

2015

# Thor Guide

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



## Table of Contents

Overview .....	6
Design Objectives.....	6
Programming Model .....	7
General Registers.....	7
Predicates.....	9
Predicate Conditions.....	9
Compiler Usage.....	10
Status Register (SR).....	11
Segmentation.....	12
Software Support.....	12
Address Formation:.....	12
Selecting a segment register .....	12
Non-Segmented Code Area .....	12
Changing the Code Segment.....	13
Segment Usage Conventions .....	13
Power-up State .....	13
TLB.....	14
TLB Registers .....	16
TLBWired (#0h) .....	16
TLBIndex (#1h) .....	16
TLBRandom (#2h).....	16
TLBPageSize (#3h) .....	16
TLBPhysPage (#5h).....	16
TLBVirtPage (#4h).....	17
TLBASID (#7h).....	17
Vectors .....	18
Vector table:.....	18
Hardware Ports .....	19
Instruction Formats.....	20
RR - Register-Register .....	20
RI - Register-Immediate .....	20

CMP Register-Register Compare.....	20
CMPI Register-Immediate Compare .....	20
TST - Register Test Compare .....	20
CTRL- Control .....	20
BR - Relative Branch.....	20
BRK/NOP .....	20
JSR - Jump To Subroutine.....	20
Instruction Set.....	22
2ADDU - Register-Register .....	22
2ADDUI - Register-Immediate.....	23
4ADDU - Register-Register .....	24
4ADDUI - Register-Immediate.....	25
8ADDU - Register-Register .....	26
8ADDUI - Register-Register .....	27
16ADDU - Register-Register .....	28
16ADDUI - Register-Register .....	29
ADD - Register-Register.....	30
ADDI - Register-Immediate .....	31
ADDU - Register-Register .....	32
ADDUI - Register-Immediate.....	33
AND - Register-Register .....	34
ANDI - Register-Immediate .....	35
BCDADD - Register-Register .....	36
BCDMUL - Register-Register .....	37
BCDSUB - Register-Register.....	38
BFCHG – Bitfield Change .....	39
BFCLR – Bitfield Clear .....	40
BFEXT – Bitfield Extract .....	41
BFEXTU – Bitfield Extract Unsigned .....	42
BFINS – Bitfield Insert .....	43
BFSET – Bitfield Set .....	44
BR - Relative Branch.....	45

BRK –Break .....	46
BSR - Branch to Subroutine .....	47
CAS – Compare And Swap .....	49
CLI – Clear Interrupt Mask .....	50
CMP Register-Register Compare .....	51
CMPI Register-Immediate Compare .....	52
EOR - Register-Register .....	53
EORI - Register-Immediate .....	54
FADD – Floating point addition .....	55
FCMP Float Register-Register Compare .....	56
FSUB – Floating point subtraction .....	57
FTOI – Float To Integer .....	58
FTST – Float Register Test Compare .....	59
IMM64,IMM56,IMM48,IMM40,IMM32,IMM24,IMM16.....	60
Immediate Extensions.....	60
INT –Interrupt .....	61
ITOF – Integer to Float .....	62
JMP - Jump To Address .....	63
JSR - Jump To Subroutine Instruction .....	64
LB – Load Byte .....	65
LBX – Load Byte Indexed .....	65
LBU – Load Byte Unsigned .....	66
LBUX – Load Byte Unsigned Indexed .....	67
LC – Load Character .....	69
LCX – Load Character Indexed .....	69
LCU – Load Character Unsigned .....	70
LCUX – Load Character Indexed .....	71
LDI - Load-Immediate .....	73
LDIS - Load-Immediate Special .....	74
LDIT10 - Load-Immediate Top Ten .....	<b>Error! Bookmark not defined.</b>
LEA – Load Effective Address .....	75
LH – Load Half-Word .....	76

