit specified by Rb 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.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$3Dh_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Operation

```
For x = 0 to VL-1
        if (Vm[x])
                Rt.bit[x] = Ra[x].bit[Rb]
        else if (z)
                Rt.bit[x] = 0
        else
                Rt.bit[x] = Rt.bit[x]
```

Exceptions: none

VBITS2V

Description

This is an alternate mnemonic for the CMOVNZ instruction where Rb = 1 and Rc = 0. The effect is to generate a Boolean vector from the contents of register Ra.

Instruction Format: R3

40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
m_3	z	2_2	0_6	2_2	1_6	Ra_6	Rt_6	1	$54h_8$

Operation

For x = 0 to VL-1

if (Vm[x]) Vt[x] = Ra.bit[x] ? 1 : 0

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

Exceptions: none

VGIDX – Generate Index

Description

A value in a register Ra is multiplied by the element number and added to a value in Rb and copied to elements of vector register Vt guided by a vector mask register. Ra is a scalar register. This operation may be used to compute memory addresses for a subsequent vector load or store operation. Only the low order 24-bits of Ra are involved in the multiply. The result of the multiply is a product less than 41 bits in size. The multiply is a fast 24x16 bit multiply.

Instruction Format: R1

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$3Ch_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Operation

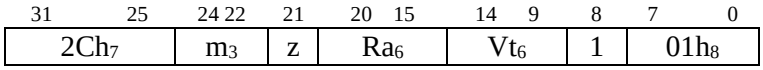
```
y = 0
for x = 0 to VL - 1
    if (Vm[x])
        Vt[y] = Ra * y + Rb
        y = y + 1
```

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



Operation

```
y = 0
for x = 0 to VL - 1
    if (Vm[x])
        Vt[y] = Va[x]
        y = y + 1
```

VEINS / VMOVSV – Vector Element Insert

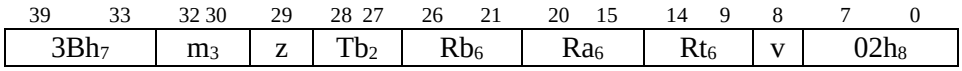
Synopsis

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.

Instruction Format: R2



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. Ra and Rt are scalar registers.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$3Ah_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	1	$02h_8$

Operation

Rt = Vb[Ra]

Exceptions: none

MFVM – Move from Vector Mask

Description

Move a mask register to a general-purpose register.

Instruction Format: VMR2

23 18	19	17 15	14 9	8	7 0
$11h_4$	~	Vmb_3	Rt_6	0	$52h_8$

Operation

Rt = Vmb

Execution Units: ALUs

MFVL – Move from Vector Length

Description

Move vector length register to a general-purpose register.

Instruction Format: R1

23 18	19	17 15	14 9	8	7 0
$13h_4$	~	$~_3$	Rt_6	0	$52h_8$

Operation

Rt = VL

Execution Units: ALUs

MTVM – Move to Vector Mask

Description

Move a general-purpose register to a mask register.

Instruction Format: VMR2

23 18	19	17 12	11 9	8	7 0
$10h_4$	~	Ra_6	Vmt_3	0	$52h_8$

Operation

Vmt = Ra

Execution Units: ALUs

MTVL – Move to Vector Length

Description

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

Instruction Format: VMR2

23 18	19	17 12	11 9	8	7 0
$12h_4$	~	Ra_6	$\sim_3$	0	$52h_8$

Operation

VL = Ra

Execution Units: ALUs

VMADD – Vector Mask Add

Description:

Add the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2

23 19	18	17 15	14 12	11 9	8	7 0
$04h_5$	~	Vmb_3	Vma_3	Vmt_3	0	$52h_8$

1 clock cycle

Exceptions: none

VMAND – Vector Mask And

Description:

Bitwise ‘and’ the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2

23 19	18	17 15	14 12	11 9	8	7 0
$08h_5$	~	Vmb_3	Vma_3	Vmt_3	0	$52h_8$

1 clock cycle

Exceptions: none

VMCNTPOP – Count Population

CNTPOP r1,vm2

Description:

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

Instruction Format: VMR2

23 19	18	17 15	14 9	8	7 0
$0Dh_5$	~	Vmb_3	Rt_6	0	$52h_8$

Execution Units: integer ALU

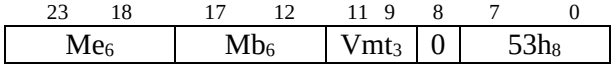
Exceptions: none

VMFILL – Vector Mask Fill

Description:

Fill the contents of a vector mask register with a mask of ones beginning at Mb_6 and ending at Me_6 inclusive. Fill the remainder of the register with zeros.

