2016

Thor Guide

This document contains information pertaining to the Thor processor including the instruction set and formats and softcore interfacing.

robfinch@Finitron.ca
Finitron
1/1/2016

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 one and eight bytes in length and handles 8, 16, 32, and 64 bit data within a 64 bit address space.

Programming Model

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.

- 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.

General Registers

There are 64 general purpose registers. General purpose registers are 64 bits wide. The general registers may hold integer or floating point values.

Register #0 is always zero.

r0	always zero
r1	return value
r2	return value
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	always zero
C1	return address
C2	
C3	
C4	
C5	
C6	
C7	
C8	
C9	
C10	catch link address
C11	debug return address
C12	exception table pointer
C13	exceptioned PC
C14	interrupted PC
C15	program counter, read only

ZS	zero segment
DS	data segment
ES	extra segment
FS	
GS	
HS	
SS	stack segment
CS	code segment

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.

Code Address Registers

The processor contains sixteen code address registers (C0-C15). Several of the registers are reserved for predefined purposes. A code address register is used in the formation and storage of code addresses.

Reg #		Usage
0	Always Zero	Absolute address formation
1		Subroutine return address
2		This register is available for general use.
3		This register is available for general use.
4		This register is available for general use.
5		
6		
7		
8		
9		
10	Catch Link Register	Used by the compiler to link to try/catch handlers.
11	Debug Exception PC	This register is set when a debug exception occurs
12	Exception Table Pointer	This register points to the exception table in memory.
13	Exceptioned PC	This register is set when an exception occurs
14	Interrupted PC	This register is automatically set during a hardware interrupt
15	Program Counter	Relative address formation.

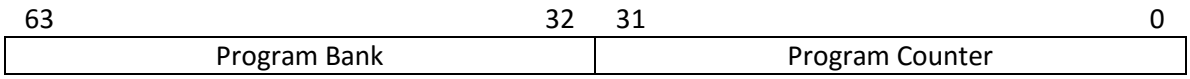
Code address registers may be used to point to a block of code from which the JSR instruction can index into with its 24 bit offset. For instance a register may contain a pointer to a class method jump list; the JSR instruction can then index into this list in order to invoke a method.

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

Setting of code address register #12 should be followed with a SYNC instruction. The core assumes c12 is essentially static and does not provide full bypassing for this register. The register value may be stale until the sync instruction executes.

Program Counter



The program counter is special in that it is always incrementing by the size of the instructions fetched as a program runs. Program code is byte aligned. To improve performance only the low order 32 bits of the program counter increment. The entire program counter may be loaded with a jump instruction. If the upper four bits of the program counter/ bank are all ones, then segmentation with the code segment is ignored.

Predicates

The processor features predicated execution of all instructions. Whether or not an instruction is executed depends on the contents of a predicate register and the predicate condition specified in the predicate byte. There are 16 predicate registers each of which hold three flags. These flags are set as the result of a compare operation. The flags represent equality (eq) signed less than (lt) and unsigned less than (ltu).

3	2	1	0
~	ltu	lt	eq

All instructions are executed conditionally determined by the value of a predicate register. The special predicate 00 executes the break vector.

Predicate Conditions

Cond.		Test	
0	PF	0	Always false – Instructions predicated with condition zero never execute regardless of the predicate register contents. This is used for extended immediate values as well. The false predicate byte for instructions is 90h.
1	PT	1	Always True – The instruction predicated with an always true condition always executes regardless of the predicate register contents. The always true predicate byte is 01h. Other true predicates are instruction short-forms.
2	PEQ	eq	Equal – instruction executes if the predicate register equal flag is set
3	PNE	!eq	Not Equal – instruction executes if the predicate register equal flag is clear
4	PLE	lt|eq	Less or Equal – predicate less or equal flag is set
5	PGT	!(lt|eq)	greater than
6	PGE	!lt	greater or equal
7	PLT	lt	less than
8	PLEU	ltu|eq	unsigned less or equal
9	PGTU	!(ltu|eq)	unsigned greater than
10	PGEU POR	!ltu	unsigned greater or equal Ordered for floating point
11	PLTU PUN	ltu	unsigned less than Unordered for floating point
12			
13	PSIG	signal	execute if external signal is true
14			
15			

Compiler Usage

The compiler uses predicate register #15 to conditionally move TRUE / FALSE values to a register when evaluating a logical operation.

Predicate registers beginning with P0 and incrementing are applied for use as the control flow nesting level increases. The compiler does not support control flow nesting more than 14 levels in a single subroutine. Predicate registers beginning with P14 and decrementing are used in the evaluation of the hook operator. Care must be taken such that the number of predicate registers in use does not exceed the number available.

Pred.	Usage	
P0	control flow level 0	
P1	control flow nesting level 1	
P2	control flow nesting level 2	
…		
Pn	control flow nesting level n (n not to exceed 14)	
…		
P12	third hook operator in an expression	
P13	second hook operator in an expression	
P14	first hook operator in an expression	
P15	conditionally moves TRUE/FALSE for logical expressions	

Status Register (SR)

This register contains bits that control the overall operation of the processor or reflect the processor's state. Bits are included for interrupt masking, and system / application mode indicator. This register is split into two halves with both halves having the same format. The lower half of the register is what determines how the processor works. The upper half of the register maintains a backup copy of the lower half for interrupt processing. There are instructions provided for manipulating the interrupt mask.

31..16	15	14	13	12	11..8	7..0
same format as 15..0	Interrupt Mask	Reserved	Kernel / Application Mode Indicator	Float Except. Enable	Register set	
	IM	~	S	FXE	RS	

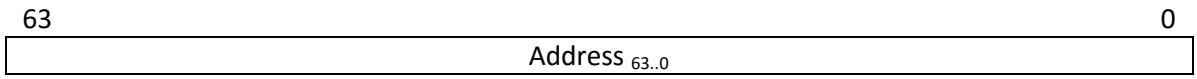
The Kernel / Application Mode indicator is read-only.

IM = interrupt mask

Maskable interrupts are disabled when this bit is set.

Debug Address Register (61,0 to 61,3)

These registers contain addresses of instruction or data breakpoints.



Debug Control Register (61,4)

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.		
7	When 1 this bit enables single stepping mode. A debug exception will be generated after the execution of an instruction.		
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 byte 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 — half 11 — all but the three LSB's should match — word		
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.		
62			
63			

Debug Status Register (61,5)

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

Summary of Special Purpose Registers

Reg #	R/W			
00-15	RW	PRED	reserved - specific predicate register #0 to 15	
16-31	RW	CREGS	Code address register array (C0 to C15)	
32-39	RW	SREGS	Segment selector register array (zs,ds,es,fs,gs,hs,ss,cs)	
40	RW	LDT	local descriptor table pointer	
41	RW	GDT	global descriptor table pointer	
42	RW	CPL	Current Privilege Level	
43	W	SEGSW	segment switch register	
44	RW	segbase	segment base address	
45	RW	seglimit	segment limit	
46	-		reserved (segmentation)	
47	RW	segacr	segment access rights	
48	RW	keys	memory keys for lot system	
49	RW	TEMP	temporary register	
50	R	TICK	Tick count	
51	RW	LC	Loop Counter	
52	RW	PREGS	Predicate register array	
53	RW	ASID	address space identifier	
54	RW	VL	Vector Length	
55	RW	SR	Status Register	
56	RW	FPSCR	Floating point status and control	
58	R	ARG1	exception / interrupt argument 1	
59	RW	EXC	exception cause register	
60	W	BIR	Breakout index register	
61	RW		Breakout register - additional spr's	
63			reserved	

Segment Descriptor for segment register selected by segsw

Additional Spr's are available by setting the breakout index register to an Sor index value, then accessing the Spr through the breakout register.

P0,P1,…P15 (PRED)

These registers allow access to the predicate registers. Accessing the predicate registers through the special purpose register array is not normally done. There are other instructions that directly access the predicate registers, such as biti and cmp. For saving / restoring all predicates at once the PREGS register can be used. These registers are four bits wide.

C0,C1,…C15 (CREGS) – SPR #16 to 31

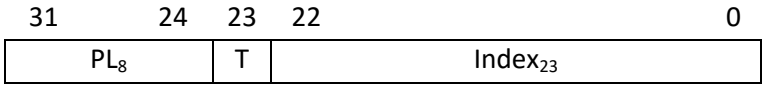
This set of registers allows access to the code address register array.

ZS,DS,ES,FS,GS,HS,SS,CS (SREGS) – SPR #32 to 39

These registers reflect the values of the segment selectors currently active in the core. Writing to a selector register triggers a segment load interrupt.

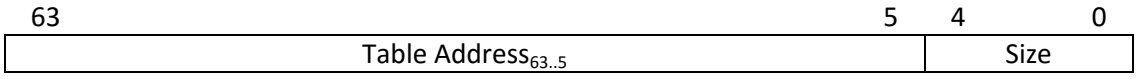
LDT – SPR #40

The LDT register holds the selector for the local descriptor table.



GDT – SPR #41

The GDT register holds the location and size of the global descriptor table. The descriptor table must be 32 byte aligned. The low order five bits of the GDT register are used to indicate the table size.



Since the table is software managed the exact meaning of the size field is up to the software implementation. It is suggested that the size field at least be a multiple of the memory page size so that the memory protection capability can be applied to the table.

SEGSW – SPR #43

The segment switch register controls which segment information is visible in the special purpose register array. Since segments are relatively large objects only one set of registers is visible at one time. This register is a three bit write-only register.

Value	Visible Components
0	ZS
1	DS
2	ES
3	FS
4	GS
5	HS
6	SS
7	CS
8	LDT

Note that the sync instruction should be used to ensure the correct register set is visible.

SEGBASE – SPR #44

The segbase register reflects the base component of the segment descriptor for the selected segment register. Which base register is present at SPR#44 is controlled by the segment switch (segsw) special purpose register. For example to access the DS segment register components set the segsw to one then the DS descriptor values will be accessible.

SEGLMT – SPR #45

This register allows access to the segment limit component of the descriptor for the selected segment register.

SEGACR – SPR #47

This register allows access to the segment access rights information component of the descriptor for the selected segment register.

KEYS – SPR #48

This register contains the collection of keys associated with the process for the memory lot system. Each key is ten bits in size. The register contains six keys.

63 60	59 50	49 40	39 30	29 20	19 10	9 0
~$_4$	key6	key5	key4	key3	key2	key1

TICK - SPR #50

The tick count register contains the number of clock cycles that have passed since the last reset.

LC – SPR #51

The loop count register is a 64 bit register used with the loop instruction to form counted loops.

PREGS – SPR #52

PREGS is a horizontal combination of all the predicate registers. Each predicate register has a four bit slot within the PREGS register. The PREGS register allows reading or writing of all the predicates in a single operation. This is useful when the predicate register state must be saved and restored across subroutine calls.

ASID – SPR #53

ASID is an acronym for Address Space Identifier. It is used in conjunction with the TLB unit during virtual address translations. The ASID value in this register must match the ASID in the TLB entry in order for that translation to be considered valid. The ASID is typically associated with a process.

Operating Modes

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

A subset of instructions is limited to kernel mode.

Segmentation

Overview

Segmentation is a low overhead means of memory protection and virtualization. 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 in order 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]	

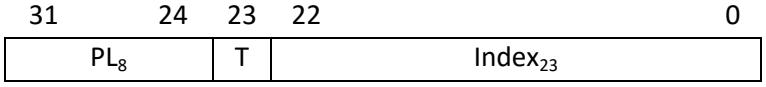
Selecting a segment register

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. 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

Non-Segmented Code Area

The address range defined as 64'hFxxxxxxxxxxxxxxx (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:

	255 244	243 192	191 128	127 64	63 0
w0	ACR_{16}	$\sim_{48}$	$\sim_{64}$	$Limit_{64}$	$Base_{64}$
w1	ACR_{16}	$\sim_{48}$	$\sim_{64}$	$Limit_{64}$	$Base_{64}$
…					

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 in order 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	$\sim_{48}$	ACR_{16}
n+2	$\sim_{64}$	
n+1	$Limit_{63..0}$	
n	$Base_{63..0}$	

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

15			12	11	10	9	8	7 0
P	~	~	1/S	Ex	C/Stk	W/R	A	DPL_8

P: 1 = segment present, 0 = segment not present

S: 0 = system descriptor, 1 = memory descriptor

EX: 1 = executable, 0 = data

Code Segment	Data Segment
C: 1= conforming	Stk: 1=stack segment
R: 1 = readable	W: 1=writeable
A: 1= accessed	

DPL_8 = descriptor privilege level

Typical Values for ACR

9A00 – executable, readable code segment, privilege level zero

9200 – read/writeable data segment, privilege level zero

9600 – 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:

$\sim_{48}$	ACR_{16}
$\sim_{64}$	
$Upper\ Limit_{34..3}$	$Lower\ Limit_{34..3}$
$Base_{63..0}$	

There is no difference between a stack segment descriptor and a memory segment descriptor except in the way that the segment limit field is used. (Bit 10 of the ACR for the data descriptor is set). For a stack segment, when the descriptor is loaded, the limit field is split in two in order to provide both an upper and lower bounds to the stack. 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 35 address bits (32GB). 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 bit12 of the access rights character set to zero. There are potentially sixteen different system descriptor types.

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

15			12	11 10 9 8	7 0
P	~	~	0	$Type_4$	DPL_8

$Type_4$	Gate	
0	unused	
2	LDT descriptor	
4	Call gate	
5	Task Gate	
6	Interrupt Gate	
7	Trap gate	

LDT Descriptor

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

n+3	$\sim_{48}$	ACR_{16}
n+2	$\sim_{64}$	
n+1	$\sim_{41}$	$Size_{22..0}$
n	$Base_{63..0}$	

Call Gate Descriptor

$\sim_{48}$		ACR_{16}
$\sim_{64}$		
$\sim_{27}$	N_5	$Selector_{31..0}$
$Base_{63..0}$		

Segment Load Exception

Moving a value to a selector register (a move to SPR #32 to 38,40) triggers a segment load exception in order 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 64'hFFFFFFFFFFC0000. 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

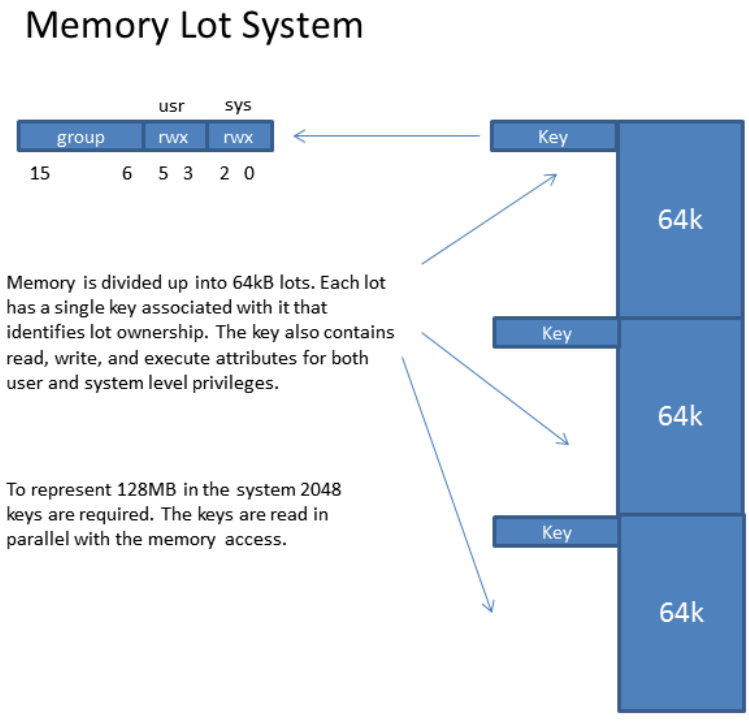
Num		Long name	Comment
0	ZS	zero (NULL) segment	by convention contains zero
1	DS	data segment	by convention – default for loads/stores
2	ES	extra segment	by convention
3	FS		
4	GS		
5	HS		
6	SS	Stack segment	default for stack load/stores
7	CS	Code segment	always used for code addressing

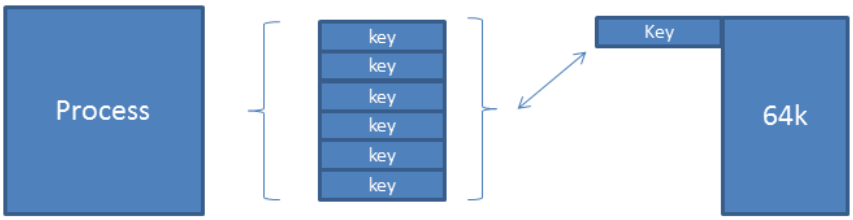
Memory Lot Protection System

The memory lot protection system offers an alternative means to incorporate memory protection into a system.

A key feature required to increase system reliability and robustness is memory protection. Memory should be protected against inadvertent access by the process that doesn't own a particular piece of memory. The system used here provides memory protection, but not address virtualization.

Memory is organized into lots which are 64kB in size. Memory is protected using a system of keys associated with each lot of memory. The key associated with a memory lot contains the lot owner's group, and read / write / execute indicators.





Each process has a collection of six ten bit keys associated with it. If any one of the process's keys matches the key associated with a memory lot, then the process has access to that lot. The process has multiple keys available in order to allow for shared memory regions.

Comparison of all the keys to the memory lot's key is done in parallel during the memory access.