LHX – Load Half-word Indexed .....	76
LHU – Load Half-word Unsigned .....	78
LHUX – Load Half-word Unsigned Indexed .....	79
LOOP – Loop Branch .....	81
LVB – Load Volatile Byte .....	82
LVC – Load Volatile Character .....	83
LVH – Load Volatile Half-word .....	84
LVW – Load Volatile Word .....	85
LW – Load Word .....	86
LWS – Load Word Special .....	87
LWX – Load Word Indexed .....	88
MEMDB – Memory Data Barrier .....	89
MEMSB – Memory Synchronization Barrier .....	90
MFSPR – Special Register-Register .....	91
MOV - Register-Register .....	92
MOVS – Move Special Register- Special Register .....	93
MTSPR –Register-Special Register .....	94
MUX – Multiplex .....	95
NEG - Negate Register .....	100
NOP – No Operation .....	101
OR - Register-Register .....	103
ORI - Register-Immediate .....	104
ROL – Rotate Left .....	105
ROLI – Rotate Left by Immediate .....	106
ROR – Rotate Right .....	107
RORI – Rotate Right by Immediate .....	108
RTE – Return From Exception Routine .....	109
<i>Description</i> .....	109
RTI – Return From Interrupt Routine .....	110
RTS – Return from Subroutine .....	111
SB – Store Byte .....	112
SBX – Store Byte Indexed .....	113

SC – Store Character .....	114
SCX – Store Character Indexed .....	115
SEI – Set Interrupt Mask.....	116
SH – Store Half-word.....	117
SHL – Shift Left .....	118
SHLI – Shift Left by Immediate .....	119
SHLU – Shift Left Unsigned.....	120
SHLUI – Shift Left Unsigned by Immediate .....	121
SHR – Shift Right.....	122
SHRI – Shift Right by Immediate .....	123
SHRU – Shift Right Unsigned .....	124
SHRUI – Shift Right Unsigned by Immediate.....	125
SHX – Store Half-word Indexed.....	126
STI – Store Immediate.....	127
STSB – Store String Byte.....	129
STSW – Store String Word.....	<b>Error! Bookmark not defined.</b>
SUB - Register-Register .....	133
SUBI - Register-Immediate .....	134
SUBU - Register-Register.....	135
SUBUI - Register-Immediate .....	136
SW – Store Word.....	137
SWS – Store Word Special.....	138
SWX – Store Word Indexed.....	139
SYS –Call system routine .....	140
TLB – TLB Command.....	142
TST - Register Test Compare .....	144
Opcode Map.....	144

## 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. 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		
	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 eight segment registers. 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

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.

### 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 <sub>12</sub>
=	
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.

### Non-Segmented Code Area

The address range defined as 64'hFxxxxxxxxxxxx (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 #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. Segment register #1 is the data segment (DS) by convention.

## 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'hFFFFFFFFFFFF0. One of the first tasks of the boot program would be to initialize the segment registers to known values. The segment register must be setup to perform data accesses properly.

### Segment Registers

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

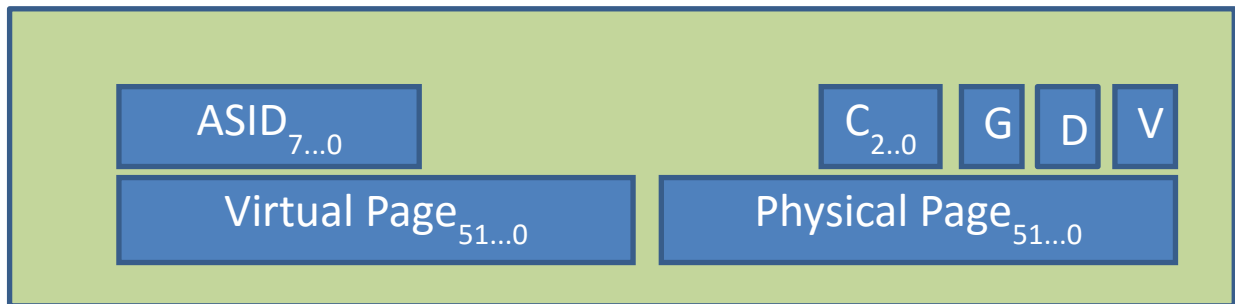
## TLB

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<sub>b</sub>, 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.