Instruction Format:



1 clock cycle

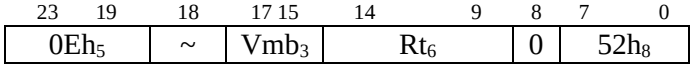
Exceptions: none

VMFIRST – 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 65536. The search begins at the least significant bit and proceeds to the most significant bit.

Instruction Format: R1



Operation

Rt = first set bit number of (Vm)

Exceptions: none

Execution Units: ALUs

VMLAST – 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 65536. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

Instruction Format: VMR2

23 19	18	17 15	14 9	8	7 0
$0Fh_5$	~	Vmb_3	Rt_6	0	$52h_8$

Operation

Rt = last set bit number of (Vm)

Exceptions: none

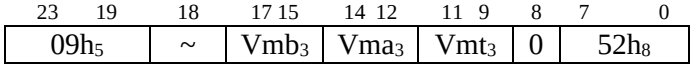
Execution Units: ALUs

VMOR – Vector Mask Or

Description:

Bitwise ‘or’ the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2



1 clock cycle

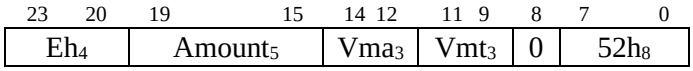
Exceptions: none

VMSLL – Vector Mask Shift Left Logical

Description:

Shift a vector mask register to the left up to 31 bits.

Instruction Format: VMR2



1 clock cycle

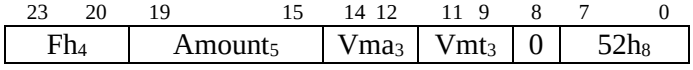
Exceptions: none

VMSRL – Vector Mask Shift Right Logical

Description:

Shift a vector mask register to the right up to 31 bits.

Instruction Format: VMR2



1 clock cycle

Exceptions: none

VMSUB – Vector Mask Subtract

Description:

Subtract the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2

23 19	18	17 15	14 12	11 9	8	7 0
$05h_5$	~	Vmb_3	Vma_3	Vmt_3	0	$52h_8$

1 clock cycle

Exceptions: none

VMXOR – Vector Mask Exclusive Or

Description:

Bitwise ‘or’ the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2

23 19	18	17 15	14 12	11 9	8	7 0
$0Ah_5$	~	Vmb_3	Vma_3	Vmt_3	0	$52h_8$

1 clock cycle

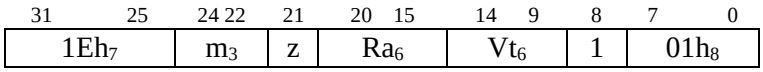
Exceptions: none

VSCAN

Description

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

Instruction Format: R1



Operation

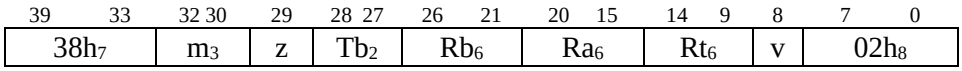
```
sum = 0
for x = 0 to VL - 1
        Vt[x] = sum
        if (Vm[x])
                sum = sum + Ra
```

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.

Instruction Format: R2



Operation

Amt = Rb

For x = VL-1 to Amt

    Vt[x] = Va[x-amt]

For x = Amt-1 to 0

    Vt[x] = 0

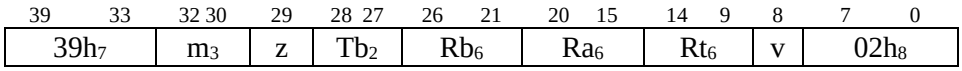
Exceptions: none

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.

Instruction Format: R2



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

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: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$50h_7$	m_3	z	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: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$50h_7$	m_3	z	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: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$50h_7$	m_3	z	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: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$50h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Operation:

Rt = Ra & Rb

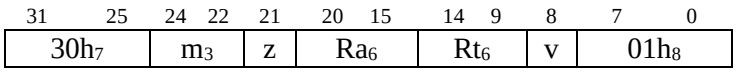
Exceptions: none

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: R1



Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

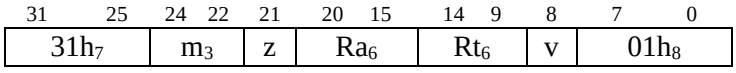
Exceptions: none

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: R1



Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

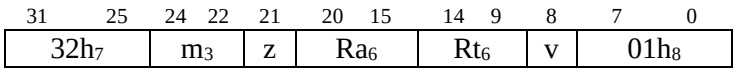
Exceptions: none

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: R1



Operation:

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

Execution Units: ALU #0

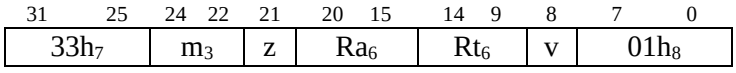
Exceptions: none

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: R1



Operation:

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

Execution Units: ALU #0

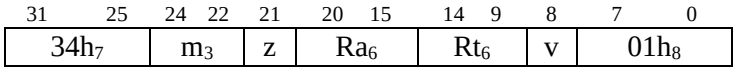
Exceptions: none

SHA512SIG0

Description:

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

Instruction Format: R1



Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

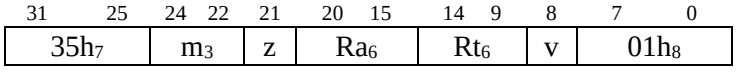
Exceptions: none

SHA512SIG1

Description:

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

Instruction Format: R1



Clock Cycles: 1

Operation:

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

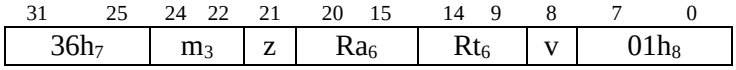
Execution Units: ALU #0

Exceptions: none

SHA512SUM0

Description:

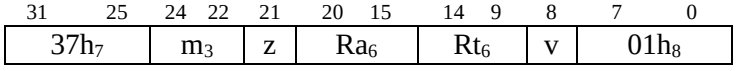
Instruction Format: R1



SHA512SUM1

Description:

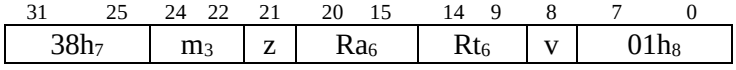
Instruction Format: R1



SM3P0

Description:

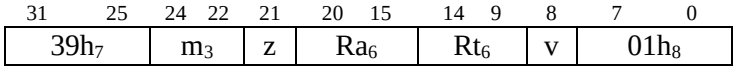
Instruction Format: R1



SM3P1

Description:

Instruction Format: R1



SM4ED

Description:

Instruction Format:

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$0E_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$03h_8$

SM4KS

Description:

Instruction Format:

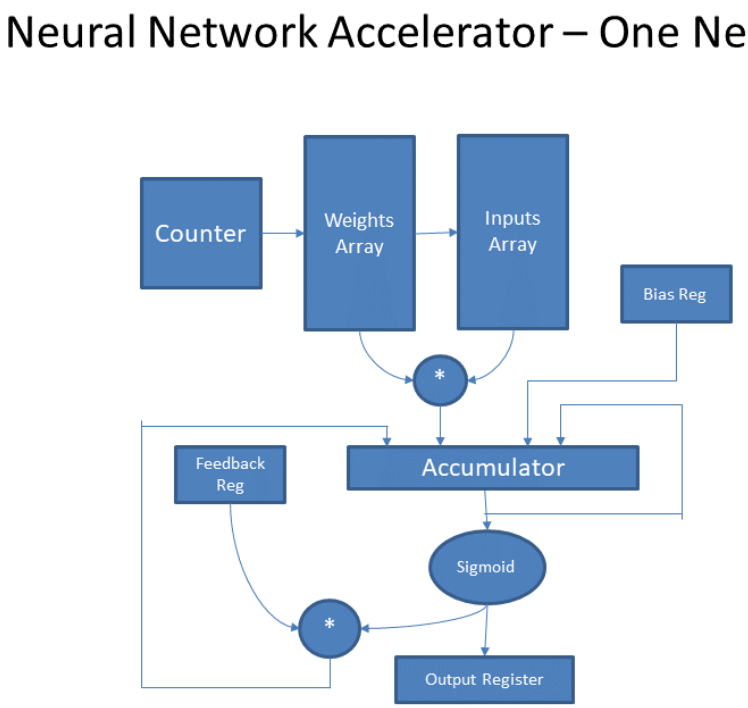
47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$0F_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$03h_8$

Neural Network Accelerator Instructions

Overview

Included in the ISA are instructions for neural network acceleration. Each neuron is composed of an accumulator that sums the product of weights and inputs and an output activation function. Neurons may be biased with a bias value and may also have feedback from output to input via a feedback constant. The neurons are implemented using 16.16 fixed-point arithmetic. There are 8 neurons in a single layer which may calculate simultaneously. Following is a sketch of the NNA organization. The weights and input arrays have a depth of 1024 entries. Not all entries need be used. The number of entries in use is configurable programmatically with the base count and maximum count register using the NNA_MTBC and NNA_MTMC instruction.

