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

Thor2023

[Document subtitle]

Robert Finch

Table of Contents

Thor2023

Preface

Who This Book is For

This book describes the Thor2023 ISA. It is for anyone interested in instruction set architectures.

Motivation

The author desired a CPU core supporting 128-bit floating-point operations for the precision. He also wanted a core he could develop himself. The simplest approach to supporting 128-bit floats is to use 128-bit wide registers, which leads to 128-bit wide busses in the CPU and just generally a 128-bit design. It is not the author's goal to develop a 128-bit machine. There are good ways of obtaining 128-bit floating-point precision on 64-bit or even 32-bit machines, but it adds some complexity. Complexity is something the author must manage to get the project done and a flat 128-bit design would be simpler. Efficiency is being traded off for design simplicity. Some of the most efficient designs are 32-bit.

The processor presented here isn't the smallest, most efficient, and fastest RISC processor. It's also not a simple beginner's example. Those weren't my goals. Instead, it offers reasonable performance with an easy-to-understand state machine and hopefully design simplicity. It's also designed around the idea of using a simple compiler. Some operations like multiply and divide could have been left out and supported with software generated by a compiler rather than having hardware support. But I was after a simple compiler design. There's lots of room for expansion in the future. I chose a 64 bit design supporting 128-bit ops in part anticipating more than 4GB of memory available sometime down the road. A 64-bit architecture is doable in FPGA's today, although it uses double or more the resources that a 32-bit design would.

About the Author

First a warning: I'm an enthusiastic hobbyist like yourself, with a ton of experience. I've spent a lot of time at home doing research and implementing several soft-core processors, almost maniacally. One of the first cores I worked on was a 6502 emulation. I then went on to develop the Butterfly32 core. Later the Raptor64. I have about 25 years professional experience working on banking applications at a variety of language levels including assembler. So, I have some real-

world experience developing complex applications. I also have a diploma in electronics engineering technology. Some of the cores I work on these days are too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come. To some extent larger boards have arrived. The author is a bit wary of larger boards. Larger FPGAs increase build times by their nature.

Nomenclature

There has been some mix-up in the naming of load and store instructions as computer systems have evolved. A while ago, a "word" referred to a 16-bit quantity. This is reflected in the mnemonics of instructions where move instructions are qualified with a ".w" for a 16-bit move. Some machines referred to 32-bits as a word. Times have changed and 64-bit workstations are now more common. In the author's parlance a word refers to the word size of a machine, which may be 16, 32, 64 bits or some other size. What does ".w" or ".d", and ".l" refer to? To some extent it depends on the architecture.

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

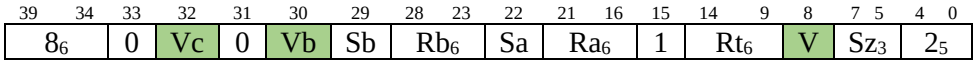
Number of Bits		Instructions	Comment
8	byte	LDB, STB	UTF8 usage
16	wyde	LDW, STW	
32	tetra	LDT, STT	
64	octa	LDO, STO	
128	hexi	LDH, STH	

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

Colorization of Opcodes

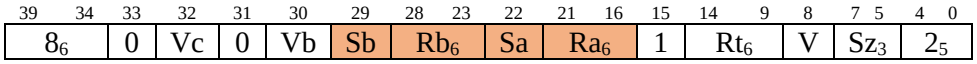
Opcodes are shown in a colorized format. The colors are kept consistent between different types of opcode fields. The bits are not always in the same position. For instance, bits representing the instruction format code are colored in medium green. Colors for a typical opcode are shown below.

NAND Rt, Ra, Rb – Instruction Format Bits



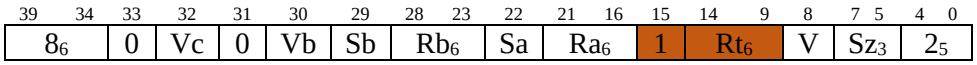
Clock Cycles: 1

NAND Rt, Ra, Rb – Source Register Specifiers



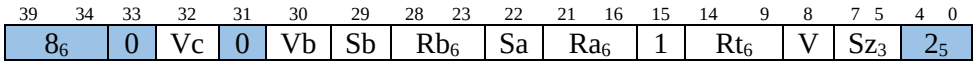
Clock Cycles: 1

NAND Rt, Ra, Rb – Target Register Specifier



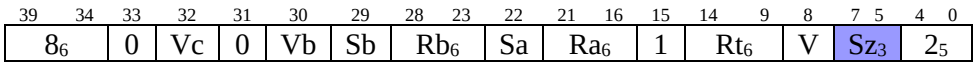
Clock Cycles: 1

NAND Rt, Ra, Rb – Opcode Type Bits



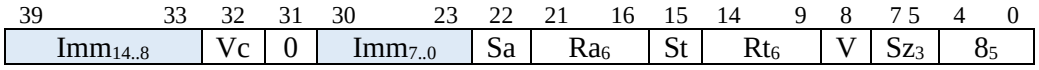
Clock Cycles: 1

NAND Rt, Ra, Rb – Size Code Bits



Clock Cycles: 1

AND Rt,Ra,Imm_{15} – Immediate Constant Bits



Little Endian vs big Endian

One choice to make is whether the architecture is little endian or big endian. There's a never-ending argument by computer folks as to which endian is better. In reality they are about the same or there wouldn't be an argument. In a little-endian architecture, the least significant byte is stored at the lowest memory address. In a big-endian architecture the most significant byte is stored at the lowest memory address. The author is partial to little endian machines; it just seems more natural to him although he knows people who swear by the opposite. Whichever endian is chosen, often the machine has instructions(s) for converting from one endian to the other. The author does not bother with endian conversion; it's a feature that he probably wouldn't use. Some implementations even allow the endian of the machine to be set by the user. This seems like overkill to the author. The endian of data is important because some file types depend on data being in little or big-endian format. Thor is a little-endian machine.

Endian

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

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

Little Endian:

Address	3	2	1	0
Byte	3	2	1	0

Big Endian:

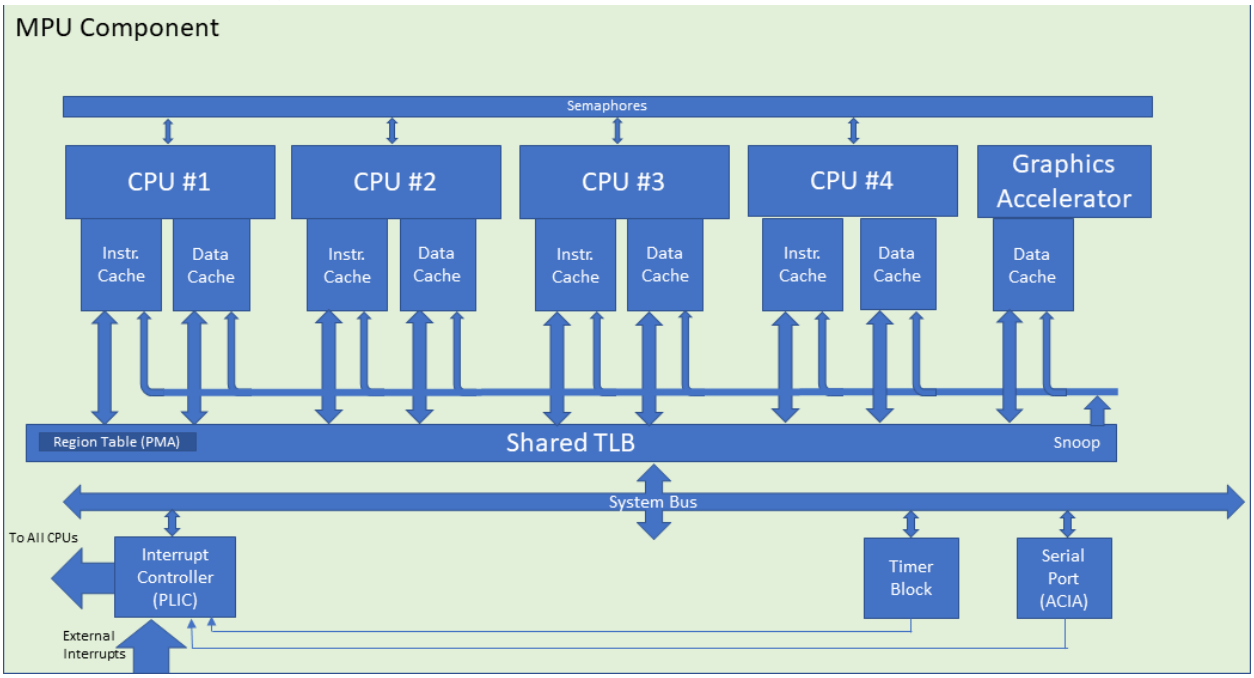
Address	3	2	1	0
Byte	0	1	2	3

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

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

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

Block Diagram



Programming Model
Register File

Rn – General Purpose Registers

The register file contains 64 128-bit general purpose registers. Each register is accessible as two half registers. 128-bit operations require an additional clock cycle to fetch and to store the register value and so are two clocks slower than operations on fewer bits. This is due to the implementation of the register file as a 64-bit wide file that is twice as deep to store the upper half of a register. It conserves space in the register file.

The register file is *unified* and may hold either integer or floating-point values. The stack pointer, register 63, is banked with a separate stack pointer for each operation mode. Registers may be loaded or stored individually or in groups of eight 64-bit values.

Register r53 is special in that when read it refers to the program counter's current value, used to form PC relative addresses. When written it refers to the stack canary register. Attempting to load the register from memory causes a stack canary check instead. The loaded value will be compared against the canary and an exception will occur if they differ.

Register r0 is special in that it always reads as a zero. Note that it may be inverted to read as -1 for some instructions.

Register ABI

Regno	ABI	Group Reg	ABI Usage
0	0	AG0	Always zero
1	A0		First argument / return value register
2	A1		Second argument / return value register
3	A2		Third argument register
4 to 7	A3 to A6		Argument registers
8 to 15	T0 to T7	TG0	Temporary register, caller save
16 to 31	S0 to S15	SG0, SG1	Saved register, register variables
32 to 39	VM0 to VM7	VMG	Vector mask
40 to 43	A7 to A10	G5	Argument register
44 to 47	T8 to T11		Temporaries
48 to 51		G6	unassigned
52	TS		Thread state pointer
53	PC / SC		Program counter; LOAD does canary check
54	CTA		Card table address
55	LC		Loop counter
56	LR0	G7	Subroutine link register #0; branch subroutine specific
57	LR1		Subroutine link register #1; milli-code routines
58	LR2		Subroutine link register #2
59	LR3		Subroutine link register #3
60	GP1		Global Pointer #1 (RO data segment)
61	GP0		Global Pointer #0 (Data segment)
62	FP		Frame Pointer
63	SP		Stack Pointer
63	ASP		Application/User Stack pointer
63	SSP		Supervisor Stack pointer
63	HSP		Hypervisor Stack pointer
63	MSP		Machine stack pointer

Vn – SIMD Registers

The SIMD register file contains 32 512-bit registers.

Regno	ABI	ABI Usage
0		
1	VA0	First argument / return value
2	VA1	Second argument / return value
3	VA2	Third argument
4 to 15	VT0 to VT11	
16 to 31	VS0 to VS15	
32 to 40		
40 to 47	VA3 to VA10	
48 to 63		

Predicate Registers

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

Mask Registers (vm0 to vm7)

Mask registers are used to mask off vector operations so that a vector instruction doesn't perform the operation on all elements of the vector. Vector instructions (loads and stores) that don't explicitly specify a mask register assume the use of mask register zero (vm0). Mask registers are a subset of the general-purpose register array, allowing instructions that operate on GPRs to operate on the mask registers. Potentially any register could be used as a mask register, the compiler will assign a register as needed. Vm0 to vm7 are just a suggestion of registers to reserve for vector masking.

Mask register specification allows the mask register to be used in an inverted form. This can be applied to r0 which will then enable all lanes of execution.

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

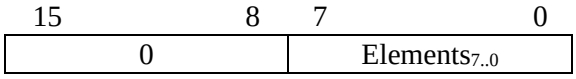
Register	Tag	Usage
vm0	32	
vm1	33	
vm2	34	
vm3	35	
vm4	36	
vm5	37	
vm6	38	
vm7	39	

Vector Length (VL register)

The vector length register controls how many elements of a vector are processed. The vector length register may not be set to a value greater than the number of elements supported by

hardware. After the vector length is set a SYNC instruction should be used to ensure that following instructions will see the updated version of the length register.

Vector length has register tag #87.



Code Address Registers

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

> *The original Thor lumped these registers together in a code address register array. For Thor2023 some of these registers are now part of the general register file.*

It is possible to do an indirect method call using any register.

LRn – Link Registers

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

PC – Program Counter

This register points to the currently executing instruction. The program counter increments as instructions are fetched, unless overridden by another flow control instruction. The program counter may be set to any byte address. There is no alignment restriction. It is possible to write position independent code, PIC, using PC relative addressing.

LC - Loop Counter (reg 55)

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

SR - Status Register (CSR 0x?004)

The processor status register holds bits controlling the overall operation of the processor, state that needs to be saved and restored across interrupts. The bits have individual bit set / clear capability using the CSRRS, CSRRC instructions. Only the user interrupt enable bit is available in user mode, other bits will read as zero.

Bit		Usage
0	uie	User interrupt enable
1	sie	Supervisor interrupt enable
2	hie	Hypervisor interrupt enable
3	mie	Machine interrupt enable
4	die	Debug interrupt enable
5 to 7	ipl	Interrupt level
8	ssm	Single step mode
9	te	Trace enable
10 to 11	om	Operating mode
12 to 13	ps	Pointer size
14 to 15	~	reserved
16	mprv	memory privilege
17	~	reserved
18	dmi	~~Decimal mode for integers~~
19	dmf	~~Decimal mode for float~~
20 to 23	~	reserved
24 to 31	cpl	Current privilege level

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

T indicates that trace mode is active.

OM processor operating mode.

PS: indicates the size of pointers in use. This may be one of 32, 64 or 128 bits.

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

IPL is the interrupt mask level

RT specifies the return type for an RTI instruction.