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	

## Vectors

The processor vectors to \$FFFFFFFFFFFF0 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	DTLB Miss			
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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>		Ra <sub>6</sub>		Opcode <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### RI - Register-Immediate

39	28	27	22	21	16	15	8	7	0
Immediate <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		Opcode <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### CMP Register-Register Compare

31	28	27	22	21	16	15	12	11	8	7	0
Opc <sub>4</sub>	Rb <sub>6</sub>			Ra <sub>6</sub>		1 <sub>4</sub>		Pt <sub>4</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### CMPI Register-Immediate Compare

31	22	21	16	15	12	11	8	7	0
Immed <sub>9..0</sub>		Ra <sub>6</sub>		2 <sub>4</sub>		Pt <sub>4</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### TST - Register Test Compare

23	22	21	16	15	12	11	8	7	0
O <sub>2</sub>	Ra <sub>6</sub>			O <sub>4</sub>		Pt <sub>4</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### BR - Relative Branch

23	16	15	8	7	0
Disp <sub>7..0</sub>		3 <sub>4</sub>	D <sub>11..8</sub>		Pc <sub>4</sub>

### JSR - Jump To Subroutine

47	24	23	16	15	8	7	0
Offset <sub>23..0</sub>		Cr <sub>4</sub>	Crt <sub>4</sub>	Opcode <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### CTRL- Control

15	8	7	0
Opcode <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### BRK/NOP

7	0
0/1 <sub>4</sub>	O <sub>4</sub>

### RTS

7	0
1 <sub>4</sub>	1 <sub>4</sub>



## 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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		40h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

#### 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 <sub>11..0</sub>			Rt <sub>6</sub>		Ra <sub>6</sub>		6Bh <sub>8</sub>		Pc <sub>4</sub>

### Operation:

$$Rt = Ra * 2 + \text{immediate}$$



## 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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		40h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### 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 <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		6Ch <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Rt = Ra * 4 + \text{immediate}$

## 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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>		Ra <sub>6</sub>		40h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### 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 <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		6Dh <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**Operation:**

$$Rt = Ra * 8 + \text{immediate}$$

## 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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		40h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### 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 <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		6Eh <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**Operation:**

$$Rt = Ra * 16 + \text{immediate}$$

**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 <sub>6</sub>	Rt <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	40h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>					

**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 <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		48h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Rt = Ra + \text{immediate}$



**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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>			Ra <sub>6</sub>	40h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>	

**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 <sub>11..0</sub>			Rt <sub>6</sub>		Ra <sub>6</sub>		4Ch <sub>8</sub>		Pc <sub>4</sub>

31	22	21	16	15	8	7	0
Immediate <sub>9..0</sub>			Rt <sub>6</sub>		47h <sub>8</sub>		Pc <sub>4</sub>

**Operation:**

$Rt = Ra + \text{Immediate}$

## 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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		50h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### 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 <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		53h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		F5h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		F5h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		F5h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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
$\sim_4$	$3_4$	$me_6$		$mb_6$		$Rt_6$		$Ra_6$		$AAh_8$		$Pn_4$	$Pc_4$	



**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:**

4744	43	40	39	34	33	28	27	22	21	16	15	8	7	0
$\sim_4$	$2_4$	$me_6$		$mb_6$		$Rt_6$		$Ra_6$		$AAh_8$		$Pn_4$		$Pc_4$

**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:**

47	44	43	40	39	34	33	28	27	22	21	16	15	8	7	0
$\sim_4$		$4_4$		$me_6$		$mb_6$		$Rt_6$		$Ra_6$		$AAh_8$		$Pn_4$	$Pc_4$

**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:**

47	44	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
$\sim_4$		$1_4$		$me_6$		$mb_6$		$Rt_6$		$Ra_6$		$AAh_8$		$Pn_4$	$Pc_4$