TLB – The Translation Lookaside Buffer

Overview

The TLB (translation look-aside buffer) offers a second means of address virtualization and memory protection in addition to segmentation. A TLB works by caching address mappings between a real physical address and a virtual address used by software. The TLB is managed by software triggered when a TLB miss occurs. 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

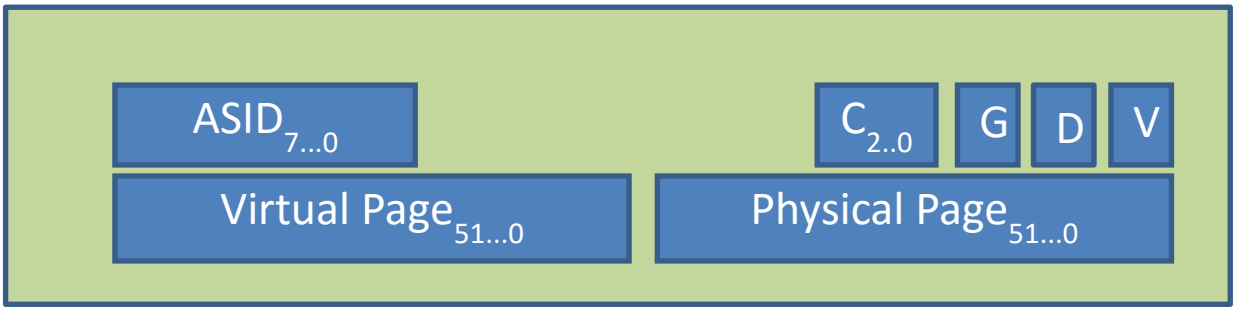
The core uses a 64 entry TLB (translation look-aside buffer) in order to support virtual memory. The TLB supports variable page sizes from 4kB to 1MB. The TLB is organized as an eight-way eight-set cache.

Updating the TLB

The TLB is updated by first placing values into the TLB holding registers using the TLB instruction, then issuing a TLB write command using the TLB command instruction.

Address translations will not take place until the TLB is enabled. An enable TLB command must be issued using the TLB command instruction.

TLB Entries:



G = Global

The global bit marks the TLB entry as a global address translation where the ASID field is not used to match addresses.

ASID = address space identifier

The ASID field in the TLB entry must match the processor's current ASID value in order for the translation to be considered valid, unless the G bit is set. If the G bit is set in the TLB entry, then the ASID field is ignored during the address comparison.

C = cachability bits

If the cachability bits are set to 001_b then the page is uncached, otherwise the page is cached.

D = dirty bit

The dirty bit is set by hardware when a write occurs to the virtual memory page identified by the TLB entry.

V = valid bit

This bit must be set in order for the address translation to be considered valid. The entire TLB may be invalidated using the invalidate all command.

TLB Registers

TLBWired (#0h)

This register limits random updates to the TLB to a subset of the available number of ways. TLB ways below the value specified in the Wired register will not be updated randomly. Setting this register provides a means to create fixed translation settings. For instance if the wired register is set to two, the sixteen fixed entries will be available.

TLBIndex (#1h)

This register contains the entry number of the TLB entry to be read from or written to.

TLBRandom (#2h)

This register contains a random three bit value used to update a random TLB entry during a TLB write operation.

TLBPageSize (#3h)

The TLBPageSize register controls which address bits are significant during a TLB lookup.

N	Page Size	
0	4KiB	
1	16kiB	
2	64kiB	
3	256kiB	
4	1MiB	

TLBPhysPage (#5h)

The TLBPhysPage register is a holding register that contains the page number for an associated virtual address. This register is transferred to or from the TLB by TLB instructions.

63	0
Physical Page Number	

TLBVirtPage (#4h)

The TLBVirtPage register is a holding register that contains the page number for an associated physical address. This register is transferred to or from the TLB by TLB instructions.

63	0
Virtual Page Number	

TLBASID (#7h)

The TLBASID register is a holding register that contains the address space identifier (ASID) , valid, dirty, global, and cachability bits associated with a TLB entry. This register is transferred to or from the TLB by TLB instructions.

63 16	15 8	6 4	2	1	0
-----	ASID	C	G	D	V

Memory Operations:

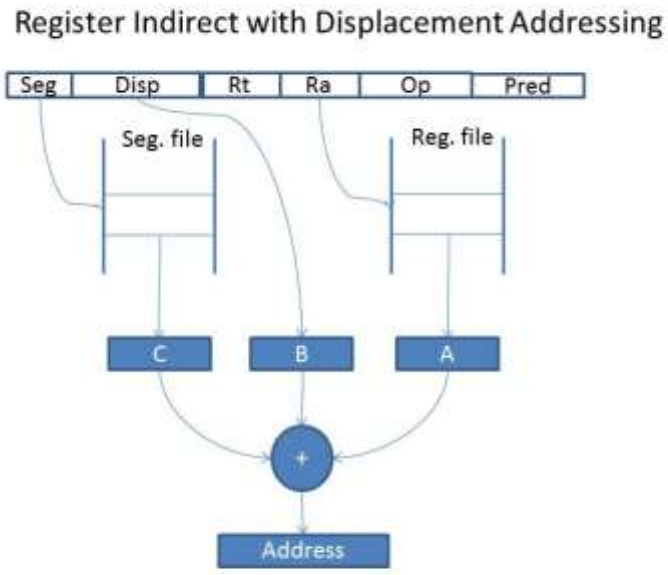
Basic Operations

Basic memory operations include loads, stores, pushes, pops and string operates. There is also a memory indirect jump instruction. Other than those operations there are no other instructions that access memory. Note that return addresses are not pushed onto the stack automatically.

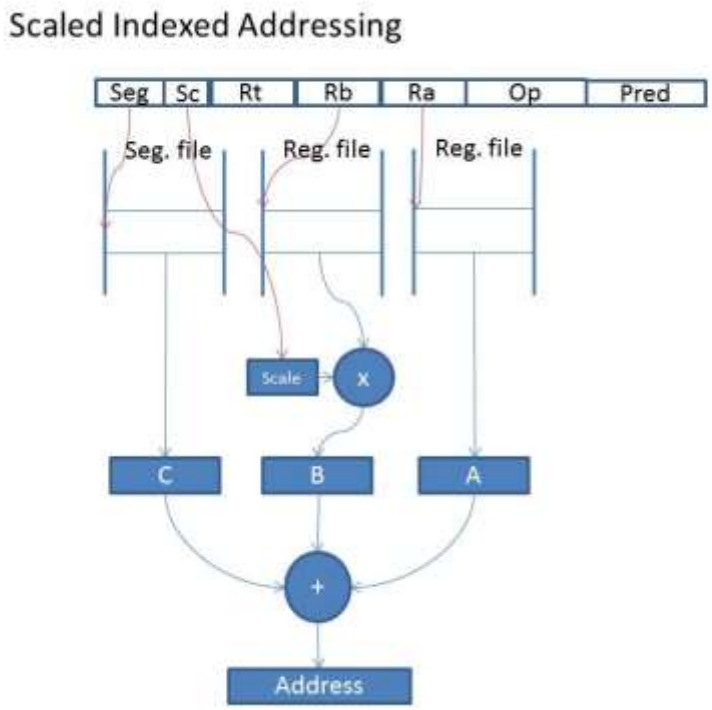
Memory Addressing Modes

The core supports both register indirect with displacement and scaled indexed addressing. Indexed addressing is supported only with the general purpose register load store operations.

Register indirect addressing looks up both a register value and a segment register value from the register files and adds a displacement from the instruction to form an address.



Scaled Indexed addressing looks up two register file values and a segment register value, multiplies the second register value by a scaling factor, then adds all three values to form an address.



Pre-fetching data

The load instructions may be used to pre-fetch data by specifying a load into register R0. If R0 is used as the load target register then the load operation will not cause any exception.

Bypassing the Data Cache

There are several load instructions that bypass the data-cache when loading – see the load volatile (LVx) instructions. These instructions are useful for I/O operations or for when it is better if the data cache is not loaded for performance reasons.

The volatile load instructions only offer sign extension and not zero extension. To zero extend data loaded by a volatile load operation follow it with one of the zero extension (ZXx) instructions.

Push Operations

The core supports a data push to stack and data pop operation. The data push operation both decrements the stack pointer and stores data to the stack. Argument pushing is commonly used in high-level languages. Subroutine arguments pushed to the stack in high-level languages are usually popped off the stack simply by adding to the stack pointer.

An additional push operation includes pushing an effective address to the stack.

Load Speculation

The core may load data speculatively in advance of its use provided there is no address overlap with a preceding store instruction.

Store Issuing

Stores will only be issued if there are no instructions that can exception before the store in the instruction queue. Since many instructions do not cause any exceptions this happens fairly often.

Atomic Operations

The core has a single atomic memory operation which is the CAS (compare-and-swap) instruction.

Address Reservation

The address reservation instructions rely on the external memory system to support address reservation. There are only two instruction (LVWAR, SWCR) associated with address reservation. The load instruction creates an address reservation and the store instruction clears it. In a multi-core system the reservation may be created or cleared by another processing core.

Synchronization Operations

The core includes memory data barrier and memory instruction barrier instructions to allow data to be synchronized during program runs.

Exceptions

Precision

Thor's exceptions are precise. They are processed in order at the location of the exception. For instance if a divide by zero exception occurred then the exception return address is the address of the divide by zero instruction. Instructions after the divide by zero are not committed to the machine state (the results are dropped).

Nesting

Software exceptions are allowed to nest up to 255 levels. The nesting level is tracked by the core and when it is non-zero the core is in kernel mode. When an exception occurs the nesting level increases, when a return from exception is performed (RTE or RTD) the nesting level decreases. From a software standpoint this allows exceptions to occur in an exception handler. For instance it may be desirable for a debug exception to occur in the handler.

Hardware interrupts do not track the nesting level. The core does not allow nested hardware interrupts. When a hardware interrupt occurs the core is switched to kernel mode.

Vectors

The processor vectors to $FFFFFFFFFFFC0000 on a reset. All other vectoring is done through a vector table. The vector table allows for 256 entries. The vector table base address is established by code address register C12. During an external IRQ the processor looks at a vector number bus to determine the vector to use for the IRQ. This vector number may be hard-coded in which case all IRQ's will be vectored to the same location. The address vectored to is the sum of C12 and an offset supplied in the instruction multiplied by sixteen. The contents of C12 are undefined at reset; this register must be loaded before interrupts can be processed. Note that segmentation is temporarily disabled during exception processing to allow the vector to be accessed.

Vector table:

Vector Number	Usage / Description		
0	BREAK instruction vector		
1	SLEEP vector (branch to self)		
2	FMTK Task reschedule		
3	FMTK Time slice scheduler		
4	FMTK System call		

…				
190	Save register state			
191	Restore register state			
192	Spurious interrupt			
193	IRQ level 1	1000 Hz interrupt		
194	IRQ level 2	100 Hz interrupt		
…	Other IRQ levels			
207	IRQ level 15	keyboard interrupt		
…				
239	check failed			
240	overflow (integer)			
241	divide by zero (integer)			
242	floating point			
243	debug			
244	segmentation			
245	privilege violation			
246	segment load			
247	segment not present			
248	DTLBMiss			
249	ITLB Miss			
250	Unimplemented instruction			
251	Bus error – data load / store			
252	Bus error – instruction fetch			
253	reserved			
254	NMI interrupt vector			
255	- reserved			
256	Load ZS			
257	Load DS			
258	Load ES			
259	Load FS			
260	Load GS			
261	Load HS			
262	Load SS			
263	- reserved			
264	Load LDT			
265	Far return			
266	Far subroutine call			

Mnemonics

DBG	debug
DBE	data bus error
TLB	TLB miss
LMT	segment limit

Hardware Interrupts

Interrupt Enable Delay

The core features a delay before interrupts are enabled by the CLI instruction or by an RTI instruction restoring the interrupt mask status. The default value for this delay is five clock cycles. See the IMCD core parameter description. The idea behind the delay is that processing may continue on non-interrupt code even if an IRQ line is stuck active due to a hardware problem.

Hardware Ports

Thor uses a WISHBONE bus to communicate with the outside world.

	I/O	Width	WB		
corenum	I	32		core number – this number is used to identify the core and is reflected in the cpuid register. Meant to be a hardcoded constant.	
rst_i	I	1	WB	reset signal	
clk_i	I	1	WB	clock	
clk2x_i	I	1		two times clock input (drives register file)	
clk_o	O	1		output (gated) clock	
km	O	1		kernel mode indicator	
nmi_i	I	1		non-maskable interrupt input	
irq_i	I	1		maskable interrupt input	
vec_i	I	8		interrupt vector	
bte_o	O	2	WB	burst type extension	
cti_o	O	3	WB	cycle type indicator	
bl_o	O	5		burst length output	
lock_o	O	1	WB	bus lock	
resv_o	O	1		reserve address	
resv_i	I	1		address reservation status in	
cres_o	O	1		clear address reservation	
cyc_o	O	1	WB	cycle is valid	
stb_o	O	1	WB	data transfer is taking place	
ack_i	I	1	WB	data transfer acknowledge	
err_i	I	1	WB	bus error occurred input	
we_o	O	1	WB	write enable	
sel_o	O	8	WB	byte lane selects	
adr_o	O	64	WB	address output	
dat_i	I	64	WB	data input bus	
dat_o	O	64	WB	data output bus	

WB = see the WISHBONE spec rev B3

Reset

On reset the core begins fetching and executing instruction at address $FFFFFFFFFFFC0000. The code segment is set to $0000 and the program counter is set to $FFFFFFFFFFFC0000. Note that the last 4k bytes of memory are unreachable to the processing core due to limitations in the segment boundary checking. The last 4k bytes should not be used to store instructions or data.

On power-up or reset interrupts are disabled automatically, In order to enable interrupts the RTI instruction must be executed. An RTI automatically enables interrupts. Note that the interrupt mask must also be cleared with the CLI instruction to allow maskable interrupts to occur.

After reset or NMI the core begins processing at a half the maximum clock rate. The STP instruction must be issued to get the processer running at full speed.

Clock Cycle Counts

The core has a minimum CPI of 0.5 clocks per instruction running trivial sample code. Many instructions can be done in pairs provided there are no dependencies between the instructions. Due to the out of order execution ability of the core the latency of longer running instructions may be hidden. The core may be busy working on up to four instructions at once: two ALU or an ALU and memory op, a floating point op and a branch instruction.

Core Parameters

DBW	32	The parameter controls the width of data processed by the core. Set to 64 for 64 bit processing. This parameter should be either 64 or 32. If the width is set to 32 bit then double precision floating point operations are unavailable. Also only eight predicate registers are available.
ABW	32	This parameter controls the width of the external address bus. This value should be the same as or less than the data bus width (DBW).
ALU1BIG	0	This parameter controls whether or not ALU1 supports all instructions or only a subset of instructions. The default is to support only the most common instructions. (0 = limited, 1 = all) in order to reduce the size of the core. Limiting the number of instructions supported may impact performance of the core because it may not be possible to issue two instructions in the same cycle.
IMCD	3Eh	This parameter controls how many clock cycles interrupts are disabled before an interrupt enable takes effect. It is a shift counter value. Valid values are 3E (5 clocks),3C(four clocks),38,30,and 20 (1 clock).

Configuration Defines

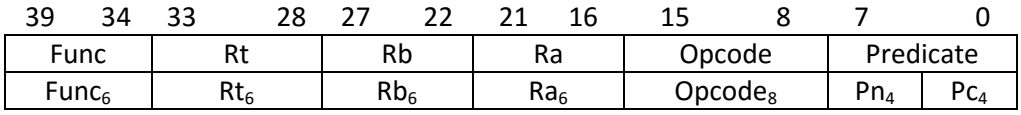
Definition	Default	Comment
SEGMENTATION	1	Causes the core to include segmentation. If segmentation is not desired then this can be commented out to produce an unsegmented core.
SEGLIMITS	1	Causes the core to include logic for segmentation limit checks, but only if SEGMENTATION is defined. Commenting out this definition will remove the segment limit checks and exceptions.
STRINGOPS	1	Causes the core to include memory string operations.
DEBUG_LOGIC	1	Causes the core to include logic to support the debug registers. Once the application is working well it may be desirable to reduce the size of the core by removing the debug registers and associated logic.
TRAP_ILLEGALOPS	1	Causes the core to include logic to trap for illegal operations during runtime. Once the target application is working well it may be desirable to remove the illegal instruction trapping.
FLOATING_POINT	1	Causes the core to include floating point operations. Comment out to reduce the size of the core.
BITFIELDOPS	1	Causes the core to include bit-field operations if defined.
PRIVCHKS	1	Causes the core to include privilege level checking logic. This may be commented out if not needed.
PCHIST	1	Causes the core to include PC history capture logic.

Instruction Formats

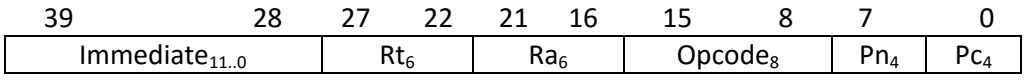
Instructions vary in length from one to eight bytes. There are only a few of single byte instructions consisting of only a predicate. Some of the more common formats are shown below.

All instruction sequences begin with a predicate byte that determines the conditions under which the instruction executes. With the exception of special predicate values, the next field in the instruction is always the opcode byte. All instructions may be preceded by an extended constant value.

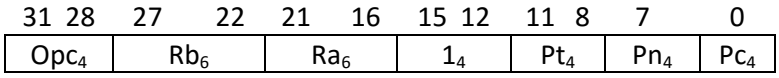
RR - Register-Register



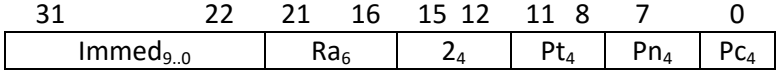
RI - Register-Immediate



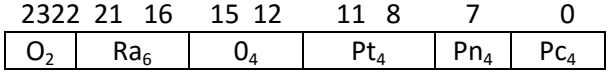
CMP Register-Register Compare



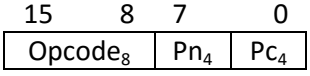
CMPI Register-Immediate Compare



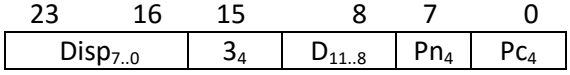
TST - Register Test Compare



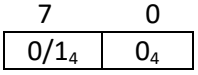
CTRL/RTx- Control



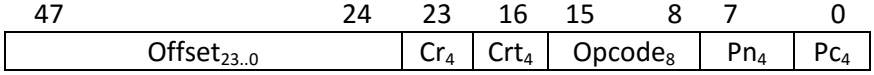
BR - Relative Branch



BRK/NOP



JSR - Jump To Subroutine



Instruction Set Summary

A number of rarely used instructions may only execute on ALU #0. These instructions are identified in the text.

