2015

Thor Guide

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

robfinch@Finitron.ca
Finitron
1/1/2015

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.

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.

Programming Model

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		
r4		
r5		
r6		
r7		
r8		
r9		
r10		
r11		
r12		
r13		
r14		
r15		
r16		
r17		
r18		
r19		
r20		
r21		
r22		
r23		
r24		
r26	Base Pointer	
r27	User Stack Pointer	
r28	Accessible only in kernel mode	
r29		
r30		
r31		
r32/F0	Floating point	
…		
r63/F31		

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		This register is available for general use.
6		This register is available for general use.
7		This register is available for general use.
8		This register is available for general use.
9		
10		
11	Catch Link Register	Used by the compiler to link to try/catch handlers.
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 program counter register is read-only. The program counter cannot be modified by moving a value to this register.

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.
1	PT	1	Always True – The instruction predicated with an always true condition always executes regardless of the predicate register contents.
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		
	IM	~	S	FXE		

The Kernel / Application Mode indicator is read-only.

IM = interrupt mask

Maskable interrupts are disabled when this bit is set.

Segmentation

The processor contains sixteen segment registers. The upper nibble of an address (bits 60 to 63) identifies which segment register to use during address formation for data addresses. For code addresses segment register #15 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

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.

Address Formation:

Non-segmented address bits 0 to 11 pass through the segmentation module unchanged. Address bits 59 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[59:12]	Address[11:0]
+	+
Segment register value[63:12]	000_{12}
=	
Segmented address[63:0]	

Selecting a segment register

The upper nybble of an address (bits 60 to 63) identifies which segment register to use. This selection applies to data addresses only. Code addresses always use segment register #15 – the code segment.

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.

Segment Usage Conventions

Segment register #15 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 #14 is the stack segment (SS) register by convention. Segment register #1 is the data segment (DS) by convention.

Segment register #13 is the volatile data segment (VDS). Addresses formed using this segment register bypass the data cache.

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'hFFFFFFFFFFFFFFF0. 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	NS	NULL segment	by convention contains zero
1	DS	data segment	by convention
2	TS	thread storage	by convention
3	BS	BSS segment	by convention
4	RS	read only segment	by convention
5	ES	extra segment	by convention
…			
13	VDS	volatile data segment	bypasses the cache
14	SS	Stack segment	by convention
15	CS	Code segment	always used for code addressing

Instruction Formats:

CS:	DS:	SS:	ES:	FS:	GS:
A0	B0	C0	D0	E0	F0

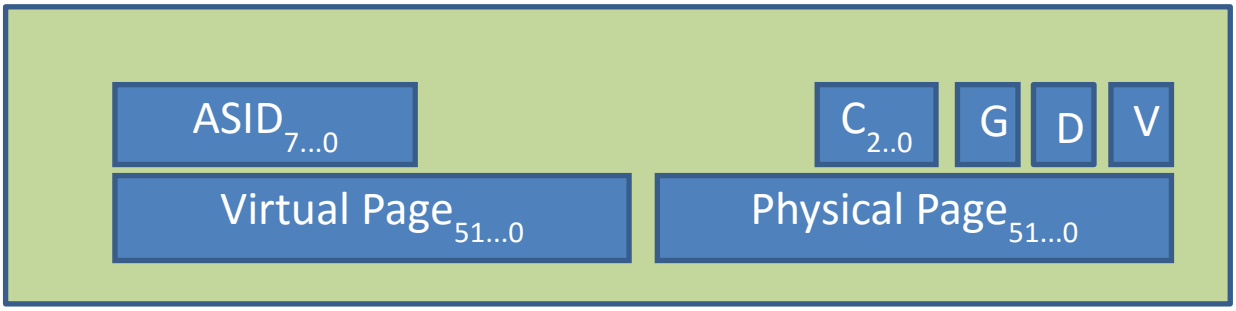
TLB

The processor 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.

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.

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.

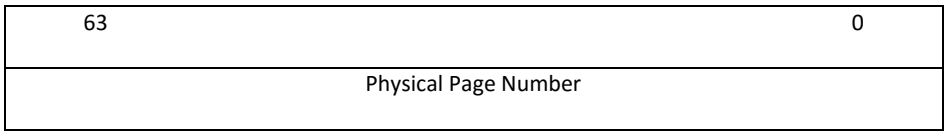
TLBPageSize (#3h)

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.