## 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 <sub>7..0</sub>		3h <sub>4</sub>	D <sub>11..8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$PC \leq PC + \text{displacement}$

**BRK -Break****Description:**

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

**Instruction Format:**

7	0
0 <sub>4</sub>	0 <sub>4</sub>

## 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:

47	24	23	20	19	16	15	8	7	0		
Displacement <sub>23..0</sub>				15 <sub>4</sub>		Crt <sub>4</sub>		A2h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

39	24	23	20	1916	15	8	7	0
Displacement <sub>15..0</sub>			15 <sub>4</sub>	Crt <sub>4</sub>	A1h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>



## CACHE – Cache Command

### Description:

This instruction issues a command to the cache.

### Instruction Format:

31	22	21	16	15	8	7	0
Immediate <sub>9..0</sub>		Ra <sub>6</sub>		55h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

### Commands:

Number	
0	Invalidate entire instruction cache
1	Invalidate instruction cache line (address in Ra)
2	Prefetch instruction catch line
256	Invalidate entire data cache
257	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.

### Instruction Format:

47	40	39	34	33	28	27	22	21	16	15	8	7	0
Displacement <sub>7..0</sub>		Rt <sub>6</sub>		Rc <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		97h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### 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 <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>	

### 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
O <sub>4</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	1 <sub>4</sub>	Pt <sub>4</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>		

### 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 at the same time.

### Instruction Format:

31	22	21	16	15	12	11	8	7	0
Immed <sub>10</sub>		Ra <sub>6</sub>		Z <sub>4</sub>		Pt <sub>4</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>	

### 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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		50h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$$Rt = Ra \wedge Rb$$

**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 <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		55h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**Operation:**

$R_t = R_a \wedge \text{immediate}$

**FADD – Floating point addition****Description:****Instruction Format:**

39	34	33	28	27	22	21	16	15	8	7	0
8 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		78h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>



## 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 <sub>4</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	1 <sub>4</sub>	Pt <sub>4</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>		

### 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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		78h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>		Ra <sub>6</sub>		77h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

## 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 <sub>2</sub>	Ra <sub>6</sub>	O <sub>4</sub>	Pt <sub>4</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>

### 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 <sub>63..8</sub>		8 <sub>4</sub>	0 <sub>4</sub>
Immediate <sub>55..8</sub>		7 <sub>4</sub>	0 <sub>4</sub>
Immediate <sub>47..8</sub>		6 <sub>4</sub>	0 <sub>4</sub>
Immediate <sub>39..8</sub>		5 <sub>4</sub>	0 <sub>4</sub>
Immediate <sub>31..8</sub>		4 <sub>4</sub>	0 <sub>4</sub>
Immediate <sub>23..8</sub>		3 <sub>4</sub>	0 <sub>4</sub>
Immediate <sub>15..8</sub>		2 <sub>4</sub>	0 <sub>4</sub>

## 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 <sub>7..0</sub>		Ch <sub>4</sub>		Eh <sub>4</sub>		A6h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>		Ra <sub>6</sub>		76h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

## 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:

47	24	23 20	19 16	15	8	7	0
Offset <sub>23..0</sub>		Cr <sub>4</sub>	O <sub>4</sub>	A2h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

39	24	23 20	19 16	15	8	7	0
Offset <sub>15..0</sub>		Cr <sub>4</sub>	O <sub>4</sub>	A1h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

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



## 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:

47	24	23 20	19 16	15	8	7	0
Offset <sub>23..0</sub>		Cr <sub>4</sub>	Crt <sub>4</sub>	A2h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

39	24	23 20	19 16	15	8	7	0
Offset <sub>15..0</sub>		Cr <sub>4</sub>	Crt <sub>4</sub>	A1h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

23 20	19 16	15	8	7	0
Cr <sub>4</sub>	Crt <sub>4</sub>	A0h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Cr_{[t]} = pc$