Several of the NNA instructions allow multiple neurons to be updated at the same time by representing the neuron update list as a bitmask.

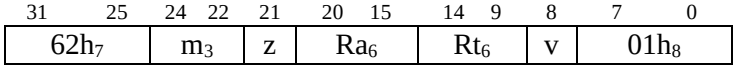


NNA_MFACT – Move from Output Activation

Description:

Move from activation output register. Move a value from the neuron's activation register output to the target register Rt. Bits 0 to 3 of Ra specify the neuron.

Instruction Format: R1



Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTBC – Move to Base Count

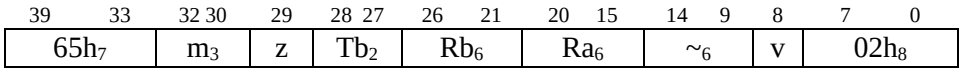
Description:

Move to base count register. Move the value in Ra to the base count register for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the base count value.

The neuron calculates the activation output using weight and input array entries between the base count and maximum count inclusive.

Manipulating the base count and maximum count registers ease the implementation of multi-layer networks that do not require the use of all array entries.

Instruction Format: R2



Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTBIAS – Move to Bias

Description:

Move to bias value. Move the value in Ra to the bias register for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the bias value.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$62h_7$	m_3	z	Tb_2	Rb_6	Ra_6	$\sim_6$	v	$02h_8$

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTFB – Move to Feedback

Description:

Move to feedback constant. Move the value in Ra to the feedback constant for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the feedback constant.

The feedback constant acts to create feedback in the neuron by multiplying the output activation level by the feedback constant and using the result as an input. If no feedback is desired then this constant should be set to zero.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$63h_7$	m_3	z	Tb_2	Rb_6	Ra_6	$\sim_6$	v	$02h_8$

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTIN – Move to Input

Description:

Move to input array. Move the value in Ra to the input memory cell identified with Rb. Bits 0 to 15 of Rb specify the memory cell address, bits 32 to 63 of Rb are a bit mask specifying the neurons to update. Bits 0 to 15 of Rb are incremented and stored in Rt.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$61h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 1

Execution Units: NNA

Notes:

Multiple neurons may have their inputs updated at the same time with the same value. All the neurons may have the same inputs but the weights for the individual neurons would be different so that a pattern may be recognized.

NNA_MTMC – Move to Max Count

Description:

Move to maximum count register. Move the value in Ra to the maximum count register for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the maximum count value.

The maximum count is the upper limit of inputs and weights to use in the calculation of the activation function. The maximum count should not exceed the hardware table size. The table size is 1024 entries.

The neuron calculates the activation output using weight and input array entries between the base count and maximum count inclusive.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$64h_7$	m_3	z	Tb_2	Rb_6	Ra_6	$\sim_6$	v	$02h_8$

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTWT – Move to Weights

Description:

Move to weights array. Move the value in Ra to the weight memory cell identified with Rb. Bits 0 to 15 or Rb specify the memory cell address, bits 32 to 63 of Rb are a bit mask specifying the neurons to update. Bits 0 to 15 of Rb are incremented and stored in Rt.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$60h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 1

Execution Units: NNA

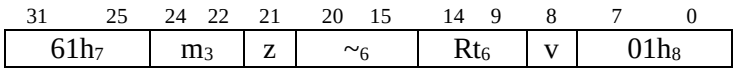
Notes:

NNA_STAT – Get Status

Description:

This instruction gets the status of the neurons. There is a bit in Rt for each neuron. A bit will be set if the neuron is finished performing the calculation of the activation function, otherwise the bit will be clear.

Instruction Format: R1



Clock Cycles: 1

Execution Units: NNA

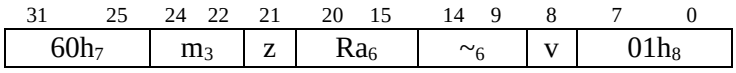
Notes:

NNA_TRIG – Trigger Calc

Description:

This instruction triggers a NNA cycle for the neurons identified in the bit mask. The bit mask is contained in register Ra.

Instruction Format: R1



Clock Cycles: 1

Execution Units: NNA

Notes:

Graphics

Co-ordinates

Co-ordinates are specified as 16.16 fixed point numbers. x, y, z co-ordinates occupy bits 0 to 31, 32 to 63, and 64 to 95 respectively of a register.

127 96	95 64	63 32	31 0
~	z coord	y coord	x coord

Colors

Colors are represented using RGB888 format. Colors are placed in the low order 24-bits of a register.