MPRV Memory Privilege, indicates to use previous operating mode for memory privileges

Decimal Mode

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

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

Decimal mode is now handled on an instruction-by-instruction basis with bits in the instruction indicating when decimal mode is in use.

Special Purpose Registers

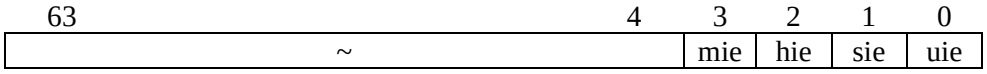
SC - Stack Canary (GPR 53)

This special purpose register is available in the general register file as register 53. The stack canary register is used to alleviate issues resulting from buffer overflows on the stack. The canary register contains a random value which remains consistent throughout the run-time of a program. In the right conditions, the canary register is written to the stack during the function's prolog code. In the function's epilog code, the value of the canary on stack is checked to ensure it is correct, if not a check exception occurs.

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

See status register.

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



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

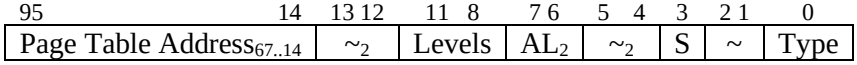
This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code to determine what to do. Only the low order 12 bits are implemented. The high order bits read as zero and are not updateable.

[U/S/H/M]_SCRATCH – CSR 0x?041

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

S_PTBR (CSR 0x1003)

This register contains the base address of the page table, which must be a multiple of 16384. Also included in this register is table parameters depth and type. Register tag #152.



Type: 0 = inverted page table, 1 = page table

S: 1=software managed TLB miss, 0 = hardware table walking

Levels are ignored for the inverted page table. For a normal page table gives the top entry level.
AL_2: TLB entry replacement algorithm, 0=fixed,1=LRU,2=random,3=reserved

S_ASID (CSR 0x101F)

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

S_KEYS (CSR 0x1020 to 0x1027)

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

			23 … 0
1020			key0
1021			key1
…			…
1027			key7

M_CORENO (CSR 0x3001)

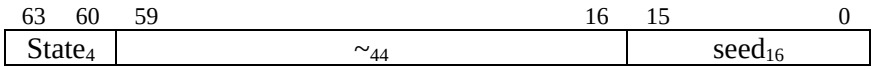
This register contains a number that is externally supplied on the coreno_i input bus to represent the hardware thread id or the core number. It should be non-zero.

M_TICK (CSR 0x3002)

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

M_SEED (CSR 0x3003)

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



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

M_BADADDR (CSR 0x3007)

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

M_BAD_INSTR (CSR 0x300B)

This register contains a copy of the exceptioned instruction.

M_SEMA (CSR 0x300C)

This register contains semaphores. The semaphores are shared between all cores in the MPU.

M_TVEC – CSR 0x3030 to 0x3034

These registers contain the address of the exception handling routine for a given operating level. TVEC[4] (0x3034) is used directly by hardware to form an address of the debug routine. The lower eight bits of TVEC[3] are not used. The lower bits of the exception address are determined from the operating level. TVEC[0] to TVEC[2] are used by the REX instruction.

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

Reg #	
0x3030	TVEC[0] – user mode
0x3031	TVEC[1] - supervisor mode
0x3032	TVEC[2] – hypervisor mode
0x3033	TVEC[3] – machine mode
0x3034	TVEC[4] - debug

M_SR_STACK (CSR 0x303C to CSR 0x303D)

This pair of registers contains a stack of the status register which is pushed during exception processing and popped on return from interrupt. There are only eight slots as that is the maximum nesting depth for interrupts.

	127 96	95 64	63 32	31 0
0x303C	SR3	SR2	SR1	SR0
0x303D	SR7	SR6	SR5	SR4

M_IOS – IO Select Register (CSR 0x3100)

The location of IO is determined by the contents of the IOS control register. The select is for a 1MB region. This address is a virtual address. The low order 16 bits of this register should be zero and are ignored.

63 16	15 0
Virtual Address$_{67..20}$	0_{16}

M_EPC (CSR 0x3108 to 0x310F)

This set of registers contains the address stack for the program counter used in exception handling.

Reg #	Name
0x3108	EIP0
…	
0x310F	EIP7

AV – Application Vector Table Address

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

63 0
App Vector Table Address$_{67..4}$

VB – Vector Base Register

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

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

Operating Modes

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

A subset of instructions is limited to machine mode.

Mode Bits	Mode
0	User / App
1	Supervisor
2	Hypervisor
3	Machine

Exceptions

External Interrupts

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

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

Effect on Machine Status

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

Exception Stack

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

Exception Table

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

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

Reset

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

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

Precision

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

Memory Management

Bank Swapping

About the simplest form of memory management is a single bank register that selects the active memory bank. This is the mechanism used on many early microcomputers. The bank register may be an eight bit I/O port supplying control over some number of upper address bits used to access memory.

The Page Map

The next simplest form of memory management is a single table map of virtual to physical addresses. The page map is often located in a high-speed dedicated memory. An example of a mapping table is the 74LS612 chip. It may map four address bits on the input side to twelve address bits on the output side. This allows a physical address range eight bits greater than the virtual address range. A more complicated page map is something like the MC6829 MMU. It may map 2kB pages in a 2MB physical address space for up to four different tasks.

Regions

In any processing system there are typically several different types of storage assigned to different physical address ranges. These include memory mapped I/O, MMIO, DRAM, ROM, configuration space, and possibly others. Thor2023 has a region table that supports up to eight separate regions.

The region table is a list of region entries. Each entry has a start address, an end address, an access type field, and a pointer to the PMT, page management table. To determine legal access types, the physical address is searched for in the region table, and the corresponding access type returned. The search takes place in parallel for all eight regions.

Once the region is identified the access rights for a particular page within the region can be found from the PMT corresponding to the region. Global access rights for the entire region are also specified in the region table. These rights are gated with value from the PMT and TLB to determine the final access rights.

PMA - Physical Memory Attributes Checker

Overview

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

Physical memory attributes are stored in an eight-entry region table. Three bits in the PTE select an entry from this table. The operating mode of the CPU also determines which 32-bit set of attributes to apply for the memory region.

Most of the entries in the table are hard-coded and configured when the system is built. However, they may be modified at the address range $F…F9F0xxx.

Physical memory attributes checking is applied in all operating modes.

The region table is accessible as a memory mapped IO, MMIO, device.

Region Table Description

Reg	Bits		
00	128	Pmt	associated PMT address
01	128	cta	Card table address
02	128	at	Four groups of 32-bit memory attributes, 1 group for each of user, supervisor, hypervisor and machine.
03	128	…	Not used
04 to 1F		…	7 more register sets

PMT Address

The PMT address specifies the location of the associated PMT.

CTA – Card Table Address

The card table address is used during the execution of the store pointer, STPTR instruction to locate the card table.