Illegal Instructions

By default the core traps all illegal instructions through vector #250. The logic to trap illegal instructions may be removed by commenting out the TRAP_ILLEGALOPS definition.

Branch Instructions

The core has only a single relative branch instruction which branches relative to the address of the next instruction. This single branch instruction may be used to implement branching on multiple complex conditions when combined with a predicate. The branch instruction supports a 12 bit displacement field.

Branch Speculation

Branches are performed speculatively in the fetch stage of the core according to branch predictor output. Branches use a (2, 2) co-relating branch predictor with a 256 entry branch history table. Both global and local branch histories are maintained.

Loops

There is a loop instruction and corresponding loop count register to support counted loops. The loop instruction is predicted as always taken and does not consume room in the branch history table. Like a branch instruction a loop instruction takes place at the fetch stage of the core. The loop instruction supports only and eight bit displacement field which may not be extended.

Subroutine Call / Return

Program counter relative jumps and calls may be achieved using the program counter as the index register in jump instructions. The jump instruction directly supports up to 24 bit addressing. A shorter jump instruction is available that supports 16 bit addressing. The addressing capabilities of the jump instruction may be increased by applying an immediate prefix to the instruction. It is envisioned that the 16/24 bit jump addressing is sufficient for most cases when combined with usage of the code segment.

Comparison Operations

Comparison operations include CMP and TST (compare to zero). Comparison operations set a predicate register to the result status of the comparison.

Arithmetic Operations

Mnemonic	
ADD	addition
ADDU	unsigned addition
SUB	subtraction
SUBU	unsigned subtraction
MUL	multiplication
MULU	
DIV	division
DIVU	
NEG	negative
ABS	absolute value
MIN	minimum value
MAX	maximum value

Bitwise Operations

Bitwise operations include ‘and’, ‘or’ and exclusive ‘or’ along with their inverted versions.

Mnemonic	Has Immediate Form	
AND	Y	
OR	Y	
EOR	Y	
NAND	N	
NOR	N	
ENOR	N	
ANDC	N	
ORC	N	
COM	N	invert bits

Logical Operations

The core includes the logical ‘not’ (NOT) operation. The NOT operation reduces the value to a one or zero result.

Detailed Instruction Set

2ADDU - Register-Register

Description:

Multiply Ra by two and add Rb and place the sum in the target register. This instruction will never cause an overflow exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$08h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra * 2 + Rb

Exceptions: none

2ADDUI - Register-Immediate

Description:

Multiply Ra by two and add immediate and place the sum in the target register. This instruction will never cause an overflow exception.

Instruction Format:

39 28	27 22	21 16	15 8	7 0	
$Immediate_{11..0}$	Rt_6	Ra_6	$6Bh_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra * 2 + immediate

Exceptions: none

4ADDU - Register-Register

Description:

Multiply Ra by four and add Rb and place the sum in the target register. This instruction will never cause an exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$09h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra * 4 + Rb

Exceptions: none

4ADDUI - Register-Immediate

Description:

Multiply Ra by four and add immediate and place the sum in the target register. This instruction will never cause an exception.

Instruction Format:

39 28	27 22	21 16	15 8	7 0
$Immed_{11..0}$	Rt_6	Ra_6	$6Ch_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra * 4 + immediate

Exceptions: none

8ADDU - Register-Register

Description:

Multiply Ra by eight and add Rb and place the sum in the target register. This instruction will never cause an exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$0Ah_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra * 8 + Rb

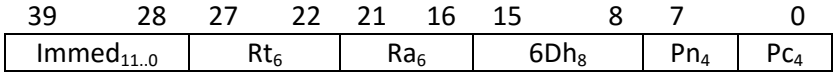
Exceptions: none

8ADDUI - Register-Immediate

Description:

Multiply Ra by eight and add immediate and place the sum in the target register. This instruction will never cause an exception.

Instruction Format:



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra * 8 + immediate

Exceptions: none

16ADDU - Register-Register

Description:

Multiply Ra by sixteen and add Rb and place the sum in the target register. This instruction will never cause an exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
$0Bh_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra * 16 + Rb

Exceptions: none

16ADDUI - Register-Immediate

Description:

Multiply Ra by sixteen and add immediate and place the sum in the target register. This instruction will never cause an exception.

Instruction Format:

39 28	27 22	21 16	15 8	7 0
$Immed_{11..0}$	Rt_6	Ra_6	$6Eh_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra * 16 + immediate

Exceptions: none

ABS – Absolute Value Register

Description:

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

Instruction Format:

31 28	27 22	21 16	15 8	7 0	
3_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

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

Exceptions: none

ADD - Register-Register

Description:

Add two registers and place the sum in the target register. This instruction may cause an overflow exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$00h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Rb

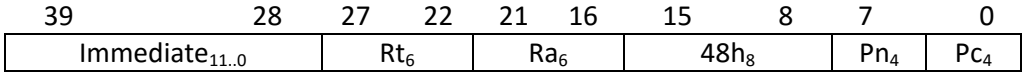
Exceptions: integer overflow

ADDI - Register-Immediate

Description:

Add a register and immediate value and place the sum in the target register. This instruction may cause an overflow exception.

Instruction Format:



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + immediate

Exceptions: integer overflow

ADDU - Register-Register

Description:

Add registers Ra and Rb and place the result into register Rt. This instruction will never cause any exceptions.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$04h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Rb

Exceptions: none

ADDUI - Register-Immediate

Description:

Add a register and immediate value and place the sum in the target register. This instruction will never cause an exception.

Instruction Format:

39 — 28	27 — 22	21 — 16	15 — 8	7 — 0
$Immediate_{11..0}$	Rt_6	Ra_6	$4Ch_8$	Pn_4	Pc_4

31 — 22	21 — 16	15 — 8	7 — 0
$Immediate_{9..0}$	Rt_6	$47h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Immediate

Exceptions: none

AND - Register-Register

Description:

Bitwise and's two registers and places the result in a target register.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$00h_6$	Rt_6	Rb_6	Ra_6	$50h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra & Rb

Exceptions: none

ANDC – And with Compliment

Description:

Bitwise and's a register Ra with the compliment of register Rb and places the result in a target register. There is no immediate form for this instruction.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$06h_6$	Rt_6	Rb_6	Ra_6	$50h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra & ~Rb

Exceptions: none

ANDI - Register-Immediate

Description:

Bitwise and's register and an immediate value and places the result in a target register.

Instruction Format:

39 28	27 22	21 16	15 8	7 0	
$Immediate_{11..0}$	Rt_6	Ra_6	$53h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra & immediate

Exceptions: none

BCDADD - Register-Register

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:

39 34	33 28	27 22	21 16	15 8	7 0	
$00h_6$	Rt_6	Rb_6	Ra_6	$F5h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra + Rb

Exceptions: none

BCDMUL - Register-Register

Description:

Multiplies 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 16 bit BCD value.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$02h_6$	Rt_6	Rb_6	Ra_6	$F5h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 Only

Operation:

Rt = Ra * Rb

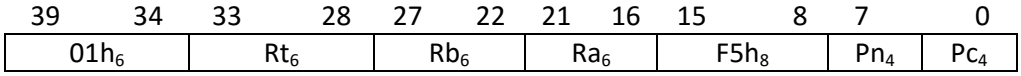
Exceptions: none

BCDSUB - Register-Register

Description:

Subtracts 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:



Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra - Rb

Exceptions: none

BFCHG – Bit-field Change

Description:

Inverts the bit-field in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

Instruction Format:

4744	43 40	39 34	33 28	27 22	21 16	15 8	7 0	
$\sim_4$	3_4	me_6	mb_6	Rt_6	Ra_6	AAh_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions: none

BFCLR – Bit-field Clear

Description:

Sets the bits to zero of the bit-field in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

Instruction Format:

47	44	43 40	39 34	33 28	27 22	21 16	15 8	7 0
~$_4$		2_4	me_6	mb_6	Rt_6	Ra_6	AAh_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions: none

BFEXT – Bit-field Extract

Description:

Extracts a bit-field from register Ra located between the mask begin (mb) and mask end (me) bits and places the sign extended result into the target register.

Instruction Format:

47	44	43 40	39 34	33 28	27 22	21 16	15 8	7 0
~$_4$	5_4	me_6	mb_6	Rt_6	Ra_6	AAh_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions: none

BFEXTU – Bit-field Extract Unsigned

Description:

Extracts a bit-field from register Ra located between the mask begin (mb) and mask end (me) bits and places the zero extended result into the target register.

Instruction Format:

47	44	43 40	39 34	33 28	27 22	21 16	15 8	7 0
	~$_4$	4_4	me_6	mb_6	Rt_6	Ra_6	AAh_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions: none

BFINS – Bit-field Insert

Description:

Inserts a bit-field into the target register located between the mask begin (mb) and mask end (me) bits from the low order bits of Ra.

Instruction Format:

4744	43 40	39 34	33 28	27 22	21 16	15 8	7 0	
$\sim_4$	0_4	me_6	mb_6	Rt_6	Ra_6	AAh_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions: none

BFINSI – Bit-field Insert Immediate

Description:

Inserts a bit-field into the target register located between the mask begin (mb) and mask end (me) bits from the bits specified in the instruction.

Instruction Format:

4744	43 40	39 34	33 28	27 22	21 16	15 8	7 0	
$\sim_4$	6_4	me_6	mb_6	Rt_6	Imm_6	AAh_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions: none

BFSET – Bit-field Set

Description:

Sets the bits to one of the bit-field in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

Instruction Format:

47	44	43 40	39 34	33 28	27 22	21 16	15 8	7 0
~$_4$		1_4	me_6	mb_6	Rt_6	Ra_6	AAh_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions: none

BITI – Test bits Register-Immediate

Description:

Logically and's register and an immediate value and places the result in a predicate register. If the result of the 'and' operation is zero the predicate register's zero flag is set, otherwise it is cleared. If the result is negative the predicate's less than flag is set, otherwise it is cleared.

Instruction Format:

39 28	26	25 22	21 16	15 8	7 0
$Immediate_{11..0}$	$\sim_2$	Pt_4	Ra_6	$46h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Pt = flag results(Ra & immediate)

Predicate Results:

Predicate flag	Setting
eq	set if result is zero
lt	set if result is negative
ltu	set if result is odd (bit 0 is set)

Exceptions: none

BR - Relative Branch

Description:

A branch is made relative to the address of the next instruction.

- The twelve bit displacement field cannot be extended with an immediate constant prefix. Branches are executed immediately in the instruction fetch stage of the processor before it is known if there is a prefix present.

Instruction Format:

23 16	15 8	7 0		
$Disp_{7..0}$	$3h_4$	$D_{11..8}$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's / Branch

Operation:

PC <= PC + displacement

Exceptions: none

BRK –Break

Description:

This instruction contains only a predicate byte. The Break exception is executed. The core will be switched to kernel mode.

Instruction Format:

7	0
0_4	0_4

BSR - Branch to Subroutine

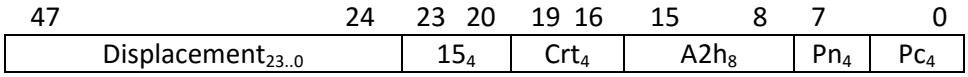
Description:

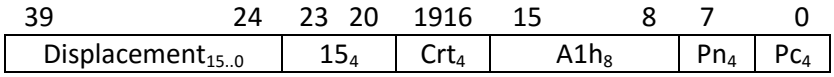
This is an alternate mnemonic for the JSR instruction. A jump is made to the sum of the sign extended displacement supplied in the displacement field of the instruction and the specified code address register Cr.

The subroutine return address is stored in a code address register specified in the Crt field of the instruction.

Typically code address register #1 is used to store the return address.

Instruction Formats:





Clock Cycles: 1

Exceptions: none

CACHE – Cache Command

Description:

This instruction issues a command to the cache.

Instruction Format:

31 26	2524	23 22	21 16	15 8	7 0	
$Func_6$	$~_2$	$~_2$	Ra_6	$9Fh_8$	Pn_4	Pc_4

Operation:

Commands:

$Func_6$	
0	Invalidate entire instruction cache
1	Invalidate instruction cache line (address in Ra)
32	Invalidate entire data cache
33	Invalidate data cache line (address in Ra)

CAS – Compare and Swap

Description:

If the contents of the addressed memory cell is equal to the contents of Rb then a sixty-four bit value is stored to memory from the source register Rc. The original contents of the memory cell are loaded into register Rt. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be word aligned. If the operation was successful then Rt and Rb will be the same value. The compare and swap operation is an atomic operation; the bus is locked during the load and potential store operation. This operation assumes that the addressed memory location is part of the volatile region of memory and bypasses the data cache.

The stack pointer cannot be used as the target register.

Instruction Format:

47 40	39 34	33 28	27 22	21 16	15 8	7 0
$Displacement_{7..0}$	Rt_6	Rc_6	Rb_6	Ra_6	$97h_8$	Pn_4	Pc_4

Operation:

```
Rt = memory [Ra + displacement]
if memory[Ra + displacement] = Rb
        memory[Ra + displacement] = Rc
```

Assembler:

CAS Rt,Rb,Rc,offset[Ra]

CHKI - Register-Immediate

Description:

Check register against bounds. The comparisons are signed comparisons. There is no unsigned form of this instruction. If the register is inside the bounds, the target predicate register equals flag is set, otherwise it is cleared.

Instruction Format:

4744	4340	39 28	27 22	21 16	15 8	7 0
$\sim_4$	Pt_4	$Immediate_{11..0}$	Rb_6	Ra_6	$45h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

```
if (Rb < Ra or Rb >= Immediate)
        Pt.eq = 0
else
        Pt.eq = 1
```

Exceptions: none

CHKX- Register-Register

Description:

Check register against bounds. The comparisons are signed comparisons. There is no unsigned form of this instruction. This instruction may cause a check exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$14h_6$	Rc_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

```
if (Rc < Ra || Rc >= Rb)
   check exception
```

Exceptions: bounds check

CHKXI - Register-Immediate

Description:

Check register against bounds. The comparisons are signed comparisons. There is no unsigned form of this instruction. This instruction may cause a check exception.

Instruction Format:

39 28	27 22	21 16	15 8	7 0
$Immediate_{11..0}$	Rb_6	Ra_6	$5Dh_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

```
if (Rb < Ra or Rb >= Immediate)
        check exception
```

Exceptions: bounds check

CLI – Clear Interrupt Mask

Description:

This instruction is used to enable interrupts. This instruction is available only while operating in kernel mode.

Instruction Format:

15 8	7 4	3 0
FAh_8	Pn_4	Pc_4

Clock Cycles: 1

Operation:

im = 0

Exceptions: privilege violation

CMP Register-Register Compare

Description:

The register compare instruction compares two registers and sets the flags in the target predict register as a result.

Instruction Format:

3128	27 22	21 16	15 12	11 8	7	0
0_4	Rb_6	Ra_6	1_4	Pt_4	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

```
if signed Ra < signed Rb
        P.lt = true
else
        P.lt = false
if unsigned Ra < unsigned Rb
        P.ltu = true
else
        P.ltu = false
if Ra = Rb
        P.eq = true
else
        P.eq = false
```

Exceptions: none

CMPI Register-Immediate Compare

Description:

The register immediate compare instruction compares a register to an immediate value and sets the flags in the target predict register as a result. Both a signed and unsigned comparison take place at the same time.

Instruction Format:

31 22	21 16	15 12	11 8	7	0
$Immed_{10}$	Ra_6	2_4	Pt_4	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

```
if signed Ra < signed immediate
        P.lt = true
else
        P.lt = false
if unsigned Ra < unsigned immediate
        P.ltu = true
else
        P.ltu = false
if Ra = immediate
        P.eq = true
else
        P.eq = false
```

CNTLO- Count Leading Ones

Description:

This instruction counts the number of leading ones in a register and places the result in a target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
6_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Exceptions: none

CNTLZ- Count Leading Zeros

Description:

This instruction counts the number of leading zeros in a register and places the result in a target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0	
5_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Exceptions: none

CNTPOP- Population Count

Description:

This instruction counts the number of one bits in a register and places the result in a target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0	
7_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Exceptions: none

COM – Bitwise Compliment

Description:

This instruction performs a bitwise compliment on a register and places the result in a target register. If bit is a one then the bit is replaced with is zero otherwise it is replaced with a one.

Instruction Format:

31 28	27 22	21 16	15 8	7 0	
B_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = ~ Ra

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:

31 28	27 22	21 16	15 8	7 0	
0_4	Rt_6	Ra_6	$41h_8$	Pn_4	Pc_4

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 “Thor”
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)

DIV - Register-Register Divide

Description:

Performs a signed division of two registers and places the quotient in the target register. This instruction may cause an overflow or divide by zero exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$03h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 65

Execution Units: ALU #0 only

Operation:

Rt = Ra / Rb

Exceptions: divide by zero

DIVI - Register-Immediate Divide

Description:

Performs a signed divide of a register and an immediate value and places the result in a target register. This instruction may cause an overflow or divide by zero exception.

Instruction Format:

39 28	27 22	21 16	15 8	7 0
$Immediate_{11..0}$	Rt_6	Ra_6	$4Bh_8$	Pn_4	Pc_4

Clock Cycles: 65

Execution Units: ALU #0 only

Operation:

Rt = Ra / immediate

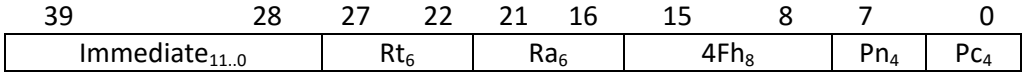
Exceptions: divide by zero

DIVIU – Unsigned Register-Immediate Divide

Description:

Performs an unsigned divide of a register and an immediate value and places the result in a target register. This instruction will not cause an overflow or divide by zero exception.