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

**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	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>			Rt <sub>6</sub>		Ra <sub>6</sub>		80h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>			Rt <sub>6</sub>		Ra <sub>6</sub>		81h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 37	36	3534	33	28	27	22	21	16	15	8	7	0
Seg <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	B1h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>				

**Operation:**

$Rt = \text{mem}[Ra + Rb]$

**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 37	36	3534	33	28	27	22	21	16	15	8	7	0
Seg <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	B0h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>				

**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	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>	Rt <sub>6</sub>	Ra <sub>6</sub>	82h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>					

### 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	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>	Rt <sub>6</sub>	Ra <sub>6</sub>	83h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>					

**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 37	36	3534	33	28	27	22	21	16	15	8	7	0
Seg <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>		Ra <sub>6</sub>		B3h <sub>8</sub>			Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$$Rt = \text{mem}[Ra + Rb * \text{scale}]$$



## 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 37	36	3534	33	28	27	22	21	16	15	8	7	0
Seg <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>		Ra <sub>6</sub>		B2h <sub>8</sub>			Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Rt = \text{mem}[Ra + Rb * \text{scale}]$

## 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 <sub>9..0</sub>	Rt <sub>6</sub>	6Fh <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>			

### 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 <sub>9..0</sub>		Spr <sub>6</sub>		9Dh <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### 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:

39	28	27	22	21	16	15	8	7	0
Offset <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		4Ch <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Rt = Ra + \text{offset}$

## 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:

3937	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>	Rt <sub>6</sub>		Ra <sub>6</sub>			84h <sub>8</sub>	Pn <sub>4</sub>		Pc <sub>4</sub>

### 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	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>	Rt <sub>6</sub>	Ra <sub>6</sub>	85h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>					

**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 37	36	3534	33	28	27	22	21	16	15	8	7	0
Seg <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>		Ra <sub>6</sub>		B5h <sub>8</sub>			Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Rt = \text{mem}[Ra + Rb * \text{scale}]$



**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 37	36	3534	33	28	27	22	21	16	15	8	7	0
Seg <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>		Ra <sub>6</sub>		B4h <sub>8</sub>			Pn <sub>4</sub>	Pc <sub>4</sub>

**Operation:**

$Rt = \text{sign extend}(\text{mem}[Ra + Rb * \text{scale}])$

## 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 <sub>7..0</sub>	A4h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>		

### 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.

### Instruction Format:

39	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>			Rt <sub>6</sub>		Ra <sub>6</sub>		ACh <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

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

## 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:

39	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>		Displacement <sub>8..0</sub>			Rt <sub>6</sub>		Ra <sub>6</sub>		ADh <sub>8</sub>		Pc <sub>4</sub>

### Operation:

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

## 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 <sub>3</sub>	Displacement <sub>8..0</sub>	Rt <sub>6</sub>		Ra <sub>6</sub>		AEh <sub>8</sub>		Pn <sub>4</sub>		Pc <sub>4</sub>

### Operation:

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

## 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 <sub>3</sub>	Displacement <sub>8..0</sub>	Rt <sub>6</sub>		Ra <sub>6</sub>		AFh <sub>8</sub>		Pn <sub>4</sub>		Pc <sub>4</sub>

### Operation:

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

**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	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>				Rt <sub>6</sub>	Ra <sub>6</sub>		86h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

31	29	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	~	Rt <sub>6</sub>		Ra <sub>6</sub>		D6h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>	

**Operation:**

$Rt = \text{mem}[Ra + \text{displacement}]$

## 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 <sub>3</sub>	Displacement <sub>8..0</sub>	Rt <sub>6</sub>		Ra <sub>6</sub>		8Eh <sub>8</sub>		Pn <sub>4</sub>		Pc <sub>4</sub>

### Operation:

$Spr = mem[Ra + displacement]$



**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	35	34	33	28	27	22	21	16	15	8	7	0
Seg <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>			Rb <sub>6</sub>			Ra <sub>6</sub>		B6h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**Operation:**
$$Rt = \text{mem}[Ra + Rb * \text{scale}]$$