Attributes

Bitno		
0	X	may contain executable code
1	W	may be written to

<table>
<tr><td>2</td><td>R</td><td>may be read</td></tr>
<tr><td>3</td><td>~</td><td>reserved</td></tr>
<tr><td>4-7</td><td>C</td><td>Cache-ability bits</td></tr>
<tr><td>8-10</td><td>G</td><td>granularity
<table>
<tr><td>G</td><td></td></tr>
<tr><td>0</td><td>byte accessible</td></tr>
<tr><td>1</td><td>wyde accessible</td></tr>
<tr><td>2</td><td>tetra accessible</td></tr>
<tr><td>3</td><td>octa accessible</td></tr>
<tr><td>4</td><td>hexi accessible</td></tr>
<tr><td>5 to 7</td><td>reserved</td></tr>
</table></td></tr>
<tr><td>11</td><td>~</td><td>reserved</td></tr>
<tr><td>12-14</td><td>S</td><td>number of times to shift address to right and store for telescopic STPTR stores.</td></tr>
<tr><td>16-23</td><td>T</td><td>device type (rom, dram, eeprom, I/O, etc)</td></tr>
<tr><td>24-31</td><td>~</td><td>reserved</td></tr>
</table>

Page Management Table - PMT

Overview

For the first translation of a virtual to physical address, after the physical page number is retrieved from the TLB, the region is determined, and the page management table is referenced to obtain the access rights to the page. PMT information is loaded into the TLB entry for the page translation. The PMT contains an assortment of information most of which is managed by software. Pieces of information include the key needed to access the page, the privilege level, and read-write-execute permissions for the page. The table is organized as rows of access rights table entries (PMTEs). There are as many PMTEs as there are pages of memory in the region.

For subsequent virtual to physical address translations PMT information is retrieved from the TLB.

As the page is accessed in the TLB, the TLB may update the PMT.

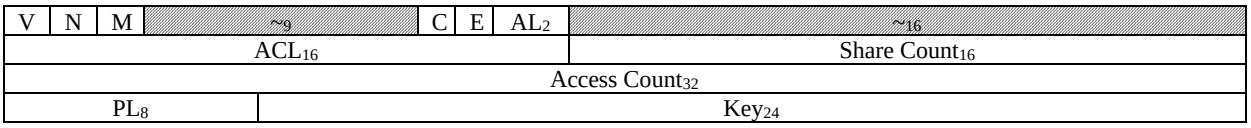
Location

The page management table is in main memory and may be accessed with ordinary load and store instructions. The PMT address is specified by the region table.

PMTE Description

There is a wide assortment of information that goes in the page management table. To accommodate all the information an entry size of 128-bits was chosen.

Page Management Table Entry



Access Control List

The ACL field is a reference to an associated access control list.

Share Count

The share count is the number of times the page has been shared to processes. A share count of zero means the page is free.

Access Count

This part uses the term ‘access count’ to refer to the number of times a page is accessed. This is usually called the reference count, but that phrase is confusing because reference counting may also refer to share counts. So, the phrase ‘reference count’ is avoided. Some texts use the term reference count to refer to the share count. Reference counting is used in many places in software and refers to the number of times something is referenced.

Every time the page of memory is accessed, the access count of the page is incremented. Periodically the access count is aged by shifting it to the right one bit.

The access count may be used by software to help manage the presence of pages of memory.

Key

The access key is a 24-bit value associated with the page and present in the key ring of processes. The keyset is maintained in the keys CSRs. The key size of 20 bits is a minimum size recommended for security purposes. To obtain access to the page it is necessary for the process to have a matching key OR if the key to match is set to zero in the PMTE then a key is not needed to access the page.

Privilege Level

The current privilege level is compared with the privilege level of the page, and if access is not appropriate then a privilege violation occurs. For data access, the current privilege level must be at least equal to the privilege level of the page. If the page privilege level is zero anybody can access the page.

N

indicates a conforming page of executable code. Conforming pages may execute at the current privilege level. In which case the PL field is ignored.

M

indicates if the page was modified, written to, since the last time the M bit was cleared. Hardware sets this bit during a write cycle.

E

indicates if the page is encrypted.

AL

indicates the compression algorithm used.

C

The C indicator bit indicates if the page is compressed.

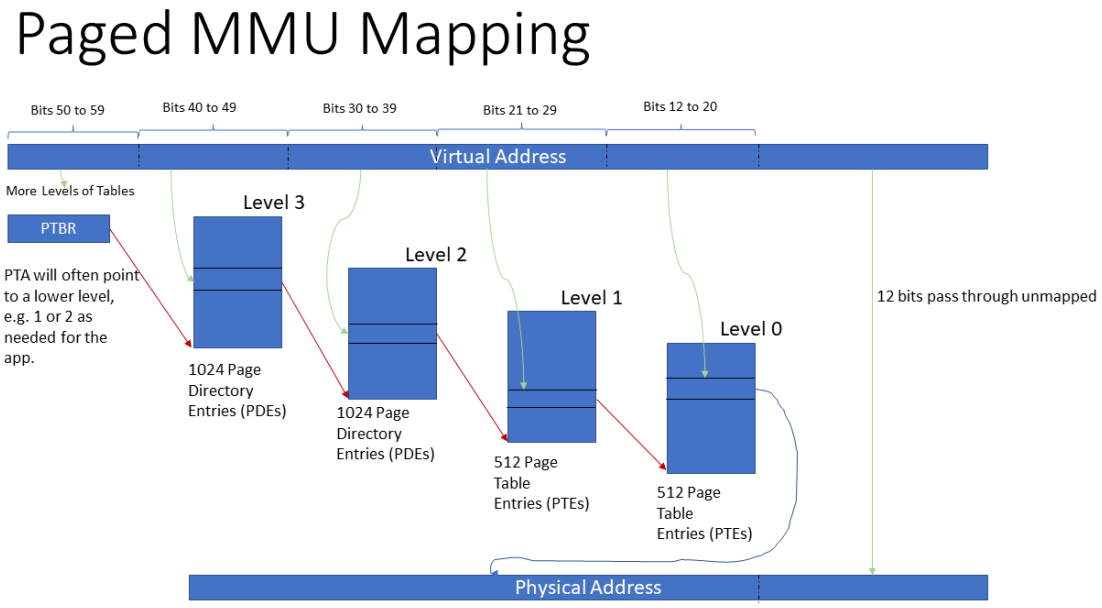
Page Tables

Intro

Page tables are part of the memory management system used map virtual addresses to real physical addresses. There are several types of page tables. Hierarchical page tables are probably the most common. Almost all page tables map only the upper bits of a virtual address, called a page. The lower bits of the virtual address are passed through without being altered. The page size often 4kB which means the low order 12-bits of a virtual address will be mapped to the same 12-bits for the physical address.

Hierarchical Page Tables

Hierarchical page tables organize page tables in a multi-level hierarchy. They can map the entire virtual address range but often only a subrange of the full virtual address space is mapped. This can be determined on an application basis. At the topmost level a register points to a page directory, that page directory points to a page directory at a lower level until finally a page directory points to a page containing page table entries. To map an entire 64-bit virtual address range approximately five levels of tables are required.