Instruction Format:



Clock Cycles: 65

Execution Units: ALU #0 only

Operation:

Rt = Ra / immediate

Exceptions: none

DIVU – Unsigned Register-Register Divide

Description:

Performs an unsigned division of two registers and places the quotient in the target register. This instruction not cause an overflow or divide by zero exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$07h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 65

Execution Units: ALU #0 only

Operation:

Rt = Ra / Rb

Exceptions: none

ENOR - Register-Register

Description:

Bitwise exclusive or register with register and place inverted result in target register.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$05h_6$	Rt_6	Rb_6	Ra_6	$50h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = ~(Ra ^ Rb)

Exceptions: none

EOR - Register-Register

Description:

Bitwise exclusive or register with register and place result in target register.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
$02h_6$	Rt_6	Rb_6	Ra_6	$50h_8$	Pn_4 Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra ^ Rb

Exceptions: none

EORI - Register-Immediate

Description:

Bitwise exclusive or register with immediate and place result in target register.

Instruction Format:

39 28	27 22	21 16	15 8	7 0	
$Immediate_{11..0}$	Rt_6	Ra_6	$55h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra ^ immediate

Exceptions: none

INC – Increment Memory

Description:

Memory is incremented by the amount specified in the instruction. The memory address is the sum of the sign extended displacement and register Ra. The amount is between -128 and +127. Note that the increment is not an atomic memory operation. The bus is not locked during the increment to allow cached data to be incremented. For atomic memory operations see the CAS instruction.

Instruction Format:

47 40	3937	36 28	27	22	21 16	15 8	7 0
Amt_8	Sg_3	$Displacement_{8..0}$	O_3	Sz_3	Ra_6	$C7h_8$	Pn_4 Pc_4

Execution Units: All Memory

Operation:

(mem[Ra+offset]) = (mem[Ra+offset]) + amt

IMM64,IMM56,IMM48,IMM40,IMM32,IMM24,IMM16

Immediate Extensions

The immediate extension predicates are used to extend the immediate constant of the following instruction. The extensions may add from one to seven bytes more to the constant. Most, but not all instructions can accept a predicated immediate.

Immediate	Predicate	
$Immediate_{63..8}$	8_4	0_4
$Immediate_{55..8}$	7_4	0_4
$Immediate_{47..8}$	6_4	0_4
$Immediate_{39..8}$	5_4	0_4
$Immediate_{31..8}$	4_4	0_4
$Immediate_{23..8}$	3_4	0_4
$Immediate_{15..8}$	2_4	0_4

Clock Cycles: 1

Execution Units: Enqueue

Exceptions: none

INT –Interrupt

Description:

This instruction calls a system function located as the sum of the zero extended offset times 16 plus code address register 12. The return address is stored in the IPC register (code address register #14).

The offset field of this instruction cannot be extended.

Note that this instruction is automatically invoked for hardware interrupt processing. This instruction would not normally be used by software and is not supported by the assembler. 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.

Instruction Format:

31 24	23 20	19 16	15 8	7 0
$Offset_{7..0}$	Ch_4	Eh_4	$A6h_8$	Pn_4 Pc_4

JCI, JCIX – Jump Character Indirect

Description:

This instruction performs a memory indirect jump to a target address. The target address is formed from the sixteen bit data located in memory combined with the upper address bits of the program counter. The return address is stored in a code address register specified in the instruction.

For the register indirect with displacement form register Ra is scaled by two before use in forming an address. For the scaled indexed form of the instruction register Rb is scaled by two before use in forming an address.

This instruction is used when a table of code addresses is present in memory. The code address table may be compressed by eliminating the upper address bits from the table. The table should be entirely located within the same upper address bits as the address of the instruction.

Instruction Formats:

3937	36 28	2726	25 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	1_2	Ct_4	Ra_6	$8Dh_8$	Pn_4 Pc_4

39 37	36	3534	3332	31 28	27 22	21 16	15 8	7 0
Seg_3	~	$~_2$	1_2	Ct_4	Rb_6	Ra_6	$B7h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's

Operation:

Ct = pc
$pc = pc_{[63:16]}$,$mem[Ra*2 + displacement]_{16}$

Exceptions: DBG TLB LMT DBE

Usage:

The following example replaces about five lines of code with just a single line of code by making use of the JCI instruction. The code is part of a dispatch routine for a BIOS call.

Using JCI

```
              jci    c1,cs:VideoBIOS_FuncTable[r6]
```

Without using JCI

```
              ldi     r10,#VideoBIOS_FuncTable
              lcu     r10,cs:[r10+r6*2]
              ori     r10,r10,#VideoBIOSCall & 0xFFFFFFFFFFFF0000    ; recover
high order bits
              mtspr  c2,r10
              jsr     [c2]
```

JHI, JHIX – Jump Half-word Indirect

Description:

This instruction performs a memory indirect jump to a target address. The target address is formed from the thirty-two bit data located in memory combined with the upper address bits of the program counter. The return address is stored in a code address register specified in the instruction.

For the register indirect with displacement form register Ra is scaled by four before use in forming an address. For the scaled indexed form of the instruction register Rb is scaled by four before use in forming an address.

This instruction is used when a table of code addresses is present in memory. The code address table may be compressed by eliminating the upper address bits from the table. The table should be entirely located within the same upper address bits as the address of the instruction.

Instruction Formats:

3937	36 28	2726	25 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	2_2	Ct_4	Ra_6	$8Dh_8$	Pn_4 Pc_4

39 37	36	3534	3332	31 28	27 22	21 16	15 8	7 0
Seg_3	~	$~_2$	2_2	Ct_4	Rb_6	Ra_6	$B7h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's

Operation:

Ct = pc
pc = $pc_{[63:32]}$,$mem[Ra*4 + displacement]_{32}$

Exceptions: DBG TLB LMT DBE

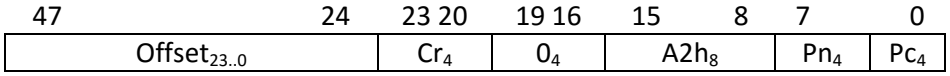
JMP - Jump To Address

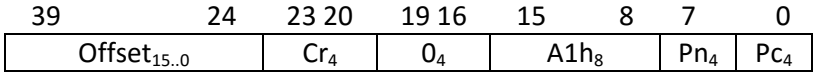
Description:

This is an alternate mnemonic for the JSR instruction.

A jump is made to the sum of the zero extended offset supplied in the offset field of the instruction and the specified code address register Cr. The JMP instruction may be used with an immediate predicate constant in order to extend the address range of the jump.

Instruction Formats:





Clock Cycles: 1

Execution Units: All ALU's

Operation:

pc = $Cr_{[n]}$ + offset

Exceptions: none

JSF – Jump to Far Subroutine

Description:

Invoke far subroutine call trap (vector #265). Typically this will cause the current program counter and code segment to be saved on stack. This instruction is typically followed by inline parameters which specify the target segment and offset. The trap routine will typically look up the desired segment from one of the descriptor tables.

Stack Frame:

Stack Offset	Item Stored
n+5	segment acr
n+4	reserved
n+3	segment limit
n+2	segment base
n+1	code segment selector
n	program counter (return address)

Instruction Formats:

15 8	7 0	
FEh_8	Pn_4	Pc_4

Execution Units: All ALU's / Branch

Operation:

Exceptions: DBE, DBG, TLB, LMT

Example:

```
jsf
align 8
dw targetOffset
dh targetSelector
```

JSR - Jump To Subroutine

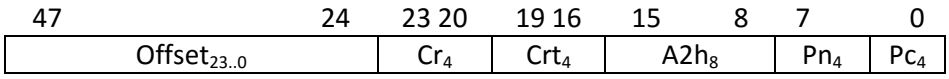
Description:

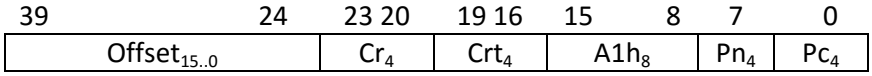
A jump is made to the sum of the zero extended offset supplied in the offset field of the instruction and the specified code address register Cr. The JSR instruction may be used with an immediate predicate constant in order to extend the address range of the jump.

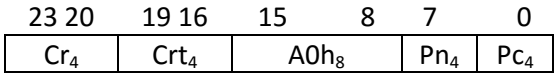
The subroutine return address is stored in a code address register specified in the Crt field of the instruction. Typically code address register #1 is used.

An immediate constant prefix applied to this instruction overrides offset bits 8 to 23 and acts like an eight bit immediate constant extension used by other instructions.

Instruction Formats:







Clock Cycles: 1

Execution Units: All ALU's

Operation:

$Cr_{[t]}$ = pc
pc = $Cr_{[n]}$ + offset

Exceptions: none

JWI, JWIX – Jump Word Indirect

Description:

This instruction performs a memory indirect jump to a target address. The target address is loaded from data located in memory. The return address is stored in a code address register specified in the instruction.

For the register indirect with displacement form register Ra is scaled by eight before use in forming an address. For the scaled indexed form of the instruction register Rb is scaled by eight before use in forming an address.

Instruction Formats:

3937	36 28	2726	25 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	3_2	Ct_4	Ra_6	$8Dh_8$	Pn_4 Pc_4

39 37	36	3534	3332	31 28	27 22	21 16	15 8	7 0
Seg_3	~	$~_2$	3_2	Ct_4	Rb_6	Ra_6	$B7h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's

Operation:

Ct = pc
pc = $mem[Ra*8 + displacement]_{64}$

Exceptions: DBG TLB LMT DBE

LB – 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.

This instruction will load data from the cache and cause a cache load operation if the data isn't in the cache. To bypass the cache use the LVB instruction.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$80h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra+offset])

Exceptions: DBE, DBG, LMT, TLB

LBU – 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.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$81h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend (mem[Ra+offset])

Exceptions: DBE, DBG, LMT, TLB

LBUX – Load Byte Unsigned Indexed

Description:

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

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$B1h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = mem[Ra+Rb]

Exceptions: DBE, DBG, LMT, TLB

LBX – Load Byte Indexed

Description:

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

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$B0h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra+Rb])

Exceptions: DBE, DBG, LMT, TLB

LC – Load Character

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 displacement and register Ra. The memory address must be character aligned.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0	
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$82h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra + displacement])

Exceptions: DBE, DBG, LMT, TLB

LCL – Load Cache Line

Description:

The cache line is loaded from memory into the cache (instruction or data). The memory address is the sum of the sign extended offset and register Ra.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Tgt_6	Ra_6	$8Fh_8$	Pn_4	Pc_4

Execution Units: Cache / Memory

Operation:

Rt = sign extend (mem[Ra+offset])

Target:

Tgt_6	Cache
0	instruction cache
1	data cache

LCU – Load Character 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 displacement and register Ra. The memory address must be character aligned.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0	
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$83h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend (mem[Ra + displacement])

Exceptions: DBE, DBG, LMT, TLB

LCUX – Load Character Unsigned Indexed

Description:

A sixteen bit value is loaded from memory, zero extended and placed in the target register Rt. The memory address is the sum of register Ra and scaled register Rb. The memory address must be character aligned.

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$B3h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = mem[Ra + Rb * scale]

Exceptions: DBE, DBG, LMT, TLB

LCX – Load Character Indexed

Description:

A sixteen bit value is loaded from memory, sign extended and placed in the target register Rt. The memory address is the sum of register Ra and scaled register Rb. The memory address must be character aligned.

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$B2h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = mem[Ra + Rb * scale]

Exceptions: DBE, DBG, LMT, TLB

LDI - Load-Immediate

Description:

This instruction loads a sign extended immediate constant into a register. The immediate constant may be extended by using an immediate prefix instruction.

Instruction Format:

31 22	21 16	15 8	7 0
$Immediate_{9..0}$	Rt_6	$6Fh_8$	Pn_4 Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = immediate

LDIS - Load-Immediate Special

Description:

This instruction loads a sign extended immediate constant into a special purpose register. The immediate constant may be extended by using an immediate prefix instruction. Typical usage is to initialize a code address register with a target address.

Instruction Format:

31 22	21 16	15 8	7 0
$Immediate_{9..0}$	Spr_6	$9Dh_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Spr = immediate

LEA – Load Effective Address

Description:

The virtual memory address is placed in the target register. The memory address is the sum of the sign extended displacement and register Ra. This is an alternate form of the ADDUI instruction where the operand is specified in a memory operand format.

Instruction Format:

39 28	27 22	21 16	15 8	7 0
$Disp_{11..0}$	Rt_6	Ra_6	$4Ch_8$	Pn_4	Pc_4

Operation:

Rt = Ra + Displacement

Execution Units: All ALU's

LEAX – Load Effective Address Indexed

Description:

A virtual memory address is computed and placed in the target register. The address is the sum of register Ra and scaled register Rb. This mnemonic is an alternate form of the ADDU, _2ADDU,_4ADDU or _8ADDU instruction. The assembler will emit the add instruction according to the scale specified for Rb.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
Fn_6	Rt_6	Ra_6	Rb_6	$40h_8$	Pn_4 Pc_4

Clock Cycles: 1

Execution Units: All ALU's / Memory

Operation:

Rt = Ra + Rb * scale

Exceptions: none

LH – Load Half-Word

Description:

A thirty-two bit value is loaded from memory and sign extended, then placed in the target register Rt. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be half-word aligned.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$84h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra + displacement])

Exceptions: DBE, DBG, LMT, TLB

LHU – Load Half-word Unsigned

Description:

A thirty-two bit value is loaded from memory and zero extended, then placed in the target register Rt. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be half-word aligned.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$85h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend (mem[Ra + displacement])

Exceptions: DBE, DBG, LMT, TLB

LHUX – Load Half-word 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 register Rb. The memory address must be half-word aligned.

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$B5h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = mem[Ra+Rb*scale]

Exceptions: DBE, DBG, LMT, TLB

LHX – Load Half-word Indexed

Description:

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

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$B4h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra + Rb * scale])

Exceptions: DBE, DBG, LMT, TLB

LLA – Load Linear Address

Description:

A linear memory address is computed and placed in the target register. The linear address is the sum of the sign extended displacement, register Ra, and the specified segment register.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$6Ah_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's / Memory

Operation:

Rt = Ra + displacement + segment

Exceptions: none

LLAX – Load Linear Address Indexed

Description:

A linear memory address is computed and placed in the target register. The linear address is the sum of register Ra, scaled register Rb, and the specified segment register.

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rt_6	Rb_6	Ra_6	$B8h_8$	Pn_4 Pc_4

Clock Cycles: 1

Execution Units: All ALU's / Memory

Operation:

Rt = Ra + Rb + segment

Exceptions: none

LOOP – Loop Branch

Description:

A branch is made relative to the address of the next instruction if the loop count register is non-zero. The loop count register is decremented by this instruction. The predicate condition must also be met. The loop branch is predicted as always taken and does not consume room in the branch predication tables. The displacement constant may not be extended as the loop takes place in the instruction fetch stage of the core.

Instruction Format:

23 16	15 8	7 0	
$Disp_{7..0}$	$A4h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's / Branch

Operation:

```
If LC <> 0
        PC <= PC + displacement
LC = LC - 1
```

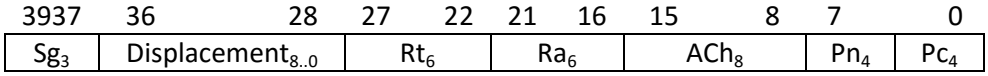
LVB – Load Volatile Byte

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 displacement and register Ra. This instruction bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices. This instruction may also be used when it is known that the data is better not cached.

There is no indexed or unsigned form for this instruction. The value loaded may be zero extended rather than sign extended by following it with the ZXB instruction.

Instruction Format:



Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra+offset])

Exceptions: DBE, DBG, LMT, TLB

LVC – Load Volatile Character

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 bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices.

There is no indexed or unsigned form for this instruction.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	ADh_8	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra+offset])

Exceptions: DBE, DBG, LMT, TLB

LVH – Load Volatile Half-word

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 bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices.

There is no indexed or unsigned form for this instruction.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	AEh_8	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra+offset])

Exceptions: DBE, DBG, LMT, TLB

LVW – Load Volatile Word

Description:

A sixty-four 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 bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices.

There is no indexed or unsigned form for this instruction.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	AFh_8	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra + displacement])

Exceptions: DBE, DBG, LMT, TLB

LVWAR – Load Volatile Word and Reserve

Description:

A sixty-four 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. Additionally the reserve signal is activated on the bus to tell the memory system to place an address reservation. This instruction bypasses the data cache. Use this instruction to load data from volatile memory regions such as I/O devices. The primary purpose of this instruction is to setup semaphores. See also the SWCR, CAS instructions.

There is no indexed form for this instruction.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$8Bh_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (mem[Ra + displacement]); reserve = 1

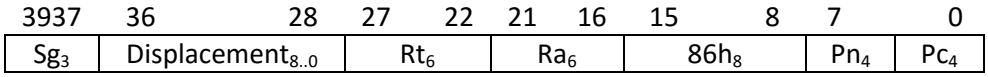
Exceptions: DBE, DBG, LMT, TLB

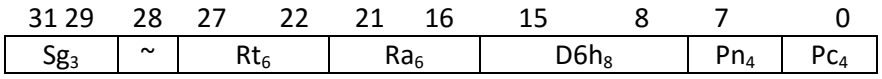
LW – Load Word

Description:

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

Instruction Format:





Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Exceptions:

If the target register is R0 then this instruction will not cause an exception. Otherwise an exception may be caused by a data-bus error signal input or a TLB miss.

Operation:

Rt = mem[Ra + displacement]

Exceptions: DBE, DBG, LMT, TLB

LWS – Load Word Special

Description:

A sixty-four bit value is loaded from memory and placed in the special purpose register. The memory address is the sum of the sign extended offset and register Ra. The memory address must be word aligned.

There is no indexed form for this instruction.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$8Eh_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Spr = mem[Ra + displacement]

Exceptions: DBE, DBG, LMT, TLB

LWX – Load Word Indexed

Description:

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

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$B6h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = mem[Ra + Rb*scale]

Exceptions: DBE, DBG, LMT, TLB

MAX - Register-Register

Description:

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

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$11h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

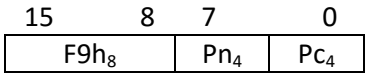
```
IF Ra < Rb
        Rt = Rb
else
        Rt = Ra
```

MEMDB – Memory Data Barrier

Description:

All memory accesses before the MEMDB command are completed before any memory accesses after the data barrier are started. Note that this instruction has an effect even if the predicate is false; this does not affect the correct operation of the program, only performance is affected.