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

Vectors

The processor vectors to $FFFFFFFFFFFFFFF0 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.

Vector table:

Vector Number	Usage / Description			
0	BREAK instruction vector			
1	SLEEP vector (branch to self)			
2	Task reschedule interrupt			
…				
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		
…				
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			

Hardware Ports

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

	I/O	Width	WB		
rst_i	I	1	WB	reset signal	
clk_i	I	1	WB	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	
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

Notes:

Stores issue only from the head of the instruction queue when it is known that no exceptions have taken place.

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 opcodes may be preceded by an extended constant value.

RR - Register-Register

39 34	33 28	27 22	21 16	15 8	7 0	
Func	Rt	Rb	Ra	Opcode	Predicate	
$Func_6$	Rt_6	Rb_6	Ra_6	$Opcode_8$	Pn_4	Pc_4

RI - Register-Immediate

39 28	27 22	21 16	15 8	7 0	
$Immediate_{11..0}$	Rt_6	Ra_6	$Opcode_8$	Pn_4	Pc_4

CMP Register-Register Compare

31 28	27 22	21 16	15 12	11 8	7 0	
Opc_4	Rb_6	Ra_6	1_4	Pt_4	Pn_4	Pc_4

CMPI Register-Immediate Compare

31 22	21 16	15 12	11 8	7 0	
$Immed_{9..0}$	Ra_6	2_4	Pt_4	Pn_4	Pc_4

TST - Register Test Compare

2322 21 16	15 12	11 8	7 0	
O_2 Ra_6	0_4	Pt_4	Pn_4	Pc_4

CTRL- Control

15 8	7 0	
$Opcode_8$	Pn_4	Pc_4

BR - Relative Branch

23 16	15 8	7 0	
$Disp_{7..0}$	3_4 $D_{11..8}$	Pn_4	Pc_4

BRK/NOP

7 0	
$0/1_4$	0_4

RTS

7 0	
1_4	1_4

JSR - Jump To Subroutine

47 24	23 16	15 8	7 0	
$Offset_{23..0}$	Cr_4 Crt_4	$Opcode_8$	Pn_4	Pc_4

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

Operation:

Rt = Ra * 2 + Rb

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

Operation:

Rt = Ra * 2 + immediate

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

Operation:

Rt = Ra * 4 + Rb

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

Operation:

Rt = Ra * 4 + immediate

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

Operation:

Rt = Ra * 8 + Rb

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:

39 28	27 22	21 16	15 8	7 0
$Immed_{11..0}$	Rt_6	Ra_6	$6Dh_8$	Pn_4 Pc_4

Operation:

Rt = Ra * 8 + immediate

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

Operation:

Rt = Ra * 16 + Rb

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

Operation:

Rt = Ra * 16 + immediate

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

Operation:

Rt = Ra + Rb

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:

39 28	27 22	21 16	15 8	7 0	
$Immediate_{11..0}$	Rt_6	Ra_6	$48h_8$	Pn_4	Pc_4

Operation:

Rt = Ra + immediate

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

Operation:

Rt = Ra + Rb

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

Operation:

Rt = Ra + Immediate

AND - Register-Register

Description:

Logically 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

Operation:

Rt = Ra & Rb

ANDI - Register-Immediate

Description:

Logically 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

Operation:

Rt = Ra & immediate

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

Operation:

Rt = Ra + Rb

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

Operation:

Rt = Ra * Rb

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:

39 34	33 28	27 22	21 16	15 8	7 0	
$01h_6$	Rt_6	Rb_6	Ra_6	$F5h_8$	Pn_4	Pc_4

Operation:

Rt = Ra - Rb

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	
~$_4$	3_4	me_6	mb_6	Rt_6	Ra_6	AAh_8	Pn_4	Pc_4

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

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:

4744	43 40	39 34	33 28	27 22	21 16	15 8	7 0	
$\sim_4$	5_4	me_6	mb_6	Rt_6	Ra_6	AAh_8	Pn_4	Pc_4

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:

4744	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

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

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

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 ifetch 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

Operation:

PC <= PC + displacement

BRK –Break

Description:

This instruction contains only a predicate byte. The Break exception is executed.