**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	0
F9h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>	

**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 <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>	

## 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)	
32-39	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
$\sim_4$	$Rt_6$	$Ra_6$	$A7_8$	$Pn_4$	$Pc_4$				

### 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
$\sim_4$	Sprt <sub>6</sub>	Spr <sub>6</sub>	AB <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>

**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
$\sim_4$	$\text{Spr}_6$	$\text{Ra}_6$	$\text{A9h}_8$	$\text{Pn}_4$	$\text{Pc}_4$			

**Operation:**

$$\text{Spr}_{[n]} = \text{Ra}$$

## 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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		40h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### 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 <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		4Ah <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$$Rt = Ra * \text{immediate}$$

## 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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		40h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$$Rt = Ra * Rb$$

## 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:

39	28	27	22	21	16	15	8	7	0
Immediate <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		4Eh <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Rt = Ra * \text{immediate}$

## 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 <sub>6</sub>		Rc <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		72h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

For n = 0 to 63

    If Ra<sub>[n]</sub> is set then

        Rt<sub>[n]</sub> = Rb<sub>[n]</sub>

    else

        Rt<sub>[n]</sub> = Rc<sub>[n]</sub>

## 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
$\sim_4$	$Rt_6$	$Ra_6$	$70h_8$	$Pn_4$	$Pc_4$

### 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 <sub>4</sub>	0 <sub>4</sub>

### 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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		50h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		54h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Rt = Ra \ll Rb$

**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 <sub>6</sub>		Rt <sub>6</sub>		Imm <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Rt = Ra \gg Rb$

## 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 <sub>6</sub>		Rt <sub>6</sub>		Imm <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### Operation:

$Rt = Ra \gg \#n$

## 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. Additionally the interrupt mask is cleared to enable interrupts.

### Instruction Format:

15	8	7	0
$F4h_8$	$Pn_4$	$PC_4$	

### 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 <sub>4</sub>	Im <sub>4</sub>	A3h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>	

### Short Form:

7	0
1 <sub>4</sub>	1 <sub>4</sub>

### 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	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>			Rt <sub>6</sub>	Ra <sub>6</sub>		90h <sub>8</sub>			Pn <sub>4</sub>	Pc <sub>4</sub>

**Operation:**

memory[Ra+offset] = Rb<sub>[7..0]</sub>

**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 <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	C0h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>				

**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	37	36	28	27	22	21	16	15	8	7	0
Sg <sub>3</sub>	Displacement <sub>8..0</sub>	Rt <sub>6</sub>	Ra <sub>6</sub>	91h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>					

### Operation:

memory[Ra+displacement] = Rb<sub>[15..0]</sub>

## 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 <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	C1h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>				

### Operation:

memory[Ra+Rb\*scale] = Rc

## SEI – Set Interrupt Mask

### Description:

The interrupt mask is set, disabling maskable interrupts.

### Instruction Format:

15	8	7	0
FB <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>	

### 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 <sub>4</sub>	Displacement <sub>7..0</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	92h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>					

### Operation:

memory[Ra + displacement] = Rb<sub>[31..0]</sub>

**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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**Operation:**

$Rt = Ra \ll Rb$

**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 <sub>6</sub>	Rt <sub>6</sub>			Imm <sub>6</sub>			Ra <sub>6</sub>		58h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**Operation:**

$Rt = Ra \ll Rb$

**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 <sub>6</sub>		Rt <sub>6</sub>		Imm <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>		Rt <sub>6</sub>		Imm <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>		Rt <sub>6</sub>		Imm <sub>6</sub>		Ra <sub>6</sub>		58h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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:**

39 37	36	3534	33	28	27	22	21	16	15	8	7	0
Seg <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	C2h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>				

**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 displacement and register Ra. The memory address must be word aligned.