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. Note that this instruction has an effect even if the predicate is false; this does not affect the correct operation of the program, only performance is affected.

Instruction Format:

15 8	7	0
$F8h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: Memory

MFSPR – Special Register-Register

Description:

This instruction moves from a special purpose register into a general purpose one.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
$\sim_4$	Rt_6	Spr_6	$A8h_8$	Pn_4 Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

$Rt = Spr_{[n]}$

Special Purpose Registers

Reg #	R/W			
00-15	RW	PRED	specific predicate register #0 to 15	
16-31	RW	CREGS	Code address register array (C0 to C15)	
32-39	RW	SREGS	Segment base register array (zs,ds,es,fs,gs,hs,ss,cs)	
40-47			- reserved for segmentation	
48	R	MID	Machine ID	
49	R	FEAT	Features	
50	R	TICK	Tick count	
51	RW	LC	Loop Counter	
52	RW	PREGS	Predicate register array	
53	RW	ASID	address space identifier	
59	RW	EXC	exception cause register	
60	W	BIR	Breakout index register	
61	RW		Breakout register - additional spr's	
63			reserved	

Additional Spr's are available by setting the breakout index register to an Sor index value, then accessing the Spr through the breakout register.

MIN - Register-Register

Description:

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

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$10h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

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

MLO – Mystery Logical Operation

Description:

The MLO instruction performs an operation that is determined at run-time as opposed to compile time. The operation to be performed is one of the register-register logical operations. Register Rc contains the function code for the operation. Registers Ra and Rb are the operands to the instruction. The result is placed in register Rt.

The MLO instruction is provided to help avoid writing self-modifying code for performance reasons.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
Rt_6	Rc_6	Rb_6	Ra_6	$51h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra op(Rc) Rb

MODI –Register-Immediate Modulus

Description:

Performs a signed divide of a register and an immediate value and places the remainder in a target register. This instruction may cause an overflow or divide by zero exception.

Instruction Format:

39 28	27 22	21 16	15 8	7 0
$Immediate_{11..0}$	Rt_6	Ra_6	$5Bh_8$	Pn_4	Pc_4

Clock Cycles: 65

Execution Units: ALU #0 only

Operation:

Rt = Ra % immediate

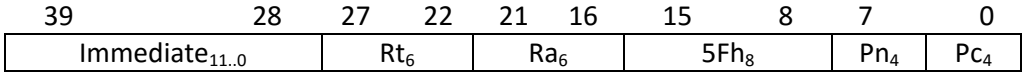
Exceptions: DBZ OFL

MODUI – Unsigned Register-Immediate Modulus

Description:

Performs an unsigned divide of a register and an immediate value and places the remainder in a target register. This instruction will not cause an overflow or divide by zero exception.

Instruction Format:



Clock Cycles: 65

Execution Units: ALU #0 only

Operation:

Rt = Ra % immediate

Exceptions: none

MOV - Register-Register

Description:

This instruction moves one general purpose register to another. This instruction is shorter and uses one less register port than using the OR instruction to move between registers.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
0_4	Rt_6	Ra_6	$A7_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVS – Move Special Register- Special Register

Description:

This instruction moves one special purpose register to another. The primary purpose of this instruction is to allow transfers directly between code address or segment registers.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
~$_4$	Sprt$_6$	Spr$_6$	AB$_8$	Pn$_4$	Pc$_4$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Sprt = Spra

MTSPR –Register-Special Register

Description:

Move a general purpose register into a special purpose register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
$\sim_4$	Spr_6	Ra_6	$A9h_8$	Pn_4 Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

$Spr_{[n]} = Ra$

MUL - Register-Register Multiply

Description:

Performs a signed multiply of two registers and places the product in the target register. This instruction may cause an overflow exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$02h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 5

Execution Units: ALU #0 Only

Operation:

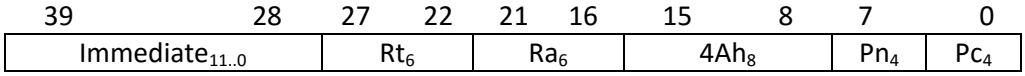
Rt = Ra * Rb

MULI - Register-Immediate Multiply

Description:

Performs a signed multiply of a register and an immediate value and places the result in a target register. This instruction may cause an overflow exception.

Instruction Format:



Clock Cycles: 5

Execution Units: ALU #0 only

Operation:

Rt = Ra * immediate

MULU – Unsigned Register-Register Multiply

Description:

Performs an unsigned multiply of two registers and places the product in the target register. This instruction will never cause an overflow exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$06h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 5

Execution Units: ALU #0 only

Operation:

Rt = Ra * Rb

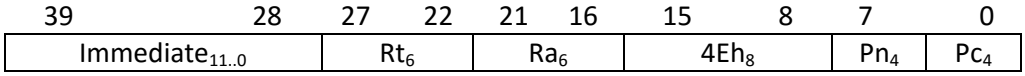
Exceptions: none

MULUI – Unsigned Register-Immediate Multiply

Description:

Performs an unsigned multiply of a register and an immediate value and places the result in a target register. This instruction will never cause an overflow exception.

Instruction Format:



Clock Cycles: 5

Execution Units: ALU #0 only

Operation:

Rt = Ra * immediate

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:

39 34	33 28	27 22	21 16	15 8	7 0
Rt_6	Rc_6	Rb_6	Ra_6	$72h_8$	Pn_4	Pc_4

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 - Register-Register

Description:

Bitwise and's two registers inverts the result and places the result in a target register.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$03h_6$	Rt_6	Rb_6	Ra_6	$50h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = ~(Ra & Rb)

Exceptions: none

NEG - Negate Register

Description:

This instruction negates a register and places the result in a target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
1_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = - Ra

NOP – No Operation

Description:

This instruction contains only a predicate byte. This is a single byte no-operation code. It can be used to align code addresses or as a fill byte.

The NOP operation is not queued by the processing core and is not present in the pipeline.

Instruction Format:

7	0
1_4	0_4

Two byte Format:

18	8	7	0
F_4	1_4	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: None

Operation:

<none>

Exceptions: none

NOR - Register-Register

Description:

Bitwise inclusively or two registers and place inverted result in the target register.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$04h_6$	Rt_6	Rb_6	Ra_6	$50h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = ~(Ra | Rb)

Exceptions: none

NOT – Logical Not

Description:

This instruction performs a logical NOT on a register and places the result in a target register. If the value in a register is non-zero then the result is zero. If the value in the register is zero then the result is one. This instruction results in either a one or zero being placed in the target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0	
2_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = ! Ra

Exceptions: none

OR - Register-Register

Description:

Bitwise inclusively or two registers and place the result in the target register.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$01h_6$	Rt_6	Rb_6	Ra_6	$50h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra | Rb

Exceptions: none

ORC – Or with Compliment

Description:

Bitwise inclusively or register Ra and the compliment of register Rb and place the result in the target register.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
$07h_6$	Rt_6	Rb_6	Ra_6	$50h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra | ~Rb

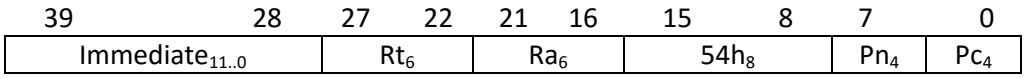
Exceptions: none

ORI - Register-Immediate

Description:

Bitwise inclusively or register with immediate and place the result in the target register.

Instruction Format:



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra | imm

Exceptions: none

PAND – Predicate And

Description:

Bitwise and's the specified predicate register bits and places the result in a target bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
0_6	Bt_6	Bb_6	Ba_6	$42h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Pr[Rt] = Pr[Ra] & Pr[Rb]

Exceptions: none

PANDC – Predicate And Compliment

Description:

Bitwise and's the specified predicate register bits and places the result in a target bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
6_6	Bt_6	Bb_6	Ba_6	$42h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Pr[Rt] = Pr[Ra] & ~Pr[Rb]

Exceptions: none

PEOR – Predicate Exclusive Or

Description:

Bitwise exclusive or's the specified predicate register bits and places the result in a target bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
2_6	Bt_6	Bb_6	Ba_6	$42h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Pr[Rt] = Pr[Ra] ^ Pr[Rb]

Exceptions: none

PENOR – Predicate Exclusive Nor

Description:

Bitwise exclusive or's the specified predicate register bits and places the inverted result in a target bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
5_6	Bt_6	Bb_6	Ba_6	$42h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Pr[Rt] = ~(Pr[Ra] ^ Pr[Rb])

Exceptions: none

PNAND – Predicate Nand

Description:

Bitwise and's the specified predicate register bits and places the inverted result in a target bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
3_6	Bt_6	Bb_6	Ba_6	$42h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Pr[Rt] = ~(Pr[Ra] & Pr[Rb])

Exceptions: none

POR – Predicate Or

Description:

Bitwise or's the specified predicate register bits and places the result in a target bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
1_6	Bt_6	Bb_6	Ba_6	$42h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Pr[Rt] = Pr[Ra] | Pr[Rb]

Exceptions: none

PORC – Predicate Or Compliment

Description:

Bitwise or's the specified predicate register bits and places the result in a target bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
7_6	Bt_6	Bb_6	Ba_6	$42h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Pr[Rt] = Pr[Ra] | ~Pr[Rb]

Exceptions: none

PNOR – Predicate Nor

Description:

Bitwise or's the specified predicate register bits and places the inverted result in a target bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
4_6	Bt_6	Bb_6	Ba_6	$42h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Pr[Rt] = ~(Pr[Ra] | Pr[Rb])

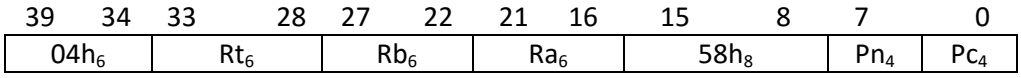
Exceptions: none

ROL – Rotate Left

Description:

Rotate register Ra left by Rb bits and place the result into register Rt. The most significant bit is shifted into the least significant bit. The rotation takes place modulo 64 of the value in register Rb (only the lower six bits of the register are used).

Instruction Format:



Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra << Rb

Exceptions: none

ROLI – Rotate Left by Immediate

Description:

Rotate register Ra left by n bits and place the result into register Rt. The most significant bit is shifted into the least significant bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$14h_6$	Rt_6	Imm_6	Ra_6	$58h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra << #n

Exceptions: none

ROR – Rotate Right

Description:

Rotate register Ra right by Rb bits and place the result into register Rt. The least significant bit is shifted into the most significant bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$05h_6$	Rt_6	Rb_6	Ra_6	$58h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra >> Rb

Exceptions: none

RORI – Rotate Right by Immediate

Description:

Rotate register Ra right by n bits and place the result into register Rt. The least significant bit is shifted into the most significant bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$15h_6$	Rt_6	Imm_6	Ra_6	$58h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra >> #n

Exceptions: none

RTD – Return from Debug Exception Routine

Description:

The program counter is loaded with the value contained in code address register #11 which is the DPC register. This instruction may cause the core to transition back to applications mode. It is only available while the core is in kernel mode.

Instruction Format:

15 8	7 4	3 0
FCh_8	Pn_4	Pc_4

Operation:

PC = $Cr_{[11]}$
if (StatusEXL > 0) StatusEXL = StatusEXL - 1

Exceptions: PRIV

RTE – Return from Exception Routine

Description:

The program counter is loaded with the value contained in code address register #13 which is the EPC register. This instruction may cause the core to transition back to applications mode. It is only available while the core is in kernel mode.

Instruction Format:

15 8	7	0
$F3h_8$	Pn_4	Pc_4

Operation:

PC = $Cr_{[13]}$
if (StatusEXL > 0) StatusEXL = StatusEXL - 1

Exceptions: PRIV

RTF – Return from Far Subroutine

Description:

Invoke the return from far subroutine trap (vector #264). This will cause the program counter and code segment to be reloaded with values prior to calling the subroutine. The information for the code segment and program counter is typically stored on the stack by a far subroutine call.

Stack Frame:

Stack Offset	Item Stored
n+5	segment acr
n+4	reserved
n+3	segment limit
n+2	segment base
n+1	code segment selector
n	program counter (return address)

Instruction Formats:

15 8	7 0	
FDh_8	Pn_4	Pc_4

Execution Units: All ALU's / Branch

Operation:

The exact operation performed depends on the system software.

CS = stack[n+1]
PC = stack[n]
SP = SP + 48

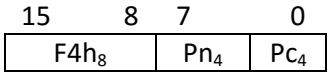
Exceptions: DBE, DBG, TLB, LMT

RTI – Return from Interrupt Routine

Description:

The program counter is loaded with the value contained in code address register #14 which is the IPC register. Additionally the interrupt mask is cleared to enable interrupts. This instruction will cause the core to transition back to applications mode. It is only available while the core is in kernel mode.

Instruction Format:



Operation:

pc = $Cr_{[14]}$
Flags = FlagsBackup
Flags.im = 0
StatusHWI = 0

Exceptions: PRIV

RTS – Return from Subroutine

Description:

The program counter is loaded with the value contained in the specified code address register plus a zero extended four bit immediate constant. The constant may not be extended. This allows the return instruction to return a few bytes past the usual return address. This is used to allow static parameters to be passed to the subroutine in inline code.

Note that the JMP instruction may also be used to return from a subroutine. Similarly this instruction may also be used to perform a jump to one of the first sixteen addresses relative to a code address register.

Instruction Formats:

C1 Implied

15 8	7 0	
$F2h_8$	Pn_4	Pc_4

Return past calling address

23 20	19 16	15 8	7 0	
Cr_4	Im_4	$A3h_8$	Pn_4	Pc_4

Execution Units: All ALU's / Branch

Operation:

$PC = Cr_{[N]} + Imm_4$

C1 Implied Operation:

$PC = Cr_{[1]}$

Exceptions: none

SB – 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:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$90h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

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

Exceptions: DBE, DBG, TLB, LMT

SBX – Store Byte Indexed

Description:

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

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$C0h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

memory[Ra+Rb] = Rb

Exceptions: DBE, DBG, TLB, LMT

SC – Store Character

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. The memory address must be character aligned.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rb_6	Ra_6	$91h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

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

Exceptions: DBE, DBG, TLB, LMT

SCX – Store Character Indexed

Description:

A sixteen bit value is stored to memory from the source register Rc. The memory address is the sum of register Ra and scaled register Rb. The memory address must be character aligned.

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$C1h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

memory[Ra+Rb*scale] = Rb

Exceptions: DBE, DBG, TLB, LMT

SEI – Set Interrupt Mask

Description:

The interrupt mask is set, disabling maskable interrupts. This instruction is available only in kernel mode.

Instruction Format:

15 8	7	0
FBh_8	Pn_4	Pc_4

Clock Cycles: 1

Operation:

im = 1

Exceptions: none

SH – Store Half-word

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. The memory address must be half-word aligned.

Instruction Format:

39 37	36 28	27 22	21 16	15 8	7 0
Seg_3	$Displacement_{8..0}$	Rb_6	Ra_6	$92h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

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

Exceptions: DBE, DBG, TLB, LMT

SHL – Shift Left

Description:

Shift register Ra left by Rb bits and place result into register Rt. A zero is shifted into the least significant bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$00h_6$	Rt_6	Rb_6	Ra_6	$58h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra << Rb

Exceptions: none

SHLI – Shift Left by Immediate

Description:

Shift register Ra left by n bits and place result into register Rt. A zero is shifted into the least significant bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$10h_6$	Rt_6	Imm_6	Ra_6	$58h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra << #n

Exceptions: none

SHLU – Shift Left Unsigned

Description:

Shift register Ra left by Rb bits and place the result into register Rt. A zero is shifted into the least significant bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$02h_6$	Rt_6	Rb_6	Ra_6	$58h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra << Rb

Exceptions: none

SHLUI – Shift Left Unsigned by Immediate

Description:

Shift register Ra left by n bits and place the result into register Rt. A zero is shifted into the least significant bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$12h_6$	Rt_6	Imm_6	Ra_6	$58h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra << #n

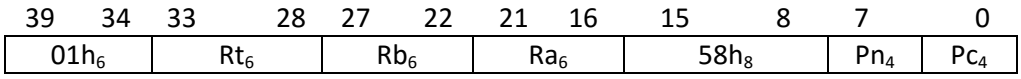
Exceptions: none

SHR – Shift Right

Description:

Shift register Ra right by Rb bits and place result in register Rt. The sign bit is preserved.

Instruction Format:



Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra >> Rb

Exceptions: none

SHRI – Shift Right by Immediate

Description:

Shift register Ra right by n bits and place result into register Rt. The sign bit is preserved.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$11h_6$	Rt_6	Imm_6	Ra_6	$58h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra >> #n

Exceptions: none

SHRU – Shift Right Unsigned

Description:

Shift register Ra right by register Rb bits. A zero is shifted into the sign bit.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
$03h_6$	Rt_6	Rb_6	Ra_6	$58h_8$	Pn_4 Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra >> Rb

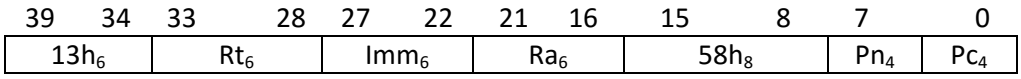
Exceptions: none

SHRUI – Shift Right Unsigned by Immediate

Description:

Shift register Ra right by n bits and place result into register Rt. A zero is shifted into the sign bit.

Instruction Format:



Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra >> #n

Exceptions: none

SHX – Store Half-word Indexed

Description:

A thirty-two bit value is stored to memory from the source register Rb. The memory address is the sum of register Ra and scaled register Rb. The memory address must be half-word aligned.

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$C2h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

memory[Ra+Rb] = Rb

Exceptions: DBE, DBG, TLB, LMT

STCMP – String Compare

Description:

This instruction compares data from the memory location addressed by Ra plus Rc to the memory location addressed by Rb plus Rc until the loop counter LC reaches zero or until a mismatch occurs. Rc acts as an index and increments or decrements by the size of the operation as the move takes place. This instruction is interruptible. The data must be in the same segment and appropriately aligned. The loop counter is set to zero when a mismatch occurs. The index of the mismatch is contained in register Rc.