127 32	31 24	23 16	15 8	7 0
~	Z-order	Blue	Green	Red

BLEND – Blend Colors

Description:

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

Instruction Format: R3

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 11	8	7 0
Ch_4	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$03h_8$

Operation:

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

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

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

Clock Cycles: 2

CLIP – Clip Point

Description:

The clip instruction checks that the point in Ra is within the graphics target area always and clip region if enabled. The target and clip areas must have been previously set. If the point should be clipped a one is set in Rt, otherwise Rt is set to zero.

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
$20h_7$	$\sim_5$	m_3	z	Ta	Ra_5	Tt	Rt_5	v	$01h_7$

Clock Cycles: 2

PLOT – Plot Point

Description:

This instruction plots a point in the graphics target area. The point's co-ordinates are in Ra, the color to use is in Rb.

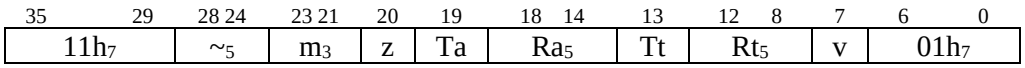
35 29	2827	2625	24 20	19	18 14	13	12 8	7	6 0
$34h_7$	$\sim_2$	Tb_2	Rb_5	Ta	Ra_5	Tt	Rt_5	v	$02h_7$

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.



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
$35h_7$	$\sim_2$	Tb_2	Rb_5	Ta	Ra_5	Tt	Rt_5	v	$02h_7$

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

Opcode Maps

Root Opcode

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	{R1}	{R2}	{R3}	ADDI	SUBFI	MULI	{SYS}	ANDI	ORI	EORI		ADCI	SBCFI	MULUI	{CSR}
1x	JGATE	{R1R}	{R2R}	{R3R}		MULFI	SEQI	SNEI	SLTI	SLTIL	SGTIL	SGTI	SLTUI	SLTUIL	SGTUIL	SGTUI
2x	BRA				BBC	BBS	BEQ	BNE	BLT	BGE	BLE	BGT	BLTU	BGEU	BLEU	BGTU
3x	BRA				BBC	BBS	BEQ	BNE	BLT	BGE	BLE	BGT	BLTU	BGEU	BLEU	BGTU
4x	DIVI	CPUID	DIVIL	PTRDIF		CHKI							BMAP	CHK	MULUI	DIVUI
5x	CMPI	MLO	{VM}	VMFILL	CMOVNZ	BYTNDX	WYDENDX	UTF21NDX					CMPUI	CHKXI		
6x		{FLT1}	{FLT2}	{FLT3}	MUX	{DFLT1}	{DFLT2}	{DFLT3}		{PST1}	{PST2}	{PST3}				
7x		{FLT1L}	{FLT2L}			{DFLT1}	{DFLT2}			{PST1R}	{PST2R}					
8x	LDB	LDBU	LDW	LDWU	LDT	LDTU	LDO			LLAL	LLAH	LDVOAR		JSRI	LWS	LCL
9x	STB	STW	STT	STO	STOC			CAS	STSET	STMOV	STCMP	STFND			SWS	CACHE
Ax						SYS	INT	MOV			{bitfld}	MOVS				
Bx	{LDxX}							JSRIX						LV	LVWS	
Cx	{STxX}								PUSH	PEA	POP	LINK	UNLINK	SV	SVWS	
Dx	LDBL	LDBUL	LDWL	LDWUL	LDTL	LDTUL	LDOL			LLALL	LLAHL					CACHEL
Ex	STBL	STWL	STTL	STOL	STOCL											
Fx		NOP	RTS			{BCD}	STP	SYNC	MEMSB	MEMDB	WFI	SEI				

{LDxX} Scaled Indexed Loads – $Func_7$

	000	001	010	011	100	101	110	111
00	LDBX	LDBUX	LDWX	LDWUX	LDTX	LDTUX	LDOX	
01	LLALX	LLAHX	CACHEX					
10								
11								

{STxX} Scaled Indexed Stores – $Func_6$

	000	001	010	011	100	101	110	111
00	STBX	STWX	STTX	STOX	STOCX			
01								
10								
11								

{R1 – 0x01} Integer Monadic Register Ops – $Func_7$

	000	001	010	011	100	101	110	111
x000	CNTLZ	CNTLO	CNTPOP				ABS	NABS
x001	SQRT			TST				
x010	PTRINC	TRANSFORM						
x011					VCMPRSS		VSCAN	
x100	CLIP							
x101	REVBIT.BP	REVBIT.WP	REVBIT.TP	REVBIT.OP				
x110	SHA256 SIG0	SHA256 SIG1	SHA256 SUM0	SHA256 SUM1	SHA512 SIG0	SHA512 SIG1	SHA512 SUM0	SHA512 SUM1
x111	SM3P0	SM3P1						
1000								
1001								
1010	AES64IM							
1011								
1100	NNA_TRIG	NNA_STAT	NNA_MFACT					
1101								