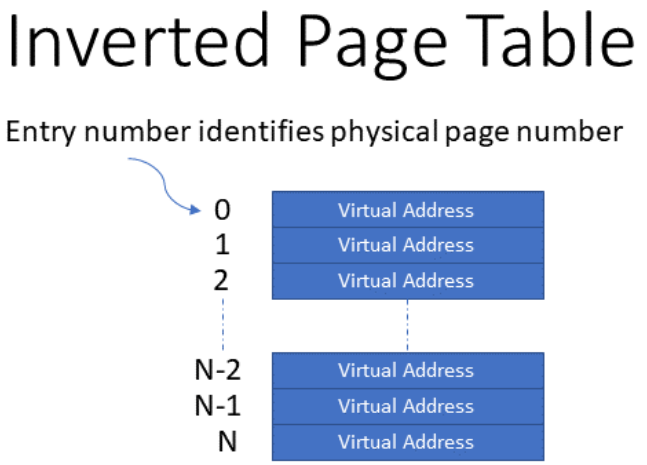


Inverted Page Tables

An inverted page table is a table used to store address translations for memory management. The idea behind an inverted page table is that there are a fixed number of pages of memory no matter how it is mapped. It should not be necessary to provide for a map of every possible address, which is what the hierarchical table does, only addresses that correspond to real pages of memory need be mapped. Each page of memory can be allocated only once. It is either allocated or it is not. Compared to a non-inverted paged memory management system where tables are used to map potentially the entire address space an inverted page table uses less memory. There is typically only a single inverted page table supporting all applications in the system. This is a different approach than a non-inverted page table which may provide separate page tables for each process.

The Simple Inverted Page Table

The simplest inverted page table contains only a record of the virtual address mapped to the page, and the index into the table is used as the physical page number. There are only as many entries in the inverted page table as there are physical pages of memory. A translation can be made by scanning the table for a matching virtual address, then reading off the value of the table index. The attraction of an inverted page table is its small size compared to the typical hierarchical page table. Unfortunately, the simplest inverted page table is not practical when there are thousands or millions of pages of memory. It simply takes too long to scan the table. The alternative solution to scanning the table is to hash the virtual address to get a table index directly.



Hashed Page Tables

Hashed Table Access

Hashes are great for providing an index value immediately. The issue with hash functions is that they are just a hash. It is possible that two different virtual address will hash to the same value. What is then needed is a way to deal with these hash collisions. There are a couple of different methods of dealing with collisions. One is to use a chain of links. The chain has each link in the chain pointing the to next page table entry to use in the event of a collision. The hash page table is slightly more complicated then as it needs to store links for hash chains. The second method is to use open addressing. Open addressing calculates the next page table entry to use in the event of a collision. The calculation may be linear, quadratic or some other function dreamed up. A linear probe simply chooses the next page table entry in succession from the previous one if no match occurred. Quadratic probing calculates the next page table entry to use based on squaring the count of misses.

Clustered Hash Tables

A clustered hash table works in the same manner as a hashed page table except that the hash is used to access a cluster of entries rather than a single entry. Hashed values may map to the same cluster which can store multiple translations. Once the cluster is identified, all the entries are searched in parallel for the correct one. A clustered hash table may be faster than a simple hash table as it makes use of parallel searches. Often accessing memory returns a cache line regardless of whether a single byte or the whole cached line is referenced. By using a cache line to store a cluster of entries it can turn what might be multiple memory accesses into a single access. For example, an ordinary hash table with open addressing may take up to 10 memory accesses to find the correct translation. With a clustered table that turns into 1.25 memory accesses on average.

Shared Memory

Another memory management issue to deal with is shared memory. Sometimes applications share memory with other apps for communication purposes, and to conserve memory space where there are common elements. The same shared library may be used by many apps running in the system. With a hierarchical paged memory management system, it is easy to share memory, just modify the page table entry to point to the same physical memory as is used by another process. With an inverted page table having only a single entry for each physical page is not sufficient to support shared memory. There needs to be multiple page table entries available for some physical pages but not others because multiple virtual addresses might map to the same physical address. One

solution would be to have multiple buckets to store virtual addresses in for each physical address. However, this would waste a lot of memory because much of the time only a single mapped address is needed. There must be a better solution. Rather than reading off the table index as the physical page number, the association of the virtual and physical address can be stored. Since we now need to record the physical address multiple times the simple mechanism of using the table index as the physical page number cannot be used. Instead, the physical page number needs to be stored in the table in addition to the virtual page number.

That means a table larger than the minimum is required. A minimally sized table would contain only one entry for each physical page of memory. So, to allow for shared memory the size of the table is doubled. This smells like a system configuration parameter.

Specifics: Thor2023 Page Tables

Thor2023 Hash Page Table Setup

Hash Page Table Entries - HPTE

We have determined that a page table entry needs to store both the physical page number and the virtual page number for the translations. To keep things simple, the page table stores only the information needed to perform an address translation. Other bits of information are stored in a secondary table called the page management table, PMT. The author did a significant amount of juggling around the sizes of various fields, mainly the size of the physical and virtual page numbers. Finally, the author decided on a 192-bit HPTE format.

V	LVL/BC$_5$	RGN$_3$	M	A	T	S	G	SW$_2$	CACHE$_4$	MRWX$_3$	HRWX$_3$	SRWX$_3$	URWX$_3$
PPN$_{31..0}$													
PPN$_{63..32}$													
VPN$_{37..6}$													
VPN$_{69..38}$													
~$_4$		ASID$_{11..0}$						~$_2$		VPN$_{83..70}$			

Fields Description

V	1	translation Valid
G	1	global translation
RGN	3	region
PPN	64	Physical page number
VPN	84	Virtual page number
RWX	3	readable, writeable, executable
ASID	12	address space identifier
LVL/BC	5	bounce count
M	1	modified

A	1	accessed
T	1	PTE type (not used)
S	1	Shared page indicator
SW	3	OS usage

The page table does not include everything needed to manage pages of memory. There is additional information such as share counts and privilege levels to take care of, but this information is better managed in a separate table.

Small Hash Page Table Entries - SHPTE

The small HPTE is used for the test system which contains only 512MB of physical RAM to conserve hardware resources. The SHPTE is 96-bits in size.

V	LVL/BC_5	RGN_3	M	A	T	S	G	SW_2	$CACHE_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$VPN_{5..0}$		$PPN_{25..0}$											
$ASID_{11..0}$						$VPN_{25..6}$							

Page Table Groups – PTG

We want the search for translations to be fast. That means being able to search in parallel. So, PTEs are stored in groups that are searched in parallel for translations. This is sometimes referred to as a clustered table approach. Access to the group should be as fast as possible. There are also hardware limits to how many entries can be searched at once while retaining a high clock rate. So, the convenient size of 1024 bits was chosen as the amount of memory to fetch.

A page table group then contains ten SHPTE page-table entries or five HPTE entries. All entries in the group are searched in parallel for a match. Note that the entries are searched as the PTG is loaded, so that the PTG group load may be aborted early if a matching PTE is found before the load is finished.

127 0

PTE0
PTE1
PTE2
PTE3
PTE4
PTE5
PTE6
PTE7

Size of Page Table

There are several conflicting elements to deal with, with regards to the size of the page table. Ideally, the hash page table is small enough to fit into the block RAM resources available in the FPGA. It may be practical to store the hash page table in block RAM as there would be only a single table for all apps in the system. This probably would not be practical for a hierarchical table.