Instruction Format:

37 34	33 28	27 22	21 16	15 8	7 0	
Sg_3	O_3	Rc_6	Rb_6	Ra_6	$9Ah_8$	Pn_4	Pc_4

O_3	Assembler Mnemonic	
0	STCMP.BI	bytes incrementing
1	STCMP.CI	characters incrementing
2	STCMP.HI	half-word incrementing
3	STCMP.WI	words incrementing
4	STCMP.BD	bytes decrementing
5	STCMP.CD	characters decrementing
6	STCMP.HD	half-word decrementing
7	STCMP.WD	word decrementing

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        mem[Rb + Rc] = mem[Ra + Rc]
        Rc = Rc +/- amt
        LC = LC – 1
```

STFND – String Find

Description:

This instruction compares data from the memory location addressed by Ra plus Rc to the data in register Rb until the loop counter LC reaches zero or until a match occurs. Rc acts as an index and increments or decrements by the size of the operation as the move takes place. This instruction is interruptible. The data must be appropriately aligned. The loop counter is set to zero when a match occurs. The index of the match is contained in register Rc.

Instruction Format:

37 34	33 28	27 22	21 16	15 8	7 0	
Sg_3	O_3	Rc_6	Rb_6	Ra_6	$9Bh_8$	Pn_4	Pc_4

O_3	Assembler Mnemonic	
0	STFND.BI	bytes incrementing
1	SFND.CI	characters incrementing
2	STFND.HI	half-word incrementing
3	STFND.WI	words incrementing
4	STFND.BD	bytes decrementing
5	STFND.CD	characters decrementing
6	STFND.HD	half-word decrementing
7	STFND.WD	word decrementing

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        if  (mem[Ra + Rc] = Rb)
           stop
        Rc = Rc +/- amt
        LC = LC – 1
```

STI – Store Immediate

Description:

A six bit value is zero extended to sixty-four bits and stored to memory. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be word aligned.

Instruction Format:

39 37	36 28	27 22	21 16	15 8	7 0
Seg_3	$Displacement_{8..0}$	Imm_6	Ra_6	$96h_8$	Pn_4	Pc_4

Execution Units: All ALU's / Memory

Operation:

memory[Ra + displacement] = zero extend ($Imm_{[5..0]}$)

STIX – Store Immediate Indexed

Description:

A ten bit value is zero extended to sixty-four bits and stored to memory. The memory address is the sum of register Ra and scaled register Rb. The memory address must be word aligned.

Instruction Format:

39 36	35 34	33 28	27 22	21 16	15 8	7 0
$Imm_{9..6}$	Sc_2	$Imm_{5..0}$	Rb_6	Ra_6	$C6h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

memory[Ra + Rb * scale] = zero extend ($Imm_{[9..0]}$)

STMOV – String Move

Description:

This instruction moves a data from the memory location addressed by Ra plus Rc to the memory location addressed by Rb plus Rc until the loop counter LC reaches zero. Rc acts as an index and increments or decrements by the size of the operation as the move takes place. This instruction is interruptible. The data moved must be in the same segment and appropriately aligned.

Instruction Format:

37 – 34	33 – 28	27 – 22	21 – 16	15 – 8	7 – 0
Sg_3	O_3	Rc_6	Rb_6	Ra_6	$99h_8$	Pn_4	Pc_4

O_3	Assembler Mnemonic	
0	STMOV.BI	move bytes incrementing
1	STMOV.CI	move characters incrementing
2	STMOV.HI	move half-word incrementing
3	STMOV.WI	move words incrementing
4	STMOV.BD	move bytes decrementing
5	STMOV.CD	move characters decrementing
6	STMOV.HD	move half-word decrementing
7	STMOV.WD	move word decrementing

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        mem[Rb + Rc] = mem[Ra + Rc]
        Rc = Rc +/- amt
        LC = LC – 1
```

STP – Stop / Slow Down

Description:

This instruction controls the core clock rate which affects power consumption. The immediate constant is loaded into a shift register that controls the frequency of clock pulses seen by the processor. Setting the constant to FFFFh provides the maximum clock rate. Setting the constant to zero stops the clock completely. With the clock stopped completely the core must be reset or an NMI interrupt must occur before the core will continue processing. After reset or NMI the core begins processing at a half the maximum clock rate.

Instruction Format:

31 16	15 8	7 0	
$Immediate_{16}$	$F6h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

$Rt = Ra_{[31:0]}$

Typical Values For Shift Register

Value	
0000	Stop clock completely
8888	25% rate
AAAA	50% rate
EEEE	75% rate
FFFF	Full power, max clock rate

Exceptions: none

STSB – Store String Byte

Description:

This instruction stores a byte contained in register Rb to consecutive memory locations beginning at the address in Ra until the loop counter LC reaches zero. Ra is updated with by the number of bytes written. This instruction is interruptible.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
Sg_3	0_3	$\sim_6$	Rb_6	Ra_6	$98h_8$	Pn_4	Pc_4

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        mem[Ra] = Rb[7:0]
        Ra = Ra + 1
        LC = LC – 1
```

STSC – Store String Character

Description:

This instruction stores a character (16 bit value) to consecutive memory locations beginning at the address in Ra until the loop counter reaches zero. The memory address must be character aligned. Ra is updated by the number of bytes written. This instruction is interruptible.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
Sg_3	1_3	$\sim_6$	Rb_6	Ra_6	$98h_8$	Pn_4	Pc_4

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        mem[Ra] = Rb[15:0]
        Ra = Ra + 2
        LC = LC – 1
```

STSET – String Set

Description:

This instruction stores data contained in register Rb to consecutive memory locations beginning at the address in Ra until the loop counter LC reaches zero. Ra is updated with by the number of bytes written. This instruction is interruptible. The data address must be appropriately aligned.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
Sg_3	0_3	$\sim_6$	Rb_6	Ra_6	$98h_8$	Pn_4	Pc_4

Execution Units: Memory

Operation:

```
while LC <> 0
        mem[Ra] = Rb
        Ra = Ra +/- amt
        LC = LC – 1
```

O_3	Assembler Mnemonic	
0	STSET.BI	set bytes incrementing
1	STSET.CI	set characters incrementing
2	STSET.HI	set half-word incrementing
3	STSET.WI	set words incrementing
4	STSET.BD	set bytes decrementing
5	STSET.CD	set characters decrementing
6	STSET.HD	set half-word decrementing
7	STSET.WD	set word decrementing

STSH – Store String Half-word

Description:

This instruction stores a half-word (32 bit value) to consecutive memory locations beginning at the address in Ra until the loop counter reaches zero. The memory address must be half-word aligned. Ra is updated by the number of bytes written. This instruction is interruptible.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
Sg_3 2_3	$\sim_6$	Rb_6	Ra_6	$98h_8$	Pn_4 Pc_4

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        mem[Ra] = Rb[31:0]
        Ra = Ra + 4
        LC = LC – 1
```

STSW – Store String Word

Description:

This instruction stores a word (64 bit value) to consecutive memory locations beginning at the address in Ra until the loop counter reaches zero. The memory address must be half-word aligned. Ra is updated by the number of bytes written. This instruction is interruptible.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
Sg_3 3_3	$\sim_6$	Rb_6	Ra_6	$98h_8$	Pn_4 Pc_4

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        mem[Ra] = Rb[63:0]
        Ra = Ra + 8
        LC = LC – 1
```

SUB - Register-Register

Description:

This instruction subtracts one register from another and places the result into a third register. This instruction may cause an overflow exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$01h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

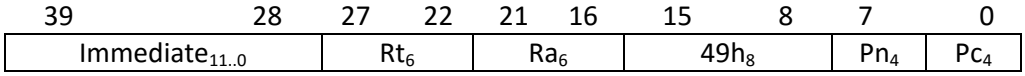
Rt = Ra - Rb

SUBI - Register-Immediate

Description:

This instruction subtracts an immediate value from a register and places the result into a register. This instruction may cause an overflow exception.

Instruction Format:



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra – Imm

SUBU - Register-Register

Description:

This instruction subtracts one register from another and places the result into a third register. This instruction never causes an exception.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$05h_6$	Rt_6	Rb_6	Ra_6	$40h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

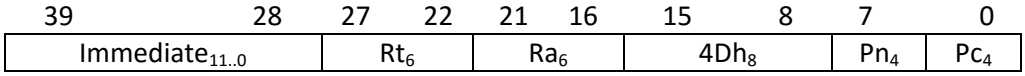
Rt = Ra - Rb

SUBUI - Register-Immediate

Description:

This instruction subtracts an immediate value from a register and places the result into a register. This instruction never causes an exception.

Instruction Format:



Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra – Imm

SW – Store Word

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 offset and register Ra. The memory address must be word aligned.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rt_6	Ra_6	$93h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

memory[Ra+offset] = Rb

Exceptions: DBE, DBG, TLB, LMT

SWCR – Store Word and Clear Reservation

Description:

If there is a reservation present on the memory address then a sixty-four bit value is stored to memory from the source register Rs and the reservation is cleared. If there is no reservation present then memory is not updated. If the update was successful then predicate register zero is set to 'ne' status, otherwise the predicate register is set to 'eq' status. The memory address is the sum of the sign extended offset and register Ra. The memory address must be word aligned. This instruction relies on the memory system for implementation.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Rs_6	Ra_6	$8Ch_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

memory[Ra+offset] = Rb, reservation cleared

Exceptions: DBE, DBG, TLB, LMT

SWS – Store Word Special

Description:

A sixty-four bit value is stored to memory from the source special purpose register Spr. The memory address is the sum of the sign extended displacement and register Ra. The memory address must be word aligned.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
Sg_3	$Displacement_{8..0}$	Spr_6	Ra_6	$9Eh_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

memory[Ra + displacement] = Spr

Exceptions: DBE, DBG, TLB, LMT

SWX – Store Word Indexed

Description:

A sixty-four bit value is stored to memory from the source register Rc. The memory address is the sum of register Ra and scaled register Rb. The memory address must be word aligned.

Instruction Format:

39 37	36	3534	33 28	27 22	21 16	15 8	7 0
Seg_3	~	Sc_2	Rc_6	Rb_6	Ra_6	$C3h_8$	Pn_4 Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

memory[Ra+Rb] = Rc

Exceptions: DBE, DBG, TLB, LMT

SXB – Sign Extend Byte

Description:

This instruction sign extends a register from bit 8 to 63 and places the result in a target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
C_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = {56{$Ra_{[7]}$}},$Ra_{[7:0]}$

Exceptions: none

SXC – Sign Extend Character

Description:

This instruction sign extends a register from bit 16 to 63 and places the result in a target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
D_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = {48{$Ra_{[15]}$}},$Ra_{[15:0]}$

Exceptions: none

SXH – Sign Extend Half-word

Description:

This instruction sign extends a register from bit 32 to 63 and places the result in a target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
E_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

$Rt = \{32\{Ra_{[31]}\}\},Ra_{[31:0]}$

Exceptions: none

SYNC – Synchronization Barrier

Description:

All instructions before the SYNC command are completed before any following instructions are started. Note that this instruction has an effect even if the predicate is false; this does not affect the correct operation of the program, only performance is affected.

Instruction Format:

15 — 8	7 — 4	3 — 0
$F7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Exceptions: none

SYS –Call system routine

Description:

This instruction calls a system function located as the sum of the offset times 16 plus code address register 12. The return address is stored in the EPC register (code address register #13). This instruction causes the core to switch to kernel mode.

Instruction Format:

31 24	23 20	19 16	15 8	7 0
$Offset_{7..0}$	Ch_4	Dh_4	$A5h_8$	Pn_4 Pc_4

Operation:

PC = offset * 16 + c12
if (StatusEXL < 255) StatusEXL = StatusEXL + 1

TLB – TLB Command

Description:

The command is executed on the TLB unit. The command results are placed in internal TLB registers which can be read or written using TLB command instruction. If the operation is a read register operation then the register value is placed into Rt. If the operation is a write register operation, then the value for the register comes from Rb. Otherwise the Rb/Rt field in the instruction is ignored.

This instruction is only available in kernel mode.

Instruction Format:

3130	29 24	23 16	15 8	7 0
$\sim_2$	Rb/Rt_6	Tn_4	Cmd_4	$F0h_8$	Pn_4	Pc_4

Clock Cycles: 3

Tn_4 – This field identifies which TLB register is being read or written.

Reg no.		Assembler
0	Wired	Wired
1	Index	Index
2	Random	Random
3	Page Size	PageSize
4	Virtual page	VirtPage
5	Physical page	PhysPage
7	ASID	ASID
8	Data miss address	DMA
9	Instruction miss address	IMA
10	Page Table Address	PTA
11	Page Table Control	PTC

TLB Commands

Cmd	Description	Assembler
0	No operation	
1	Probe TLB entry	TLBPB
2	Read TLB entry	TLBRD
3	Write TLB entry corresponding to random register	TLBWR
4	Write TLB entry corresponding to index register	TLBWI
5	Enable TLB	TLBEN
6	Disable TLB	TLBDIS

7	Read register	TLBRDREG
8	Write register	TLBWRREG
9	Invalidate all entries	TLBINV

Probe TLB – The TLB will be tested to see if an address translation is present.

Read TLB – The TLB entry specified in the index register will be copied to TLB holding registers.

Write Random TLB – A random TLB entry will be written into from the TLB holding registers.

Write Indexed TLB – The TLB entry specified by the index register will be written from the TLB holding registers.

Disable TLB – TLB address translation is disabled so that the physical address will match the supplied virtual address.

Enable TLB – TLB address translation is enabled. Virtual address will be translated to physical addresses using the TLB lookup tables.

The TLB will automatically update the miss address registers when a TLB miss occurs only if the registers are zero to begin with. System software must reset the registers to zero after a miss is processed. This mechanism ensures the first miss that occurs is the one that is recorded by the TLB.

PageTableAddr – This is a scratchpad register available for use to store the address of the page table.

PageTableCtrl – This is a scratchpad register available for use to store control information associated with the page table.

TST - Register Test Compare

Description:

The register test compare compares a register against the value zero and sets the predicate flags appropriately.

Instruction Format:

2322	21 16	15 12	11 8	7	0
0_2	Ra_6	0_4	Pt_4	Pn_4	Pc_4

Clock Cycles: 1

Operation:

```
if Ra < 0
        Pt.lt = 1
else
        Pt.lt = 0
if Ra = 0
        Pt.eq = 1
else
        Pt.eq = 0
Pt.ltu = 0
```

Exceptions: none

ZXB – Zero Extend Byte

Description:

This instruction zero extends a register from bit 8 to 63 and places the result in a target register. This instruction is typically used to perform an unsigned load operation with the LVB instruction.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
C_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

$Rt = Ra_{[7:0]}$

Exceptions: none

ZXC – Zero Extend Character

Description:

This instruction zero extends a register from bit 16 to 63 and places the result in a target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
D_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

$Rt = Ra_{[15:0]}$

Exceptions: none

ZXH – Zero Extend Half-word

Description:

This instruction zero extends a register from bit 32 to 63 and places the result in a target register.

Instruction Format:

31 28	27 22	21 16	15 8	7 0	
E_4	Rt_6	Ra_6	$A7h_8$	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

$Rt = Ra_{[31:0]}$

Exceptions: none

Stack Operations

Stack based operations are currently not supported. This document outlines proposed stack based instructions.

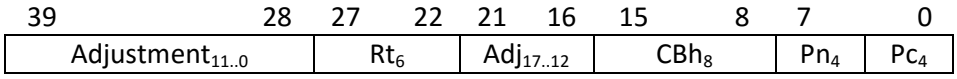
Stack based operations require en-queueing two sub-instructions in place of the stack instruction specified in program code. Logic cell requirements for the additional operations approx. 10,000 LC's.

LINK – Link Stack

Description:

The specified base pointer register Rt is pushed onto the stack, the stack pointer is loaded into register Rt and then is adjusted by the amount specified. The adjustment field of the instruction is multiplied by eight and sign extended before being applied allowing up to 128k bytes to be allocated. Note the adjustment field may not be extended with an immediate prefix. Also note the adjustment field value should be eight less than the desired value.

Instruction Format:



Clock Cycles: 4 (one memory access)

Execution Units: All ALU's / Memory

Operation:

memory[SP-8] = Rt
Rt = SP - 8
SP = SP + adjustment

PEA – Push Effective Address

Description:

An address value is calculated as the sum of the sign extended displacement and register Ra then pushed onto the stack.

Push and pop operations are unique as they enque as two instructions. This has a tendency to serialize the operation of the processor. It may improve performance in some applications to manually adjust the stack pointer, and use load / store operations instead.

Instruction Format:

3937	36 28	27 22	21 16	15 8	7 0
$\sim_3$	$Displacement_{8..0}$	$\sim_6$	Ra_6	$C9h_8$	Pn_4	Pc_4

Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

SP = SP - 8
memory[SP] = Ra + displacement

POP – Pop Register

Description:

The register is popped from the stack then the stack pointer is incremented.

Push and pop operations are unique as they enque as two instructions. This has a tendency to serialize the operation of the processor. It may improve performance in some applications to manually adjust the stack pointer, and use load / store operations instead.

Instruction Format:

23	22 16	15 8	7 0
$\sim_1$	Rt_7	CAh_8	Pn_4 Pc_4

Clock Cycles: 4 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = mem[r27]
r27 = r27 + 8

Registers Popped:

Regno (Rt_7)	Register Pushed
00 to 63	general register file
64 to 79	predicate registers #0 to #15
80 to 95	code address registers
96 to 111	segment registers
112	predicate register array
115	loop counter

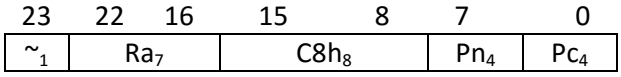
PUSH – Push Register

Description:

The stack pointer is decremented then the register is pushed onto the stack.

Push and pop operations are unique as they enque as two instructions. This has a tendency to serialize the operation of the processor. It may improve performance in some applications to manually adjust the stack pointer, and use load / store operations instead.

Instruction Format:



Clock Cycles: 3 (one memory access)

Execution Units: All ALU's / Memory

Operation:

r27 = r27 - 8
mem[r27] = Ra

Registers Pushed:

Regno (Ra_7)	Register Pushed
00 to 63	general register file
64 to 79	predicate registers #0 to #15
80 to 95	code address registers
96 to 111	segment registers
112	predicate register array
115	loop counter

UNLINK – Unlink Stack

Description:

The specified base pointer register Ra is loaded into the stack pointer, then register Ra is popped from the stack.

Instruction Format:

2322	21 16	15 8	7 0
$\sim_2$	Ra_6	CCh_8	Pn_4 Pc_4

Clock Cycles: 4 (one memory access)

Execution Units: All ALU's / Memory

Operation:

SP = Ra
Ra = memory[SP]
SP = SP + 8

Floating Point

Operations Supported

Only the most basic floating point operations are supported with hardware. Supported operations include addition, subtraction, multiplication, division, absolute value, integer to float and float to integer conversions. Also supported are comparison operations. There are also a number of control and status instructions.

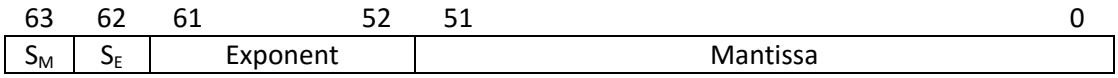
Supported Operations:

Mnemonic	Precision	Clocks	Operation
FADD	S,D	5	addition
FSUB	S,D	5	subtraction
FMUL	S,D	5	multiplication
FDIV	S,D	29,56	division
FABS	S,D	1	absolute value
FNEG	S,D	1	negation
FTOI	S,D	2	float to integer
ITOF	S,D	2	integer to float
FSIGN	S,D	1	sign of value
FMAN	S,D	1	mantissa of value
FSTAT	-	1	get status register
FRM	-	1	set rounding mode
FTX	-	1	trigger exception
TCX	-	1	clear exception
TDX	-	1	disable exception
FEX	-	1	enable exception
FCMP	S, D	1	comparison
FTST	S, D	1	test against zero

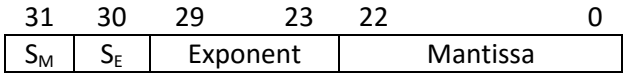
Representation

The floating point format is an IEEE-754 representation for both single and double precision. Briefly,

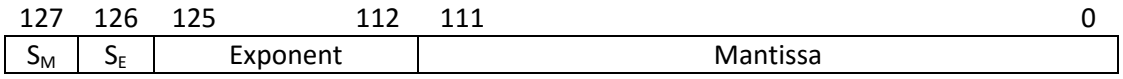
Double Precision Format:



Single Precision Format:



Quad Precision Format:



S_M – sign of mantissa
S_E – sign of exponent

The exponent and mantissa are both represented as two's complement numbers, however the sign bit of the exponent is inverted.

S_eEEEEEEEEE	
11111111111	Maximum exponent
….	
01111111111	exponent of zero
….	
00000000000	Minimum exponent

The exponent ranges from -1024 to +1023 for double precision numbers

If the core is built with the 32 bit data-bus 64 bit double precision floating point is unavailable.

Floating point comparisons and tests are executed on the integer ALU. This allows a comparison operation to proceed in parallel with another floating point operation.

Performance

Generally, double precision operations are just as fast as single precision operations with the exception of the divide operation which takes multiple clock cycles.

The floating point divider uses a radix 8 division. (three bits are processed each clock cycle).

Floating Point Instruction Set

FABS – Absolute Value

Description:

This instruction takes the absolute value of a double precision floating point number contained in a general purpose register. The sign bit of the number is cleared. The precision of the number is not affected and the number is not rounded.

Instruction Format:

31 28	27 22	21 16	15 8	7 0	
5_4	Rt_6	Ra_6	77_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

FABS.S – Single Precision Absolute Value

Description:

This instruction takes the absolute value of a single precision floating point number contained in a general purpose register. The sign bit of the number is cleared.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
5_4	Rt_6	Ra_6	79_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

FADD – Floating point addition

Description:

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

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
8_6	Rt_6	Rb_6	Ra_6	$78h_8$	Pn_4 Pc_4

Clock Cycles: 5

Execution Units: All Floating Point

FADD.S – Floating Point Single Precision addition

Description:

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

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
18_6	Rt_6	Rb_6	Ra_6	$78h_8$	Pn_4 Pc_4

Clock Cycles: 5

Execution Units: All Floating Point

FCMP - Float Compare

Description:

The register compare instruction compares two registers as floating point doubles and sets the flags in the target predict register as a result. While this is a floating point operation it is executed on the integer ALU.

Instruction Format:

3128	27 22	21 16	15 12	11 8	7	0
2_4	Rb_6	Ra_6	1_4	Pt_4	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

```
if Ra < Rb
        P.lt = true
else
        P.lt = false
if mag Ra < mag Rb
        P.ltu = true
else
        P.ltu = false
if Ra = Rb
        P.eq = true
else
        P.eq = false
if unordered
        P.un = true
else
        P.un = false
```

FCMP.S - Float Compare Single

Description:

The register compare instruction compares two registers as floating point singles and sets the flags in the target predict register as a result. While this is a floating point operation it is executed on the integer ALU.

Instruction Format:

3128	27 22	21 16	15 12	11 8	7	0
1_4	Rb_6	Ra_6	1_4	Pt_4	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

```
if Ra < Rb
        P.lt = true
else
        P.lt = false
if mag Ra < mag Rb
        P.ltu = true
else
        P.ltu = false
if Ra = Rb
        P.eq = true
else
        P.eq = false
if unordered
        P.un = true
else
        P.un = false
```

FDIV – Floating point division

Description:

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

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
Bh_6	Rt_6	Rb_6	Ra_6	$78h_8$	Pn_4 Pc_4

Clock Cycles: 56

Execution Units: All Floating Point

FDIV.S – Single Precision Floating point division

Description:

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

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
$1Bh_6$	Rt_6	Rb_6	Ra_6	$78h_8$	Pn_4 Pc_4

Clock Cycles: 29

Execution Units: All Floating Point

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:

31 28	27 22	21 16	15 8	7 0
D_4	$Immed_6$	Ra_6	79_8	Pn_4	Pc_4

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

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:

31 28	27 22	21 16	15 8	7 0
F_4	$Immed_6$	Ra_6	79_8	Pn_4	Pc_4

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:

31 28	27 22	21 16	15 8	7 0
E_4	$Immed_6$	Ra_6	79_8	Pn_4	Pc_4

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

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:

31 28	27 22	21 16	15 8	7 0
C_4	$Immed_6$	Ra_6	79_8	Pn_4	Pc_4

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

FMAC – Floating Point Multiply Accumulate (planned)

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.

Instruction Format:

4745	44 40	39 34	33 28	27 22	21 16	15 8	7 0	
~$_3$	O_5	Rt_6	Rc_6	Rb_6	Ra_6	$76h_8$	Pn_4	Pc_4

Clock Cycles: 10

Execution Units: All Floating Point

O_5	Precision	Mnemonic	Operation	
8	S	FMAC.S	Rt = (Ra * Rb) + Rc	multiply accumulate
9	S	FMAS.S	Rt = (Ra * Rb) - Rc	multiply subtract
10	S	FNMAC.S	Rt = -((Ra * Rb) + Rc)	negate multiply accumulate
11	S	FNMAS.S	Rt = -((Ra * Rb) - Rc)	negate multiply subtract
16	D	FMAC	Rt = (Ra * Rb) + Rc	multiply accumulate
17	D	FMAS	Rt = (Ra * Rb) - Rc	multiply subtract
18	D	FNMAC	Rt = -((Ra * Rb) + Rc)	negate multiply accumulate
19	D	FNMAS	Rt = -((Ra * Rb) - Rc)	negate multiply subtract

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.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
7_4	Rt_6	Ra_6	77_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

FMAN.S – Mantissa of Number

Description:

This instruction provides the mantissa of a single precision floating point number contained in a general purpose register as a 23 bit zero extended result. The hidden bit of the floating point number remains hidden.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
7_4	Rt_6	Ra_6	79_8	Pn_4 Pc_4

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

FMOV – Move Double Precision

Description:

This instruction moves one general purpose register to another. This instruction is shorter and uses one less register port than using the OR instruction to move between registers. See also the MOV instruction. This instruction currently performs the same operation as the MOV instruction.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
0_4	Rt_6	Ra_6	77_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

FMOV.S – Move Single Precision

Description:

This instruction moves one general purpose register to another. This instruction is shorter and uses one less register port than using the OR instruction to move between registers. See also the MOV instruction. This instruction currently performs the same operation as the MOV instruction.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
0_4	Rt_6	Ra_6	79_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

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:

39 34	33 28	27 22	21 16	15 8	7 0
Ah_6	Rt_6	Rb_6	Ra_6	$78h_8$	Pn_4	Pc_4

Clock Cycles: 5

Execution Units: All Floating Point

FMUL.S – Single Precision Floating point multiplication

Description:

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

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
$1Ah_6$	Rt_6	Rb_6	Ra_6	$78h_8$	Pn_4 Pc_4

Clock Cycles: 5

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.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
4_4	Rt_6	Ra_6	77_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

FNEG.S – Negate Single Precision

Description:

This instruction negates a single precision floating point number contained in a general purpose register. The sign bit of the number is inverted.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
4_4	Rt_6	Ra_6	79_8	Pn_4	Pc_4

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:

31 28	27 22	21 16	15 8	7 0
D_4	Imm_6	Ra_6	77_8	Pn_4 Pc_4

Execution Units: All Floating Point

Operation:

FPSCR.RM = Ra | Immediate

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:

31 28	27 22	21 16	15 8	7 0
6_4	Rt_6	Ra_6	77_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

FSIGN.S – Single Precision Sign of Number

Description:

This instruction provides the sign of a single precision floating point number contained in a general purpose register as a floating point single 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:

31 28	27 22	21 16	15 8	7 0
6_4	Rt_6	Ra_6	79_8	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

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:

31 28	27 22	21 16	15 8	7 0
C_4	Rt_6	$\sim_6$	77_8	Pn_4	Pc_4

Execution Units: All Floating Point

Operation:

Rt = FPSCR

Floating Point Status And Control Register Format:

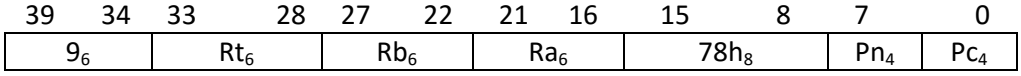
Bit		Symbol	Description
31:29	**RM**	rm	rounding mode (unimplemented)
28	**E5**	inexe	- inexact exception enable
27	**E4**	dbzxe	- divide by zero exception enable
26	**E3**	underxe	- underflow exception enable
25	**E2**	overxe	- overflow exception enable
24	**E1**	invopxe	- invalid operation exception 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 – Floating point subtraction

Description:

Instruction Format:



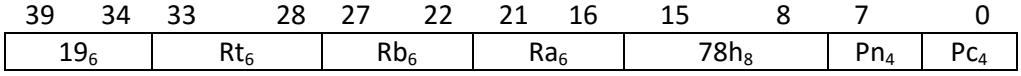
Clock Cycles: 5

Execution Units: All Floating Point

FSUB.S – Single Precision Floating point subtraction

Description:

Instruction Format:



Clock Cycles: 5

Execution Units: All Floating Point

FTOI – Float to Integer

Description:

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

Instruction Format:

31 28	27 22	21 16	15 8	7 0
2_4	Rt_6	Ra_6	$77h_8$	Pn_4	Pc_4

Clock Cycles: 2

Execution Units: All Floating Point

FTOI.S – Single Precision Float to Integer

Description:

This instruction converts a floating point single value to an integer value.

Instruction Format:

31 28	27 22	21 16	15 8	7 0
2_4	Rt_6	Ra_6	$79h_8$	Pn_4	Pc_4

Clock Cycles: 2

Execution Units: All Floating Point

FTST – Float Register Test Compare

Description:

The register test compare compares floating point double in a register against the value zero and sets the predicate flags appropriately. This instruction is executed on the integer ALU.

Instruction Format:

2322	21 16	15 12	11 8	7	0
2_2	Ra_6	0_4	Pt_4	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

```
if Ra < 0
        Pt.lt = 1
else
        Pt.lt = 0
if Ra = 0
        Pt.eq = 1
else
        Pt.eq = 0
if unordered
        Pt.un = 1
else
        Pt.un = 0
Pt.ltu = 0
```

Exceptions: none

FTST.S – Float Single Test Compare

Description:

The register test compare compares floating point single in a register against the value zero and sets the predicate flags appropriately. This instruction is executed on the integer ALU.

Instruction Format:

2322	21 16	15 12	11 8	7	0
1_2	Ra_6	0_4	Pt_4	Pn_4	Pc_4

Clock Cycles: 1

Execution Units: All ALU's

Operation:

```
if Ra < 0
        Pt.lt = 1
else
        Pt.lt = 0
if Ra = 0
        Pt.eq = 1
else
        Pt.eq = 0
if unordered
        Pt.un = 1
else
        Pt.un = 0
Pt.ltu = 0
```

Exceptions: none

ITOF – Integer to Float

Description:

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

Instruction Format:

31 28	27 22	21 16	15 8	7 0
3_4	Rt_6	Ra_6	$77h_8$	Pn_4	Pc_4

Clock Cycles: 2

Execution Units: All Floating Point

ITOF.S – Integer to Float Single

Description:

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

Instruction Format:

31 28	27 22	21 16	15 8	7 0
3_4	Rt_6	Ra_6	$79h_8$	Pn_4 Pc_4

Clock Cycles: 2

Execution Units: All Floating Point

Vector Programming Model

The ISA supports up to 64 vector registers of length 64. In the proof of concept RTL code there are eight, sixteen element vector registers.

Reg no	
0	
1	
2	
3	
4	
5	
6	
7	

Vector Length (VL register)

The vector length register controls how many elements of a vector are processed.

7	6 ... 0
0	$Length_{6..0}$

Vector Predicates

The ISA supports up to sixteen, sixty-four element vector predicate registers. These registers take the place of the vector mask (VM) register in other architectures. In the proof-of-concept version there is a single sixteen element vector predicate register. All vector instructions are executed conditionally based on the value in a vector predicate register.

Regno	
0	

Predicate Conditions

Cond.		Test	
0	PF	0	Always false – Instructions predicated with condition zero never execute regardless of the predicate register contents. This is used for extended immediate values as well. The false predicate byte for instructions is 90h.
1	PT	1	Always True – The instruction predicated with an always true condition always executes regardless of the predicate register contents. The always true predicate byte is 01h. Other true predicates are instruction short-forms.
2	PEQ	eq	Equal – instruction executes if the predicate register equal flag is set
3	PNE	!eq	Not Equal – instruction executes if the predicate register equal flag is clear
4	PLE	lt|eq	Less or Equal – predicate less or equal flag is set
5	PGT	!(lt|eq)	greater than
6	PGE	!lt	greater or equal
7	PLT	lt	less than
8	PLEU	ltu|eq	unsigned less or equal
9	PGTU	!(ltu|eq)	unsigned greater than
10	PGEU POR	!ltu	unsigned greater or equal Ordered for floating point
11	PLTU PUN	ltu	unsigned less than Unordered for floating point
12			
13	PSIG	signal	execute if external signal is true
14			
15			

Detailed Vector Instruction Set

VADD

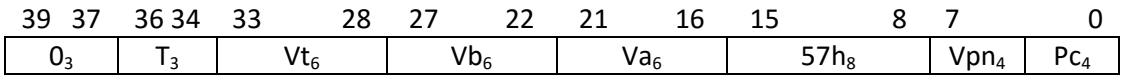
Synopsis

Vector register add. Vt = Va + Vb

Description

Two vector registers (Va and Vb) are added together and placed in the target vector register Vt.

Instruction Format



Operation

for x = 0 to VL - 1

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

Operand Type

T_3		
0	Integer	
1	Float single	
2	Float double	
4	Float quad	

VADDS

Synopsis

Vector register add. Vt = Va + Rb

Description

A vector and a scalar (Va and Rb) are added together and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
4_3	T_3	Vt_6	Rb_6	Va_6	$57h_8$	Vpn_4	Pc_4

Operation

for x = 0 to VL-1

if (VM[x]) Vt[x] = Vb[x] + Rb

Operand Type

T_3		
0	Integer	
1	Float single	
2	Float double	
4	Float quad	

VAND

Synopsis

Vector register bitwise and. Vt = Va & Vb

Description

Two vector registers (Va and Vb) are bitwise and'ed together and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
0_3	T_3	Vt_6	Vb_6	Va_6	$52h_8$	Vpn_4	Pc_4

Operation

```
for x = 0 to VL-1
    if (VM[x]) Vt[x] = Va[x] & Vb[x]
```

Operand Type

T_3		
0	Integer	

VANDS

Synopsis

Vector register bitwise and. Vt = Va & Rb

Description

A vector registers (Va) is bitwise and'ed with a scalar register and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
4_3	T_3	Vt_6	Rb_6	Va_6	$52h_8$	Vpn_4	Pc_4

Operation

```
for x = 0 to VL-1
    if (VM[x]) Vt[x] = Va[x] & Rb
```

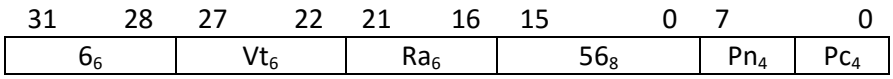
Operand Type

T_3		
0	Integer	

VBITS2V

Synopsis

Convert bits to Boolean vector.



Description

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

Operation

For x = 0 to [Ra]-1

Vt[x] = Ra[x]

Exceptions: none

VCMP

Synopsis

Vector register compare. Vt = Va ? Vb

Description

Two vector registers (Va and Vb) are compared and the comparison result is placed in the target vector predicate register Vpt.

Instruction Format

3128	27 22	21 16	15 12	11 8	7 0
O_4	Vb_6	Va_6	1_4	Vpt_4	Vpn_4	Pc_4

Operation

for x = 0 to VL-1

Vt[x] = Va[x] ? Vb[x]

Operation:

For each vector element