{R2 – 0x02} Integer Dyadic Register Ops – Func7

	000	001	010	011	100	101	110	111
0000	NAND	NOR	XNOR	ORC	ADD	SUB	MUL	
0001	AND	OR	XOR	ANDC	ADC	SBC	MULU	MULH
0010	DIV	DIVU	DIVSU			MULF	MULSU	PERM
0011	DIF		BYTNDX	WYDNDX	U21NDX	MULSUH	MULUH	MYST
0100	SLT / SGT	SGE / SLE	SLTU / SGTU	SGEU / SLEU			SEQ	SNE
0101	MIN	MAX	CMP	CMPU	BIT		CLMUL	CLMULH
0110	BMM.or	BMM.xor	BMM	BMM	PLOT	RW_COEFF	LIN2PL	LIN2PH
0111	VSLLV	VSLRV	VEX	VEINS	VGIDX	V2BITS	VBITS2V	
1000	SLL	SRL	SRA	ROL	ROR		SLLP	SRLP
1001	SLLT	SRLT	SRAT	ROLT	RORT			
1010	AES64DS	AES64DSM	AES64ES	AES64ESM	AES64KS1I	AES64KS2		
1011								
1100	NNA_MTWT	NNA_MTIN	NNA_MTBIAS	NNA_MTFB	NNA_MTMC	NNA_MTBC		
1101								
1110								
1111								

{R3/R4 – 0x03} Triadic Register Ops

	000	001	010	011	100	101	110	111
0	BMAP	PTRDIF	CHK		MUX		CMOVNZ	
1	SLLP	SLLPI			BLEND	FDP	SM4ED	SM4KS

{F1/F1L - 0x61/0x71} Floating-Point Monadic Ops – Funct$_7$

	000	001	010	011	100	101	110	111
x000	FMOV	FRSQRTE	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/F2L – 0x62,0x72} Floating-Point Dyadic Ops – Funct$_7$

	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 – 0x63} Floating-Point Dyadic Ops – Funct$_7$

	000	001	010	011	100	101	110	111
0	FMA	FMS	FNMA	FNMS				
1								

{DF2} Decimal Floating-Point Dyadic Ops – Funct$_7$

	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 – 0x52} Vector Mask Register Ops – $Func_5$

	000	001	010	011	100	101	110	111
00					VMADD	VMSUB		
01	VMAND	VMOR	VMXOR			VMCNTPOP	VMFIRST	VMLAST
10	MTVM	MFVM	MTVL	MFVL				
11					VMSLL	VMSLL	VMSRL	VMSRL

{OSR2} System Ops

	000	001	010	011	100	101	110	111
x000								EXEC
x001	PUSHQ	POPQ	PEEKQ	STATQ		POPQI	PEEKQI	STATQI
x010	REX	PFI	WAI	RTE	SETKEY		DI	SEI
x011	SETTO	GETTO	GETZL			MVSEG	TLBRW	SYNC
x100	CSAVE	CRESTORE					BASE	
x101	MFSEL	MTSEL						
x110	MVCI							
x111								

Glossary

AMO

AMO stands for atomic memory operation. An atomic memory operation typically reads then writes to memory in a fashion that may not be interrupted by another processor. Some examples of AMO operations are swap, add, and, and or.

ATC

ATC stands for address translation cache. This buffer is used to cache address translations for fast memory access in a system with an mmu capable of performing address translations. The address translation cache is more commonly known as the TLB.

Burst Access

A burst access is several bus accesses that occur rapidly in a row in a known sequence. If hardware supports burst access the cycle time for access to the device is drastically reduced. For instance, dynamic RAM memory access is fast for sequential burst access, and somewhat slower for random access.

BTB

An acronym for Branch Target Buffer. The branch target buffer is used to improve the performance of a processing core. The BTB is a table that stores the branch target from previously executed branch instructions. A typical table may contain 1024 entries. The table is typically indexed by part of the branch address. Since the target address of a branch type instruction may not be known at fetch time, the address is speculated to be the address in the branch target buffer. This allows the machine to fetch instructions in a continuous fashion without pipeline bubbles. In many cases the calculated branch address from a previously executed instruction remains the same the next time the same instruction is executed. If the address from the BTB turns out to be incorrect, then the machine will have to flush the instruction queue or pipeline and begin fetching instructions from the correct address.