About 1/3 of the block RAMs available are dedicated to MMU use. At the same time a multiple of the number of physical pages of memory should be supported to support page sharing and swapping pages to secondary storage. To support swapping pages, double the number of physical entries were chosen. To support page sharing, double that number again. Therefore, a minimum size of a page table would contain at least four times the number of physical pages for entries. By setting the size of the page table instead of the size of pages, it can be worked backwards how many pages of memory can be supported.

For a system using 512k block RAM to store PTEs. 512k / 96 = 5,461 entries. 5,461 / 4 = 1,365 physical pages. Since the RAM size is 512MB, each page would be 512MB/1,365 = 393kB. Rounding up, 512kB. Since half the pages may be in secondary storage, 1GB of address range is available. A 512kB page is probably too large to be useful, so either more block RAM is required, or the table could be place in main memory.

Since there are 5,461 entries in the table and they are grouped into groups of ten, there are 546 PTGs. To get to a page table group fast a hash function is needed then that returns a 10-bit number.

Hash Function

The hash function needs to reduce the size of a virtual address down to a 11-bit number. The asid should be considered part of the virtual address. Including the asid an address is 76 bits. The first thing to do is to throw away the lowest fourteen bits as they pass through the MMU unaltered. We now have 62-bits to deal with. We can probably throw away some high order bits too, as a process is not likely to use the full 64-bit address range.

The hash function chosen uses the asid combined with virtual address bits 18 to 28 and bits 29 to 39. This should space out the PTEs according to the asid. Address bits 16 and 17 select one of four address ranges. the PTG supports ten PTEs. The translations where address bits 16 and 17 are involved are likely consecutive pages that would show up in the same PTG. The hash is the asid exclusively or'd with address bis 18 to 28 exclusively or'd with address bits 29 to 39.

Collision Handling

Quadratic probing of the page table is used when a collision occurs. The next PTG to search is calculated as the hash plus the square of the miss count. On the first miss the PTG at the hash plus one is searched. Next the PTG at the hash plus four is searched. After that the PTG at the hash plus nine is searched, and so on.

Finding a Match

Once the PTG to be searched is located using the hash function, which PTE to use needs to be sorted out. The match operation must include both the virtual address bits and the asid, address space identifier, as part of the test for a match. It is possible that the same virtual address is used by two or more different address spaces, which is why it needs to be in the match.

Locality of Reference

The page table group may be cached in the system read cache for performance. It is likely that the same PTG group will be used multiple times due to the locality of reference exhibited by running software.

Access Rights

To avoid duplication of data the access rights are stored in another table called the PMT for access rights table. The first time a translation is loaded the access rights are looked-up from the PMT. A bit is set in the TLB entry indicating that the access rights are valid. On subsequent translations the access rights are not looked up, but instead they are read from values cached in the TLB.

Thor2023 Hierarchical Page Table Setup

Page Table Entries - PTE

For hierarchical tables the structure is like that of hashed page tables except that there is no need to store the virtual address. We know the virtual address because it is what is being translated and there is no chance of collisions unlike the hash table. The structure is 96 bits in size. This allows 1024 PTEs to fit into an 16kB page. ¼ of the 16kB page is not used. Note the size of pages in the table is a configuration parameter used to build the system.

There are two types of page table entries. The first type, T=0, is a pointer to a page of memory, the second type, T=1, is an entry that points to lower-level page tables. PTE's that point to lower-level page tables are sometimes called page table pointers, PTPs.

Page Table Entry Format – PTE

V	LVL/BC$_5$	RGN$_3$	M	A	T	S	G	SW$_2$	CACHE$_4$	MRWX$_3$	HRWX$_3$	SRWX$_3$	URWX$_3$
PPN$_{31..0}$													
PPN$_{63..32}$													

Small Page Table Entry Format – SPTE

The small PTE format is used when the physical address space is less than 46-bits in size. The small PTE occupies only 64-bits. 2048 SPTEs will fit into an 16kB page.

V	LVL/BC$_5$	RGN$_3$	M	A	T	S	G	SW$_2$	CACHE$_4$	MRWX$_3$	HRWX$_3$	SRWX$_3$	URWX$_3$
PPN$_{31..0}$													

Field	Size	Purpose
PPN	64	Physical page number
URWX	3	User read-write-execute override
SRWX	3	Supervisor read-write-execute override
HRWX	3	Hypervisor read-write-execute override
MRWX	3	Machine read-write-execute override
CACHE	4	Cache-ability bits
A	1	1=accessed/used
M	1	1=modified
V	1	1 if entry is valid, otherwise 0
S	1	1=shared page
G	1	1=global, ignore ASID
T	1	0=page pointer, 1= table pointer
RGN	3	Region table index
LVL/BC	5	the page table level of the entry pointed to

Super Pages

The hierarchical page table allows “super pages” to be defined. These pages bypass lower levels of page tables by using an entry at a high level to represent a block containing many pages.

Normally a PTE with LVL=0 is a pointer to an 16kB memory page. However, super-pages may be defined by specifying a page pointer with a LVL greater than zero. For instance, if T=0 and LVL=1 then the page pointed to is a super-page within an 16MB block of contiguous memory.

T=0, LVL=	Page Size
0	16 kB page
1	16 MB page
2	16 GB page
3	16 TB page
4	16 EB page
5	
6	
7	reserved

A super page pointer contains both a pointer to the block of pages and a super page length field. The length field is provided to restrict memory access to an address range between the super page pointer and the super page pointer plus the number of pages specified in the length. A typical use would be to point to the system ROM which may be several megabytes and yet shorter than the maximum size of the super page.

For example, a system ROM is located 512 MB before the end of physical memory. The ROM is only 1MB in size. So, it is desired to setup a super page pointer to the ROM and restrict access to a single megabyte. The PTE for this would look like:

V	1_5	RGN_3	M	A	0	S	G	SW_2	$\sim_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$PPN=0x3FFFE0_{22}$										$NPG=0x03F_{10}$			
$PPN=0xFFFFFFFF_{63..32}$													

The PTE would be pointed to by a LVL=1 pointer resulting in a 16MB super-page size. 512MB is 32 pages before the end of memory, reflected in the value $0x3FFFE0_{22}$ for the PPN above. There are 64 x 16kB pages in 1MB so the length field, NPG, is set to $0x03f_{10}$.

PTE Format for 16MB page

V	1_5	RGN_3	M	A	0	S	G	SW_2	$\sim_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$PPN_{31..10}$										NPG_{10}			
$PPN_{63..32}$													

PTE Format for 16GB page

V	2_5	RGN_3	M	A	0	S	G	SW_2	$\sim_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$PPN_{31..20}$					NPG_{20}								
$PPN_{63..32}$													

TLB – Translation Lookaside Buffer

Overview

A simple page map is limited in the translations it can perform because of its size. The solution to allowing more memory to be mapped is to use main memory to store the translations tables.

However, if every memory access required two or three additional accesses to map the address to a final target access, memory access would be quite slow, slowed down by a factor or two or three, possibly more. To improve performance, the memory mapping translations are stored in another unit called the TLB standing for Translation Lookaside Buffer. This is sometimes also called an address translation cache ATC. The TLB offers a means of address virtualization and memory protection. A TLB works by caching address mappings between a real physical address and a virtual address used by software. The TLB deals with memory organized as pages. Typically, software manages a paging table whose entries are loaded into the TLB as translations are required.