Instruction Format:

7	0
0_4	0_4

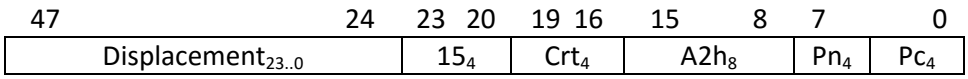
BSR - Branch to Subroutine

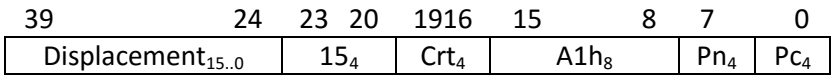
Description:

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.

Instruction Formats:





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.

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]

CLI – Clear Interrupt Mask

Description:

This instruction is used to enable interrupts.

Instruction Format:

15 8	7	0
FAh_8	Pn_4	Pc_4

Operation:

im = 0

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

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
```

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.

Instruction Format:

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

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
```

EOR - Register-Register

Description:

Logically 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

Operation:

Rt = Ra ^ Rb

EORI - Register-Immediate

Description:

Logically 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

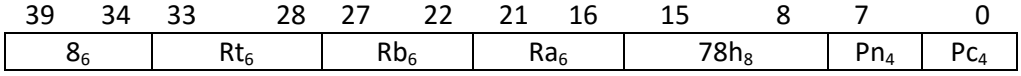
Operation:

Rt = Ra ^ immediate

FADD – Floating point addition

Description:

Instruction Format:



FCMP Float Register-Register 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.

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

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
```

FSUB – Floating point subtraction

Description:

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0
9_6	Rt_6	Rb_6	Ra_6	$78h_8$	Pn_4	Pc_4

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
~4	Rt_6	Ra_6	$77h_8$	Pn_4 Pc_4

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.

Instruction Format:

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

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
```

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

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

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	
~4	Rt_6	Ra_6	$76h_8$	Pn_4	Pc_4

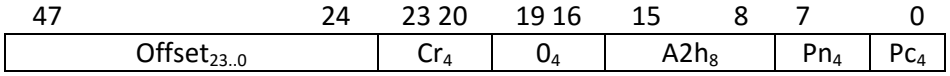
JMP - Jump To Address

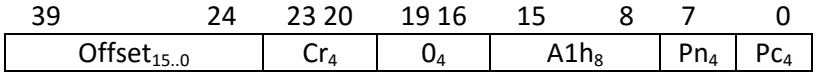
Description:

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:





Operation:

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

JSR - Jump To Subroutine Instruction

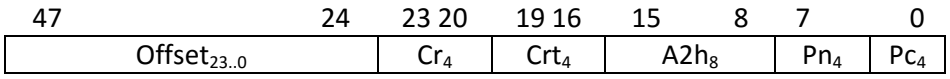
Description:

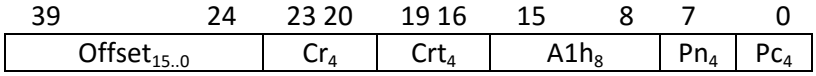
A jump is made to the sum of the zero extended offset supplied in the offset field of the instruction and the specified code address register Cr. The JSR instruction may be used with an immediate predicate constant in order to extend the address range of the jump.

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:





Operation:

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

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.

Instruction Format:

39 28	27 22	21 16	15 8	7 0	
$Immediate_{11..0}$	Rt_6	Ra_6	$80h_8$	Pn_4	Pc_4

Operation:

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

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:

39 28	27 22	21 16	15 8	7 0
$Immediate_{11..0}$	Rt_6	Ra_6	$81h_8$	Pn_4	Pc_4

Operation:

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

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 36	35 34	33 28	27 22	21 16	15 8	7 0	
$\sim_4$	Sc_2	Rt_6	Rb_6	Ra_6	$B1h_8$	Pn_4	Pc_4

Operation:

Rt = mem[Ra+Rb]

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 36	3534	33 28	27 22	21 16	15 8	7 0
$\sim_4$	Sc_2	Rt_6	Rb_6	Ra_6	$B0h_8$	Pn_4	Pc_4

Operation:

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

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:

39 28	27 22	21 16	15 8	7 0
$Displacement_{11..0}$	Rt_6	Ra_6	$82h_8$	Pn_4	Pc_4

Operation:

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

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:

39 28	27 22	21 16	15 8	7 0	
$Displacement_{11..0}$	Rt_6	Ra_6	$83h_8$	Pn_4	Pc_4

Operation:

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

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 36	3534	33 28	27 22	21 16	15 8	7 0	
$\sim_4$	Sc_2	Rt_6	Rb_6	Ra_6	$B3h_8$	Pn_4	Pc_4

Operation:

Rt = mem[Ra + Rb * scale]

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 36	3534	33 28	27 22	21 16	15 8	7 0	
$\sim_4$	Sc_2	Rt_6	Rb_6	Ra_6	$B2h_8$	Pn_4	Pc_4

Operation:

Rt = mem[Ra + Rb * scale]

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

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

Operation:

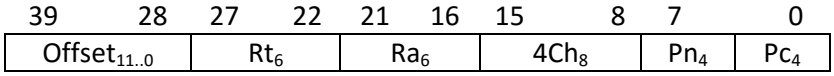
Spr = immediate

LEA – Load Effective Address

Description:

This is an alternate mnemonic for the ADDUI instruction. The memory address is placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

Instruction Format:



Operation:

Rt = Ra+offset

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:

39 28	27 22	21 16	15 8	7 0
$Displacement_{11..0}$	Rt_6	Ra_6	$84h_8$	Pn_4	Pc_4

Operation:

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

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:

39 28	27 22	21 16	15 8	7 0
$Displacement_{11..0}$	Rt_6	Ra_6	$85h_8$	Pn_4	Pc_4

Operation:

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

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 36	3534	33 28	27 22	21 16	15 8	7 0
$\sim_4$	Sc_2	Rt_6	Rb_6	Ra_6	$B5h_8$	Pn_4	Pc_4

Operation:

Rt = mem[Ra+Rb*scale]

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 36	3534	33 28	27 22	21 16	15 8	7 0
$\sim_4$	Sc_2	Rt_6	Rb_6	Ra_6	$B4h_8$	Pn_4	Pc_4

Operation:

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

LOOP – Loop Branch

Description:

A branch is made relative to the current value of the program counter if the loop count register is non-zero. The loop count register is decremented by this instruction. The predicate condition must also be met.

Instruction Format:

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

Operation:

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

LVB – Load Volatile Byte

39 28	27 22	21 16	15 8	7 0
Immediate	Rt	Ra	Opcode	Predicate
$Immediate_{11..0}$	Rt_6	Ra_6	$D0h_8$	Pn_4 Pc_4

Operation:

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

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

LVC – Load Volatile Character

39 32	31 24	23 16	15 8	7 0
Offset	Rt	Ra	Opcode	Predicate
$Offset_{7..0}$	Rt_8	Ra_8	$D1h_8$	Pn_4 Pc_4

Operation:

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

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.

LVH – Load Volatile Half-word

39 32	31 24	23 16	15 8	7 0
Offset	Rt	Ra	Opcode	Predicate
$Offset_{7..0}$	Rt_8	Ra_8	$D2h_8$	Pn_4 Pc_4

Operation:

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

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.

LVW – Load Volatile Word

39 32	31 24	23 16	15 8	7 0
Offset	Rt	Ra	Opcode	Predicate
$Offset_{7..0}$	Rt_8	Ra_8	$D3h_8$	Pn_4 Pc_4

Operation:

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

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.

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:

39 28	27 22	21 16	15 8	7 0
$Displacement_{11..0}$	Rt_6	Ra_6	$86h_8$	Pn_4	Pc_4

Operation:

Rt = mem[Ra + displacement]

LWS – Load Word Special

39 28	27 22	21 16	15 8	7 0	
Immediate	Spr	Ra	Opcode	Predicate	
$Immediate_{11..0}$	Spr_6	Ra_6	$8Eh_8$	Pn_4	Pc_4

Operation:

Spr = mem[Ra+offset]

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.

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 36	35 34	33 28	27 22	21 16	15 8	7 0
$\sim_4$	Sc_2	Rt_6	Rb_6	Ra_6	$B6h_8$	Pn_4 Pc_4

Operation:

Rt = mem[Ra+Rb*scale]

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:

15 — 8	7 — 4	3 — 0
$F9h_8$	Pn_4	Pc_4

MEMSB – Memory Synchronization Barrier

Description:

All memory accesses before the MEMSB command are completed before execution continues. 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
$F8h_8$	Pn_4	Pc_4

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

Operation:

$Rt = Spr_{[n]}$

Special Purpose Registers

Reg #	R/W			
0	R	MID	Machine ID	
1	R	FEAT	Features	
2	R	TICK	Tick count	
3	RW	LC	Loop Counter	
4	RW	PREGS	Predicate register array	
6	RW	ASID	address space identifier	
16-31	RW	CREGS	Code address register array (C0 to C15)	
32-47	RW	SREGS	Segment register array (SEG0-SEG15)	

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
~$_4$	Rt_6	Ra_6	$A7_8$	Pn_4	Pc_4

Operation:

Rt = Ra

MOVS – Move Special Register- Special Register

Description:

This instruction moves one special purpose register to another.