Card Memory

A card memory is a memory reserved to record the location of pointer stores in a garbage collection system. The card memory is much smaller than main memory; there may be card memory entry for a block of main memory addresses. Card memory covers memory in 128 to 512-byte sized blocks. Usually a byte is dedicated to record the pointer store status even though a bit would be adequate, for performance reasons. The location of card memory to update is found by shifting the pointer value to the right some number of bits (7 to 9 bits) and then adding the base address of the table. The update to the card memory needs to be done with interrupts disabled.

FPGA

An acronym for Field Programmable Gate Array. FPGA's consist of a large number of small RAM tables, flip-flops, and other logic. These are all connected with a programmable connection network. FPGA's are 'in the field' programmable, and usually re-programmable. An FPGA's re-programmability is typically RAM based. They are often used with configuration PROM's so they may be loaded to perform specific functions.

HDL

An acronym that stands for 'Hardware Description Language'. A hardware description language is used to describe hardware constructs at a high level.

Instruction Bundle

A group of instructions. It is sometimes required to group instructions together into bundle. For instance, all instructions in a bundle may be executed simultaneously on a processor as a unit. Instructions may also need to be grouped if they are oddball in size for example 41 bits, so that they can be fit evenly into memory. Typically, a bundle has some bits that are global to the bundle, such as template bits, in addition to the encoded instructions.

Instruction Pointers

A processor register dedicated to addressing instructions in memory. It is also often called a program counter. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

Instruction Prefix

An instruction prefix applies to the following instruction to modify its operation. An instruction prefix may be used to add more bits to a following immediate constant, or to add additional register fields for the instruction. The prefix essentially extends the number of bits available to encode instructions. An instruction prefix usually locks out interrupts between the prefix and following instruction.

Instruction Modifier

An instruction modifier is similar to an instruction prefix except that the modifier may apply to multiple following instructions.

ISA

An acronym for Instruction Set Architecture. The group of instructions that an architecture supports. ISA's are sometimes categorized at extreme edges as RISC or CISC. RTF64 falls somewhere in between with features of both RISC and CISC architectures.

Keyed Memory

A memory system that has a key associated with each page to protect access to the page. A process must have a matching key in its key list in order to access the memory page. The key is often 20 bits or larger. Keys for pages are usually cached in the processor for performance reasons. The key may be part of the paging tables.

Linear Address

A linear address is the resulting address from a virtual address after segmentation has been applied.

Opcode

A short form for operation code, a code that determines what operation the processor is going to perform. Instructions are typically made up of opcodes and operands.

Operand

The data that an opcode operates on, or the result produced by the operation. Operands are often located in registers. Inputs to an operation are referred to as source operands, the result of an operation is a destination operand.

Physical Address

A physical address is the final address seen by the memory system after both segmentation and paging have been applied to a virtual address. One can think of a physical address as one that is "physically" wired to the memory.

Physical Memory Attributes (PMA)

Memory usually has several characteristics associated with it. In the memory system there may be several different types of memory, rom, static ram, dynamic ram, eeprom, memory mapped I/O devices, and others. Each type of memory device is likely to have different characteristics. These characteristics are called the physical memory attributes. Physical memory attributes are associated with address ranges that the memory is located in. There may be a hardware unit dedicated to verifying software is adhering to the attributes associated with the memory range. The hardware unit is called a physical memory attributes checker (PMA checker).

Program Counter

A processor register dedicated to addressing instructions in memory. It is also often and perhaps more aptly called an instruction pointer. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

ROB

An acronym for ReOrder Buffer. The re-order buffer allows instructions to execute out of order yet update the machine's state in order by tracking instruction state and variables. In FT64 the re-order buffer is a circular queue with a head and tail pointers. Instructions at the head are committed if done to the machine's state then the head advanced. New instructions are queued at the buffer's tail as long as there is room in the queue. Instructions in the queue may be processed out of the order that they entered the queue in depending on the availability of resources (register values and functional units).

RSB

An acronym that stands for return stack buffer. A buffer of addresses used to predict the return address which increases processor performance. The RSB is usually small, typically 16 entries. When a return instruction is detected at time of fetch the RSB is accessed to determine the address of the next instruction to fetch. Predicting the return address allows the processing core to continuously fetch instructions in a speculative fashion without bubbles in the pipeline. The return address in the RSB may turn out to be detected as incorrect during execution of the return instruction, in which case the pipeline or instruction queue will need to be flushed and instructions fetched from the proper address.