**Instruction Format:**

39	32	31	22	21	16	15	8	7	0
Displacement <sub>7..0</sub>	Imm <sub>10</sub>	Ra <sub>6</sub>	96h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>				

**Operation:**

memory[Ra + displacement] = zero extend (Imm<sub>[9..0]</sub>)



## 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 <sub>9..6</sub>	Sc <sub>2</sub>	Imm <sub>5..0</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	C6h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>					

### Operation:

memory[Ra + Rb \* scale] = zero extend (Imm<sub>[9..0]</sub>)

## 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 <sub>3</sub>	O <sub>3</sub>	~ <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	98h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>				

### 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:

31 28	27	22	21	16	15	8	7	0
1 <sub>4</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	98h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>			

### Operation:

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

## 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:

31 28	27	22	21	16	15	8	7	0
$Z_4$	$Rb_6$	$Ra_6$	$98h_8$			$Pn_4$	$Pc_4$	

### 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:

31 28	27	22	21	16	15	8	7	0
3 <sub>4</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	98h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>			

### 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 <sub>6</sub>	Rt <sub>6</sub>			Rb <sub>6</sub>		Ra <sub>6</sub>		40h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### 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:**

39	28	27	22	21	16	15	8	7	0
Immediate <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		49h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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 <sub>6</sub>		Rt <sub>6</sub>		Rb <sub>6</sub>		Ra <sub>6</sub>		40h <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

**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:

39	28	27	22	21	16	15	8	7	0
Immediate <sub>11..0</sub>		Rt <sub>6</sub>		Ra <sub>6</sub>		4Dh <sub>8</sub>		Pn <sub>4</sub>	Pc <sub>4</sub>

### 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 <sub>3</sub>	Displacement <sub>8..0</sub>	Rt <sub>6</sub>		Ra <sub>6</sub>		93h <sub>8</sub>		Pn <sub>4</sub>		Pc <sub>4</sub>

### 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 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 <sub>3</sub>	Displacement <sub>8..0</sub>	Spr <sub>6</sub>	Ra <sub>6</sub>	9Eh <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>				

### Operation:

memory[Ra + displacement] = 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 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 <sub>3</sub>	~	Sc <sub>2</sub>	Rc <sub>6</sub>	Rb <sub>6</sub>	Ra <sub>6</sub>	C3h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>				

### Operation:

$\text{memory}[\text{Ra} + \text{Rb}] = \text{Rc}$

## 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	0
F7h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>	

**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 <sub>7..0</sub>	Ch <sub>4</sub>	Dh <sub>4</sub>	A5h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>		

## 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 <sub>6</sub>	Tn <sub>4</sub>	Cmd <sub>4</sub>	F0h <sub>8</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>		

Tn<sub>4</sub> – 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
O <sub>2</sub>	Ra <sub>6</sub>	O <sub>4</sub>	Pt <sub>4</sub>	Pn <sub>4</sub>	Pc <sub>4</sub>	

### 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}							ADDUI	ADDI	SUBI	MULI	DIVI	ADDUI	SUBUI	MULUI	DIVUI
5x	{logic}			ANDI	ORI	EORI			{shift}							
6x												_2ADD UI	_4ADD UI	_8ADD UI	_16ADD UI	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	STS					LDIS	SWS	CACHE
Ax	JSR	JSR	JSR	RTS	LOOP	SYS	INT	MOV	MFSPR	MTSPR	{bitfld}	MOVS	LVB	LVC	LVH	LVW
Bx	LBX	LBUX	LCX	LCUX	LHX	LHUX	LWX									
Cx	SBX	SCX	SHX	SWX				STIX								
Dx							LW									
Ex		NOP														
Fx	{TLB}			RTE	RTI	{BCD}		SYNC	MEMSB	MEMDB	CLI	SEI				IMM

{RR} Opcodes –Func<sub>6</sub>[illegible]

{logic} Opcodes – Func<sub>6</sub>

[illegible]

{BCD} Opcodes – Func<sub>6</sub>

[illegible]{double} Opcodes –Func<sub>6</sub>[illegible]