The TLB is a cache specialized for address translations. Thor2023's TLB is quite large being six-way associative with 1024 entries per way. This choice of size was based on the minimum number of block RAMs that could be used to implement the TLB. On a TLB miss the page table is searched for a translation and if found the translation is stored in one of the ways of the TLB. The way selected is determined either randomly or in a least-recently-used fashion as one of the first four ways. The last way may not be updated automatically by a page table search, it must be updated by software.

Size / Organization

The TLB has 1024 entries per set. The size was chosen as it is the size of one block ram for 32-bit data in the FPGA. This is quite a large TLB. Many systems use smaller TLBs. Typically, systems vary between 64 and 1024 entries. There is not really a need for such a large one, however it is available.

The TLB is organized as a six-way set associative cache. The last way may only be updated by software. The last way allows translations to be stored that will not be overwritten. The first four ways may use hardware LRU replacement in addition to fixed or random replacement.

Way	Page size
0	16kB pages
1	16kB pages
2	16kB pages
3	16kB pages
4	16MB pages
5	16kB pages

Note that 16MB pages do not need multiple ways as there are sufficient TLB entries to allow distinct entries for each 16MB page if the virtual address space is 34-bits or less.

TLB Entries - TLBE

Closely related to page table entries are translation look-aside buffer, TLB, entries. TLB entries have additional fields to match against the virtual address. The count field is used to invalidate the entire TLB. Note that the least significant 10-bits of the virtual address are not stored as these bits are used as an index for the TLB entry.

$Count_6$	LRU_3

V	LVL/BC_5	RGN_3	M	A	T	S	G	SW_2	$CACHE_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$PPN_{31..0}$													
$PPN_{63..32}$													

$VPN_{41..10}$			
$VPN_{73..42}$			
$\sim_4$	$ASID_{11..0}$	$\sim_5$	$VPN_{83..73}$

Small TLB Entries - TLBE

The small TLB is used for the test system which contains only 512MB of physical RAM to conserve hardware resources. The address ranges are more limited, 40-bits for the physical address and 70-bits for the virtual address.

$Count_6$	LRU_3

V	LVL/BC_5	RGN_3	M	A	T	S	G	SW_2	$CACHE_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$
$\sim_6$		$PPN_{25..0}$											

$VPN_{41..10}$				
$\sim_4$	$ASID_{11..0}$	PS	~	$VPN_{55..42}$

What is Translated?

The TLB processes addresses including both instruction and data addresses for all modes of operation. It is known as a *unified* TLB.

Page Size

Because the TLB caches address translations it can get away with a much smaller page size than the page map can for a larger memory system. 4kB is a common size for many systems. There are some indications in contemporary documentation that a larger page size would be better. In this

case the TLB uses 16kB. For a 512MB system (the size of the memory in the test system) there are 32768 16kB pages.

Ways

The first four ways in the TLB are reserved for 16kB page translations. The next way, 4 is reserved for 16MB page translations. The last way is reserved for fixed translations of 16kB pages.

Management

The TLB unit may be updated by either software or hardware. This is selected in the page table base register. If software miss handling is selected when a translation miss occurs, an exception is generated to allow software to update the TLB. It is left up to software to decide how to update the TLB. There may be a set of hierarchical page tables in memory, or there could be a hash table used to store translations.

Accessing the TLB

A TLB entry contains too much information to be updated with a single register write. Since the information must also be updated atomically to ensure correct operation, the TLB update occurs in an indirect fashion. First holding registers are loaded with the desired values, then all the holding registers are written to the TLB in a single atomic cycle. The TLB is addressed in the physical memory space in the address range $F…FE000xx. There are eight buckets which must be filled with TLB info using store instructions. Then address $F…FE0007E is written to causing the TLB to be updated.

The low order bits of the bucket six determine which way to update in the TLB if the algorithm is a fixed way algorithm. Otherwise, if LRU is selected the LRU entry will be updated, otherwise a way to update will be selected randomly. The data is octa-byte aligned.

00	TLBE ($PTE_{63..0}$)									
08					TLBE ($PTE_{95..64}$)					
10	TLBE ($VPN_{63..0}$)									
18					TLBE ($VPN_{95..64}$)					
20	TLB Miss $Address_{63..0}$									
28	$\sim_4$	Miss $ASID_{12}$		$\sim_{16}$	TLB Miss $Address_{95..64}$					
30 to 68										
70						AL_2	0	Entry Num_{10}	~	Way_4
78	RWTRIG	WTRIG	RTRIG	$\sim_8$	$\sim_{32}$					

ADR	
7C	No operation
7D	Read TLBE
7E	Write TLBE
7F	Read and Write TLBE

?RWX$_3$

If RWX3 attributes are specified non-zero, then they will override the attributes coming from the region table. Otherwise RWX attributes are determined by the region table.

CACHE$_4$

The cache$_4$ field is combined with the cache attributes specified in the region table. The region table takes precedence; however, if the cache$_4$ field indicates non-cache-ability then the data will not be cached.

Example TLB Update Routine

```
_TLBMap:
        ldo     a0,0[sp]
        ldo     a1,8[sp]
        ldo     a2,16[sp]
        ldo     a3,24[sp]
        ; <lock TLB update semaphore>
        sto     a0,0xFFE00000           # TLBE value
        sto     a1,0xFFE00008           # TLBE value
        sto     a2,0xFFE00010           # TLBE value
        sto     a3,0xFFE00070           # control
        stb     a0,0xFFE0007E           # triggers a TLB update
        ; <unlock TLB update semaphore>
        add     sp,sp,32
        rts
```

TLB Entry Replacement Policies

The TLB supports three algorithms for replacement of entries with new entries on a TLB miss. These are fixed replacement (0), least recently used replacement (1) and random replacement (2). The replacement method is stored in the AL$_2$ bits of the page table base register.

For fixed replacement, the way to update must be specified by a software instruction. Least recently used replacement, LRU, selects the least recently used address translation to be overwritten. Random replacement chooses a way to replace at random.

Flushing the TLB

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

Reset

On a reset the TLB is preloaded with translations that allow access to the system ROM.

Global Bit

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

Card Table

Overview

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

Organization

At the lowest level memory is divided into 256-byte card memory pages. Each card has a single byte recording whether a pointer store has taken place in the corresponding memory area. To cover a 512MB memory system 2MB card memory is required at the outermost layer. A byte is used rather than a bit to allow byte store operations to update the table directly without having to resort to multiple instructions to perform a bit-field update.

To improve the performance of scanning a hardware card table, HCT, is present which divides memory at an upper level into 8192-byte pages. The hardware card table indicates if a pointer store operation has taken place in one of the 8192-byte pages. It is then necessary to scan only

cards representing the 8192-byte page rather than having to scan the entire 2MB card table. Note that this memory is organized as 2048 32-bit words. Allowing 32-bits at a time to be tested.

To further improve performance a master card table, MCT, is present which divides memory at the uppermost layer into 16-MB pages.