```
if signed Va < signed Vb
        P.lt = true
else
        P.lt = false
if unsigned Va < unsigned Vb
        P.ltu = true
else
        P.ltu = false
if Va = Vb
        P.eq = true
else
        P.eq = false
```

Operand Type

O_4		
8	Integer	
9	Float single	
A	float double	
C	Float Quad	

VCMPS

Synopsis

Vector register compare. Vt = Va ? Rb

Description

A vector registers (Va) is compared to a scalar register (Rb) and the comparison result is placed in the target vector predicate register Vpt.

Instruction Format

39 35	3432	31 28	27 22	21 16	15 8	7 0
$\sim_5$	T_3	Vpt_4	Rb_6	Va_6	$5Ch_8$	Vpn_4	Pc_4

Operation

for x = 0 to VL-1

Vt[x] = Va[x] ? Vb[x]

Operation:

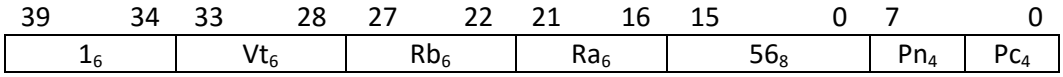
For each vector element

```
if signed Va < signed Rb
        P.lt = true
else
        P.lt = false
if unsigned Va < unsigned Rb
        P.ltu = true
else
        P.ltu = false
if Va = Rb
        P.eq = true
else
        P.eq = false
```

VEINS / VMOVSV

Synopsis

Vector element insert.



Description

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

Operation

Rt = Va[Ra]

Exceptions: none

VEOR

Synopsis

Vector register bitwise exclusive or. Vt = Va ^ Vb

Description

Two vector registers (Va and Vb) are exclusively or'ed together and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
2_3	T_3	Vt_6	Vb_6	Va_6	$52h_8$	Vpn_4	Pc_4

Operation

```
for x = 0 to VL-1
    Vt[x] = Va[x] & Vb[x]
```

Operand Type

T_3		
0	Integer	

VEORS

Synopsis

Vector register bitwise exclusive or. Vt = Va ^ Rb

Description

A vector registers (Va) is exclusively or'ed with a scalar register (Rb) and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
6_3	T_3	Vt_6	Rb_6	Va_6	$52h_8$	Vpn_4	Pc_4

Operation

```
for x = 0 to VL-1
    if (VM[x]) Vt[x] = Va[x] & Vb[x]
```

Operand Type

T_3		
0	Integer	

VDIV

Synopsis

Vector register divide. Vt = Va / Vb

Description

Two vector registers (Va and Vb) are divided and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
1_3	T_3	Vt_6	Vb_6	Va_6	$5Eh_8$	Vpn_4	Pc_4

Operation

Vt[Ra] =Va[Ra] + Vb[Ra]

or

for x = 1 to [Ra]

Vt[x] = Va[x] + Vb[x]

Operand Type

T_3		
0	Integer	
1	Float single	
2	float double	
4	Float Quad	

VDIVS

Synopsis

Vector register divide. Vt = Va / Rb

Description

A vector register (Va) is divided by a scalar and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
5_3	T_3	Vt_6	Vb_6	Va_6	$5Eh_8$	Vpn_4	Pc_4

Operation

for x = 0 to VL-1

Vt[x] = Va[x] / Rb

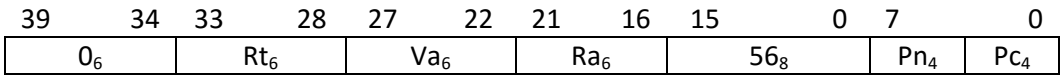
Operand Type

T_3		
0	Integer	
1	Float single	
2	float double	
4	Float Quad	

VEX / VMOVS

Synopsis

Vector element extract.



Description

One element of a vector register Va is transferred to a general purpose register Rt. The element to extract is identified by Ra.

Operation

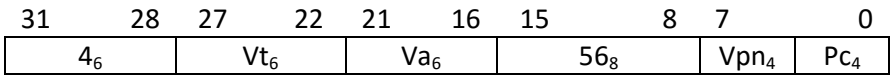
Rt = Va[Ra]

Exceptions: none

VFLT2INT

Synopsis

Vector float to integer.



Description

Elements of the vector are converted from floating point to integer.

Operation

For x = 0 to [Ra]-1

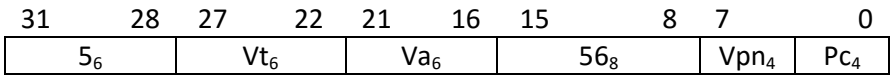
Vt[x] = (int)Va[x]

Exceptions: none

VINT2FLT

Synopsis

Vector float to integer.



Description

Elements of the vector are converted from integer to floating point.

Operation

For x = 0 to VL-1

Vt[x] =(float) Va[x]

Exceptions: none

LV

Synopsis

Load vector

Description:

Load a vector register from memory using register indirect.

Instruction Format:

31 29	28	27 22	21 16	15 0	7 0
Seg_3	$~_1$	Vt_6	Ra_6	BDh_8	Vpn_4	Pc_4

Operation

for x = 0 to [Ra]-1

$Vt[x] = memory_{64}[Ra + 8 * x]$

Exceptions: DBE, DBG, LMT, TLB

LVWS

Synopsis

Load vector using stride

Description:

Load a vector register from memory using register indirect with stride addressing.

Instruction Format:

39 37	36 34	33 28	27 22	21 16	15 0	7 0	
Seg_3	$\sim_3$	Vt_6	Rb_6	Ra_6	BEh_8	Vpn_4	Pc_4

Operation

for x = 0 to VL-1

$Vt[x] = memory_{64}[Ra+Rb * x]$

Exceptions: DBE, DBG, LMT, TLB

LVX

Synopsis

Load vector

Description:

Load a vector register from memory using vector indexed addressing.

Instruction Format:

39 37	36 34	33 28	27 22	21 16	15 0	7 0	
Seg_3	$\sim_3$	Vt_6	Vb_6	Ra_6	BFh_8	Vpn_4	Pc_4

Operation

for x = 0 to [Ra]-1

Vt[x] = $memory_{64}$[Ra+Vb[x]]

Exceptions: DBE, DBG, LMT, TLB

VMAC

Synopsis

Vector register multiply accumulate. Vt = +-(+- Va *+- Vb +- Vc)

Description

Vector registers Va and Vb are multiplied together then accumulated with vector register Vc. The sign of Va, Vb, and Vc may be inverted. The sign of the entire result may be inverted.

Instruction Format

47 43	42 40	39 34	33 28	27 22	21 16	15 8	7 0
Sn_5	T_3	Vt_6	Vc_6	Vb_6	Va_6	$5Ah_8$	Vpn_4	Pc_4

Operation

Vt[Ra] =Va[Ra] * Vb[Ra] + Vc

Operand Type

T_3		
0	Integer	
1	Float single	
2	float double	
4	Float Quad	

VMUL

Synopsis

Vector register add. Vt = Va * Vb

Description

Two vector registers (Va and Vb) are multiplied together and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
0_3	T_3	Vt_6	Vb_6	Va_6	$5Eh_8$	Vpn_4	Pc_4

Operation

```
for x = 0 to VL-1

    if (VM[x]) Vt[x] = Va[x] * Vb[x]
```

Operand Type

T_3		
0	Integer	
1	Float single	
2	float double	
4	Float Quad	

VMULS

Synopsis

Vector register add. Vt = Va * Rb

Description

A vector register (Va) is multiplied by a scalar register (Rb) and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
4_3	T_3	Vt_6	Rb_6	Va_6	$5Eh_8$	Vpn_4	Pc_4

Operation

for x = 0 to VL-1

if (VM[x]) Vt[x] = Va[x] * Rb

Operand Type

T_3		
0	Integer	
1	Float single	
2	float double	
4	Float Quad	

VOR

Synopsis

Vector register bitwise or. Vt = Va | Vb

Description

Two vector registers (Va and Vb) are or'ed together and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
1_3	T_3	Vt_6	Vb_6	Va_6	$52h_8$	Vpn_4	Pc_4

Operation

for x = 01 to VL-1

Vt[x] = Va[x] | Vb[x]

Operand Type

T_3		
0	Integer	

VORS

Synopsis

Vector register bitwise or. Vt = Va | Rb

Description

A vector register (Va) is or'ed with a scalar register (Rb) and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 8	7 0	
5_3	T_3	Vt_6	Rb_6	Va_6	$52h_8$	Vpn_4	Pc_4

Operation

```
for x = 0 to VL-1
    Vt[x] = Va[x] | Rb
```

Operand Type

T_3		
0	Integer	

VREC

Synopsis

Vector reciprocal. Vt = 1/Va

Description

The reciprocal of a vector (Va) is calculated placed in the target vector register Vt.

Instruction Format

31 28	27 22	21 16	15 0	7 0	
6_6	Vt_6	Va_6	56_8	Vpn_4	Pc_4

Operation

for x = 0 to VL-1

if (VM[x]) Vt[x] = 1/ Va[x]

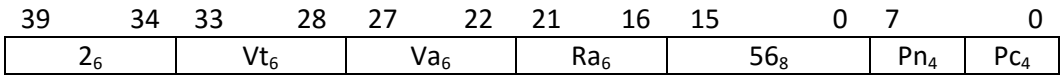
Operand Type

T_3		
0	Integer	
1	Float single	
2	float double	
4	Float Quad	

VSHLV

Synopsis

Vector shift left.



Description

Elements of the vector are transferred upwards to the next element position. Element #0 is loaded with the value zero.

Operation

For x = 0 to [Ra]-1

Vt[x+1] = Va[x]

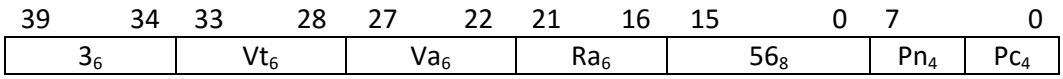
Vt[0] = 0

Exceptions: none

VSHRV

Synopsis

Vector shift right.



Description

Elements of the vector are transferred downwards to the next element position. The last is loaded with the value zero.

Operation

For x = 0 to [Ra]-1

Vt[x] = Va[x+1]

Vt[Ra-1] = 0

Exceptions: none

SV

Synopsis

Store vector

Description:

Store a vector register to memory using register indirect.

Instruction Format:

31 29	28	27 22	21 16	15 0	7 0
Seg_3	$\sim_1$	Vs_6	Ra_6	CDh_8	Vpn_4	Pc_4

Operation

for x = 0 to VL-1

$memory_{64}$[Rb + x * 8] = Vs[x]

Exceptions: DBE, DBG, LMT, TLB

SVWS

Synopsis

Store vector using stride

Description:

Store a vector register to memory using register indirect with stride addressing.

Instruction Format:

39 37	36 34	33 28	27 22	21 16	15 0	7 0	
Seg_3	$\sim_3$	Vt_6	Rb_6	Ra_6	CEh_8	Vpn_4	Pc_4

Operation

for x = 0 to [Ra]-1

$memory_{64}$[Ra+Rb * x] = Vs[x]

Exceptions: DBE, DBG, LMT, TLB

SVX

Synopsis

Store vector

Description:

Store a vector register to memory using vector indexed addressing.

Instruction Format:

39 37	36 34	33 28	27 22	21 16	15 0	7 0	
Seg_3	$\sim_3$	Vs_6	Vb_6	Ra_6	CFh_8	Vpn_4	Pc_4

Operation

for x = 0 to [Ra]-1

$memory_{64}$[Ra+Vb[x]] = Vs[x]

Exceptions: DBE, DBG, LMT, TLB

VSUB

Synopsis

Vector register subtract. Vt = Va - Vb

Description

Two vector registers (Va and Vb) are subtracted and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 0	7 0	
2_3	T_3	Vt_6	Vb_6	Va_6	$57h_8$	Vpn_4	Pc_4

Operation

Vt[Ra] =Va[Ra] + Vb[Ra]

or

for x = 1 to [Ra]

Vt[x] = Va[x] + Vb[x]

Operand Type

T_3		
0	Integer	
1	Float single	
2	float double	
4	Float Quad	

VSUBS

Synopsis

Vector register subtract. Vt = Va - Rb

Description

A scalar register is subtracted from a vector register and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 0	7 0	
5_3	T_3	Vt_6	Rb_6	Va_6	$57h_8$	Vpn_4	Pc_4

Operation

for x = 0 to [Ra]-1

Vt[x] = Va[x] + Rb

Operand Type

T_3		
0	Integer	
1	Float single	
2	float double	
4	Float Quad	

VSUBRS

Synopsis

Vector register subtract. Vt = Rb - Va

Description

A vector register is subtracted from a scalar register and placed in the target vector register Vt.

Instruction Format

39 37	36 34	33 28	27 22	21 16	15 0	7 0	
6_3	T_3	Vt_6	Rb_6	Va_6	$57h_8$	Vpn_4	Pc_4

Operation

for x = 0 to [Ra]-1

Vt[x] = Va[x] + Rb

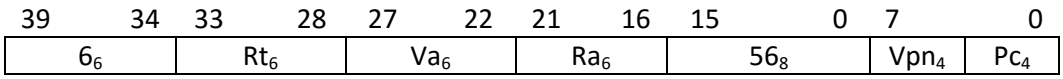
Operand Type

T_3		
0	Integer	
1	Float single	
2	float double	
4	Float Quad	

V2BITS

Synopsis

Convert Boolean vector to bits.



Description

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

Operation

For x = 0 to VL-1

Rt[x] = Va[x].LSB

Exceptions: none

Notes:

The register tag associated with a vector register contains both the element number and vector register number. The vector element being processed needs to be uniquely identified in the processor's pipeline. With a large number of vector registers and a large number of elements the tag becomes quite large. The current core has 8 vector registers with sixteen elements each. This is to keep the tag within seven bits. The core ends up needing to process an eight bit register tag in order to handle all the registers.

Use of vector instructions serializes the core's queuing of instructions. A separate instruction is queued for each element of the vector. This is done by stalling the instruction queued indicator until instructions for all the elements have been queued. In order to queue the vector instruction, queuing an element isn't complete until the length is known. The length is in special register VL. If this register is valid the instruction will queue right away, otherwise it will be delayed until VL (argument a1) becomes valid. Once a1 becomes valid a new instruction should enqueue every clock cycle.

Opcode Map

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	TST / FTST / FSTST															
1x	CMP / FCMP / FSCMP															
2x	CMPI															
3x	BR															
4x	{RR}	{R}	{P}			CHKI	BITI	ADDUI	ADDI	SUBI	MULI	DIVI	ADDUI	SUBUI	MULUI	DIVUI
5x	{logic}	MLO	{VLOG}	ANDI	ORI	EORI	{VR}	{VAdd }	{shift}		VMAC	MODI	VCMPS	CHKXI	{VMUL}	MODUI
6x											LLA	_2ADDUI	_4ADDUI	_8ADDUI	_16ADDUI	LDI
7x	JGR		MUX				{FMAC}	{doubler}	{float rr}	{singler}						
8x	LB	LBU	LC	LCU	LH	LHU	LW	LFS	LFD			LVWAR	SWCR	JSRI	LWS	LCL
9x	SB	SC	SH	SW	SFS	SFD	STI	CAS	STSET	STMOV	STCMP	STFND		LDIS	SWS	CACHE
Ax	JSR	JSR	JSR	RTS	LOOP	SYS	INT	{R}	MFSPR	MTSPR	{bitfld}	MOVS	LVB	LVC	LVH	LVW
Bx	LBX	LBUX	LCX	LCUX	LHX	LHUX	LWX	JSRIX	LLAX					LV	LVWS	LVX
Cx	SBX	SCX	SHX	SWX	SFSX	SFDX	STIX	INC	PUSH	PEA	POP	LINK	UNLINK	SV	SVWS	SVX
Dx							LW									
Ex																
Fx	{TLB}	NOP	RTS	RTE	RTI	{BCD}	STP	SYNC	MEMSB	MEMDB	CLI	SEI	RTD	RTF	JSF	IMM

40 - {RR} Opcodes –$Func_6$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	ADD	SUB	MUL	DIV	ADDU	SUBU	MULU	DIVU	2ADDU	4ADDU	8ADDU	16ADDU				
1x	MIN	MAX	SQRT	MOD	CHKX	CHK		MODU								
2x																
3x																

50 - {logic} Opcodes – $Func_6$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	AND	OR	EOR	NAND	NOR	ENOR	ANDC	ORC								
1x																
2x																
3x																

F5 {BCD} Opcodes – $Func_6$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BCDADD	BCDSUB	BCDMUL													

78 - {float -rr} Opcodes –Func$_6$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x								FCMP	FADD	FSUB	FMUL	FDIV				
1x								FCMPS	FADDS	FSUBS	FMULS	FDIVS				
2x																
3x																

77 - Double {R} Opcodes – Func$_4$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	FMOV		FTOI	ITOF	FNEG	FABS	FSIGN	FMAN	FNABS				FSTAT	FRM		

79 - Single {R} Opcodes – Func$_4$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	FMOVS		FTOIS	ITOFS	FNEGS	FABSS	FSIGNS	FMANS	FNABSS				FTX	FCX	FEX	FDX

A7 {R} Opcodes – Func$_4$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	MOV	NEG	NOT	ABS	SGN	CNTLZ	CNTLO	CNTPOP	SXB	SXC	SXH	COM	ZXB	ZXC	ZXH	

41 {R} Opcodes – Func$_4$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	CPUID	REDOR	REDAND	PAR												

42 - {Predicate Logic} Opcodes – Func$_6$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	PAND	POR	PEOR	PNAND	PNOR	PENOR	PANDC	PORC								
1x																
2x																
3x																

52 {VLog} Opcodes – Func$_3$

	x0	x1	x2	x3	x4	x5	x6	x7
0x	VAND	VOR	VEOR		VANDS	VORS	VEORS	

57 {VAdd} Opcodes – Func$_3$

	x0	x1	x2	x3	x4	x5	x6	x7
0x	VADD	VSUB			VADDS	VSUBS	VSUBRS	

58 - {shift} Opcodes – $Func_6$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	SHL	SHR	SHLU	SHRU	ROL	ROR										
1x	SHLI	SHRI	SHLUI	SHRUI	ROLI	RORI										
2x																
3x																

AA {Bitfield} Opcodes – $Func_4$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BFINS	BFSET	BFCLR	BFCHG	BFEXTU	BFEXT	BFINSI									