Instruction Format:

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

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

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

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:

39 28	27 22	21 16	15 8	7 0	
$Immediate_{11..0}$	Rt_6	Ra_6	$4Ah_8$	Pn_4	Pc_4

Operation:

Rt = Ra * immediate

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

Operation:

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

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
~$_4$	Rt$_6$	Ra$_6$	70h$_8$	Pn$_4$	Pc$_4$

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.

Instruction Format:

7	0
1_4	0_4

Operation:

<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
$\sim_4$	Rt_6	Ra_6	$71h_8$	Pn_4	Pc_4

Operation:

Rt = ! Ra

OR - Register-Register

Description:

Logically 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

Operation:

Rt = Ra | Rb

ORI - Register-Immediate

Description:

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

Instruction Format:

39 28	27 22	21 16	15 8	7 0	
$Immediate_{11..0}$	Rt_6	Ra_6	$54h_8$	Pn_4	Pc_4

Operation:

Rt = Ra | imm

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.

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$04h_6$	Rt_6	Rb_6	Ra_6	$58h_8$	Pn_4	Pc_4

Operation:

Rt = Ra << Rb

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

Operation:

Rt = Ra << #n

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

Operation:

Rt = Ra >> Rb

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

Operation:

Rt = Ra >> #n

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.

Instruction Format:

15 8	7	0
$F3h_8$	Pn_4	Pc_4

Operation:

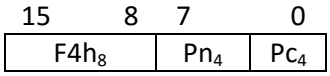
PC = $Cr_{[13]}$
Flags = FlagsBackup

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.

Instruction Format:



Operation:

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

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.

This instruction has a single byte short form that always executes when encountered. For the short form the program counter is loaded from code address register one.

Instruction Formats:

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

Short Form:

7 0	
1_4	1_4

Operation:

$PC = Cr_{[N]}$

Short Form Operation:

$PC = Cr_{[1]}$

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:

39 28	27 22	21 16	15 8	7 0
$Displacement_{11..0}$	Rb_6	Ra_6	$90h_8$	Pn_4	Pc_4

Operation:

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

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 36	3534	33 28	27 22	21 16	15 8	7 0	
$\sim_4$	Sc_2	Rc_6	Rb_6	Ra_6	$C0h_8$	Pn_4	Pc_4

Operation:

memory[Ra+Rb] = Rb

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:

39 28	27 22	21 16	15 8	7 0
$Displacement_{11..0}$	Rb_6	Ra_6	$91h_8$	Pn_4 Pc_4

Operation:

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

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 34	3534	33 28	27 22	21 16	15 8	7 0
Seg_4	Sc_2	Rc_6	Rb_6	Ra_6	$C1h_8$	Pn_4 Pc_4

Operation:

memory[Ra+Rb*scale] = Rb

SEI – Set Interrupt Mask

Description:

The interrupt mask is set, disabling maskable interrupts.

Instruction Format:

15 8	7	0
FBh_8	Pn_4	Pc_4

Operation:

im = 1

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 36	35 28	27 22	21 16	15 8	7 0
Seg_4	$Displacement_{7..0}$	Rb_6	Ra_6	$92h_8$	Pn_4 Pc_4

Operation:

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

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

Operation:

Rt = Ra << Rb

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

Operation:

Rt = Ra << #n

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

Operation:

Rt = Ra << Rb

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

Operation:

Rt = Ra << #n

SHR – Shift Right

Description:

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

Instruction Format:

39 34	33 28	27 22	21 16	15 8	7 0	
$01h_6$	Rt_6	Rb_6	Ra_6	$58h_8$	Pn_4	Pc_4

Operation:

Rt = Ra >> Rb

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

Operation:

Rt = Ra >> #n

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

Operation:

Rt = Ra >> Rb

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:

39 34	33 28	27 22	21 16	15 8	7 0	
$13h_6$	Rt_6	Imm_6	Ra_6	$58h_8$	Pn_4	Pc_4

Operation:

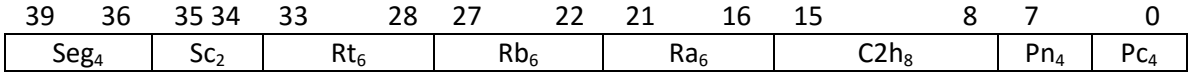
Rt = Ra >> #n

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:



Operation:

memory[Ra+Rb] = Rb

STI – Store Immediate

Description:

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

Instruction Format:

39 32	31 22	21 16	15 8	7 0
$Displacement_{7..0}$	Imm_{10}	Ra_6	$96h_8$	Pn_4	Pc_4

Operation:

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

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

Operation:

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

STSB – Store String Byte

27 22	21 16	15 8	7 0	
Rb	Ra	Opcode	Predicate	
Rb_6	Ra_6	$98h_8$	Pn_4	Pc_4

Operation:

temp = 0
while LC <> 0
    mem[Ra+temp] = $Rb_{[7:0]}$
    temp = temp + 1
    LC = LC – 1

Description:

This instruction stores a byte to consecutive memory locations beginning at the address in Ra until the loop counter reaches zero. This instruction is interruptible.

STSW – Store String Word

27 22	21 16	15 8	7 0	
Rb	Ra	Opcode	Predicate	
Rb_6	Ra_6	$9Ah_8$	Pn_4	Pc_4

Operation:

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

Description:

This instruction stores a word to consecutive memory locations beginning at the address in Ra until the loop counter reaches zero. The memory address contained in Ra must be word aligned. This instruction is interruptible.

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

Operation:

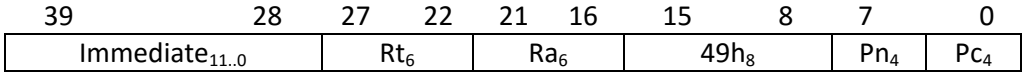
Rt = Ra - Rb

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:



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

Operation:

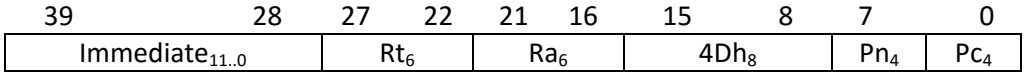
Rt = Ra - Rb

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:



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:

39 28	27 22	21 16	15 8	7 0	
$Immediate_{11..0}$	Rb_6	Ra_6	$93h_8$	Pn_4	Pc_4

Operation:

memory[Ra+offset] = Rb

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

Instruction Format:

39 28	27 22	21 16	15 8	7 0
$Immediate_{11..0}$	Spr_6	Ra_6	$9Eh_8$	Pn_4	Pc_4

Operation:

memory[Ra+offset] = Spr

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 Rb. The memory address must be word aligned.

Instruction Format:

35 34	33 28	27 22	21 16	15 8	7 0	
$\sim_2$	Rc_6	Rb_6	Ra_6	$C3h_8$	Pn_4	Pc_4

Operation:

memory[Ra+Rb] = Rc

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).

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

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.

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

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

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

Operation:

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

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}								ADDI	SUBI	MULI	DIVI	ADDUI	SUBUI	MULUI	DIVUI
5x	{logic}			ANDI	ORI	EORI			{shift}							
6x												_2ADDUI	_4ADDUI	_8ADDUI	_16ADDUI	LDI
7x	NEG	NOT	MUX				ITOF	FTOI	{double}							
8x	LB	LBU	LC	LCU	LH	LHU	LW	LFS	LFD					LEA	LWS	PFLD
9x	SB	SC	SH	SW	SFS	SFD	STI	CAS						LDIS	SWS	CACHE
Ax		JSR	JSR	RTS	LOOP	SYS	INT	MOV			{bitfld}	MOVS				
Bx	LBX	LBUX	LCX	LCUX	LHX	LHUX	LWX									
Cx	SBX	SCX	SHX	SWX			STIX									
Dx																
Ex		NOP							CS:	DS:	SS:	ES:	FS:	GS:		
Fx	{TLB}			RTE	RTI	{BCD}			MEMSB	MEMDB	CLI	SEI				IMM

{RR} Opcodes –$Func_6$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	ADD	SUB	MUL		ADDU	SUBU			2ADDU	4ADDU	8ADDU	16ADDU				
1x																
2x																
3x																

{logic} Opcodes – $Func_6$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	AND		EOR													
1x																
2x																
3x																

{BCD} Opcodes – $Func_6$

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

{double} Opcodes –$Func_6$

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x										FSUB						
1x																
2x																
3x																