SIMD

An acronym that stands for 'Single Instruction Multiple Data'. SIMD instructions are usually implemented with extra wide registers. The registers contain multiple data items, such as a 128-bit register containing four 32-bit numbers. The same instruction is applied to all the data items in the register at the same time. For some applications SIMD instructions can enhance performance considerably.

Stack Pointer

A processor register dedicated to addressing stack memory. Sometimes this register is assigned by convention from the general register pool. This register may also sometimes index into a small dedicated stack memory that is not part of the main memory system. Sometimes machines have multiple stack pointers for different purposes, but they all work on the idea of a stack. For instance, in Forth machines there are typically two stacks, one for data and one for return addresses.

Telescopic Memory

A memory system composed of layers where each layer contains simplified data from the topmost layer downwards. At the topmost layer data is represented verbatim. At the bottom layer there may be only a single bit to represent the presence of data. Each layer of the telescopic memory uses far less memory than the layer above. A telescopic memory could be used in garbage collection systems. Normally however the extra overhead of updating multiple layers of memory is not warranted.

TLB

TLB stands for translation look-aside buffer. This buffer is used to store address translations for fast memory access in a system with an mmu capable of performing address translations.

Vector Length (VL register)

The vector length register controls the maximum number of elements of a vector that are processed. The vector length register may not be set to a value greater than the number of elements supported by hardware. Vector registers often contain more elements than are required by program code. It would be wasteful to process all elements when only a few are needed. To improve the processing performance only the elements up to the vector length are examined.

Vector Mask (VM)

A vector mask is used to restrict which elements of a vector are processed during a vector operation. A one bit in a mask register enables the processing for that element, a zero bit disables it. The mask register is commonly set using a vector set operation.

Miscellaneous

Reference Material

Below is a short list of some of the reading material the author has studied. The author has downloaded a fair number of documents on computer architecture from the web. Too many to list.

Modern Processor Design Fundamentals of Superscalar Processors by John Paul Shen, Mikko H. Lipasti. Waveland Press, Inc.

Computer Architecture A Quantitative Approach, Second Edition, by John L Hennessy & David Patterson, published by Morgan Kaufman Publishers, Inc. San Franciso, California is a good book on computer architecture. There is a newer edition of the book available.

Memory Systems Cache, DRAM, Disk by Bruce Jacob, Spencer W. Ng., David T. Wang, Samuel Rodriguez, Morgan Kaufman Publishers

PowerPC Microprocessor Developer's Guide, SAMS publishing. 201 West 103rd Street, Indianapolis, Indiana, 46290

80386/80486 Programming Guide by Ross P. Nelson, Microsoft Press

Programming the 286, C. Vieillefond, SYBEX, 2021 Challenger Drive #100, Alameda, CA 94501

Tech. Report UMD-SCA-2000-02 ENEE 446: Digital Computer Design — An Out-of-Order RiSC-16

Programming the 65C816, David Eyes and Ron Lichty, Western Design Centre Inc.

Microprocessor Manuals from Motorola, and Intel,

The SPARC Architecture Manual Version 8, SPARC International Inc, 535 Middlefield Road. Suite210 Menlo Park California, CA 94025

The SPARC Architecture Manual Version 9, SPARC International Inc, Sab Jose California, PTR Prentice Hall, Englewood Cliffs, New Jersey, 07632

The MMIX processor: http://mmix.cs.hm.edu/doc/instructions-en.html

RISCV 2.0 Spec, Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovi´c CS Division, EECS Department, University of California, Berkeley {waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu

The Garbage Collection Handbook, Richard Jones, Antony Hosking, Eliot Moss published by CRC Press 2012

Trademarks

WISHBONE Compatibility Datasheet

The Thor2021 core may be directly interfaced to a WISHBONE compatible bus.

WISHBONE Datasheet WISHBONE SoC Architecture Specification, Revision B.3		
Description:	Specifications:	
General Description:	Central processing unit (CPU core)	
Supported Cycles:	MASTER, READ / WRITE MASTER, READ-MODIFY-WRITE MASTER, BLOCK READ / WRITE, BURST READ (FIXED ADDRESS)	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing	128 bit 8 bit 128 bit Little Endian any (undefined)	
Clock frequency constraints:	tm_clk_i must be >= 10MHz	
Supported signal list and cross reference to equivalent WISHBONE signals	Signal Name: ack_i adr_o(31:0) clk_i dat_i(127:0) dat_o(127:0) cyc_o stb_o wr_o sel_o(7:0) cti_o(2:0) bte_o(1:0)	WISHBONE Equiv. ACK_I ADR_O() CLK_I DAT_I() DAT_O() CYC_O STB_O WE_O SEL_O CTI_O BTE_O
Special Requirements:		