Layer	Resolving Power	
0	2 MB	256B pages
1	64k bits	8kB pages
2	32 bits	16 MB pages

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

Location

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

Operation

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

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

Sample Write Barrier

```
; Milli-code routine for garbage collect write barrier.
; This sequence is short enough to be used in-line.
; Three level card memory.
; a2 is a register pointing to the card table.
; STPTR will cause an update of the master card table, and hardware card table.
;
GCWriteBarrier:
        STPTR       a0,[a1]                 ; store the pointer value to memory at a1
```

```
LSR     t0,a1,#8        ; compute card address
STB     r0,[a2+t0]      ; clear byte in card memory
```

System Memory Map

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

Address	Usage	
$00000000 to $001FFFFF	Card Table (2 MB)	
$00210000 to $0022FFFF	PAM (128kB 2 copies)	
$00280000 to $0029FFFF	Key memory (128 kB)	

Instruction Set

Overview

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

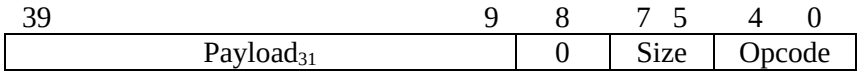
Predicated Instruction Execution

Some processors include the ability to execute virtually any instruction conditionally, for example the ARM processor or INTEL Itanium IA64. It's a powerful means of removing branches from the instruction stream. Sequences of instructions executed with predicates rather than branching around the instructions should be kept short. The issue is the amount of time spent fetching the instructions and treating them as NOPs versus the time it would take to branch around the instructions. A compiler can optimize this and choose the best means. One of the problems of predicates is that they use up bits in the instruction regardless of whether they are useful. For instance, the Itanium has a six-bit field in virtually every instruction. The result is that a wider instruction format of 41 bits is used. A second problem with predicates is that they act like a second instruction being executed at the same time as the instruction they are associated with. The predicate operation requires a predicate register read, and a predicate evaluation operation. This adds complexity to the processor. Predicate registers are another form of register that must be present and bypassed in an overlapped or superscalar design.

The first Thor processing core features uses a whole byte for predicates, but gains back some of the opcode space by using redundant forms of the predicates as single byte instructions. The most recent version of Thor has two means of predication. A vector mask register may be specified for a scalar operation in which case the scalar operation takes place only if the mask register is equal to one. The second means of predication is via an instruction modifier. An instruction modifier precedes the instruction to add to or modify its operation. Since predicates are used infrequently the use of a modifier is an efficient manner to encode the operation.

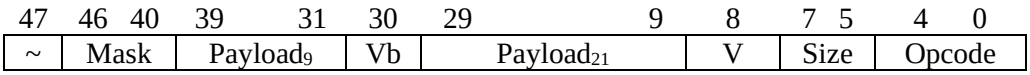
Instruction Descriptions

Scalar Instructions Layout



Vector Instruction Layout

A vector instruction is identical to its scalar counterpart except that one of the vector bits of the instruction is set and there may be an additional field present to specify the mask register. This field adds one byte to the instruction.



Register-Register Vector Decode

Vc, Vb, V	Rb	Ra	Rt	Mask
000	scalar	scalar	scalar	No
001	scalar	scalar	scalar	Yes
010	scalar	vector	vector	No
011	scalar	vector	vector	Yes
100	vector	vector	vector	No
101	vector	vector	vector	Yes

Register-Immediate Vector Decode

Vc, V	Ra	Rt	Mask
00	scalar	scalar	No
01	scalar	scalar	Yes
10	vector	vector	No
11	vector	vector	Yes

Opcode Maps

Major Opcode

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

{R2} Operations

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

{BIT – Func3}

	0	1	2	3	4	5	6	7
0x	0 CLR	1 SET	2 COM	3 SBX	4 EXTU	5 EXTS	6	7 {BITRR}

{SHIFT – Func5}

	0	1	2	3	4	5	6	7
0x	0 ASL	1 ASR	2 LSL	3 LSR	4 ROL	5 ROR	6	7
	8 ASLI	9 ASRI	10 LSLI	11 LSRI	12 ROLI	13 RORI	14	15
1x	16 VSHLV	17 VSHRV	18	19	20	21	22	23
	24 VSHLVI	25 VSHRVI	26	27	28	29	30	31

{FLT2} Operations

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

{FLT1} Operations

	0	1	2	3	4	5	6	7
0x	0	1	2 FOTI	3 ITOF	4	5	6 FSIGN	7 FSIG
	8 FSQRT	9 FS2D	10 FS2Q	11 FD2Q	12	13	14 ISNAN	15 FINITE
1x	16	17	18	19	20	21 FTRUNC	22	23 FRES
	24	25 FD2S	26 FQ2S	27 FQ2D	28	29	30 FCLASS	31
2x	32 FABS	33	34 FNEG	35	36	37	38	39
	40							
3x	48							
	56							

{AMO} Operations

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

Operand Swapping

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

Operand Swap

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

Operand Sizes

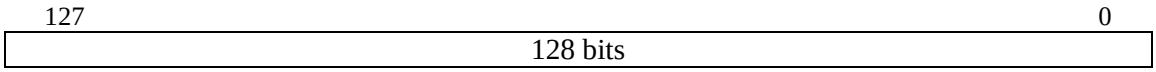
Many instructions support five different operand sizes: byte, wyde, tetra, octa and hexi. The operand size is selected by suffixing the mnemonic with 'b' for byte, 'w' for wyde, 't' for tetra, 'o' for octa and 'h' for hexi. Size code 6 selects decimal arithmetic mode.

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa
4	.h	128-bit Hexi
5		Reserved
6	.d	128-bit decimal
7		reserved

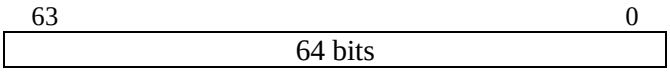
Arithmetic Operations

Representations

long

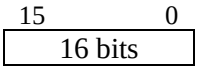


int

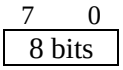


short

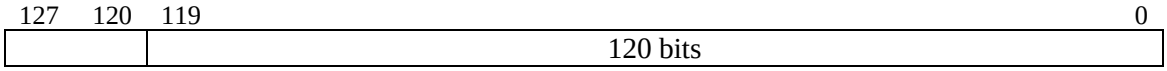
char



byte



decimal

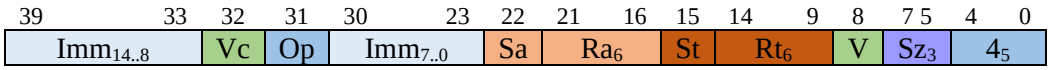


Decimal integers use densely packed decimal format which provide 38 digits of precision.

Arithmetic Operations

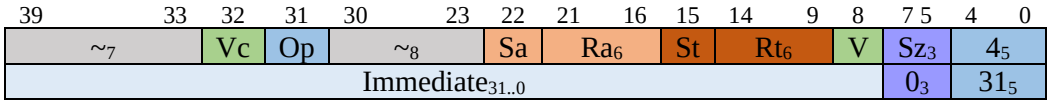
Arithmetic operations include addition, subtraction, multiplication and division. These are available with the ADD, SUB, CMP, MUL, and DIV instructions. There are several variations of the instructions to deal with signed and unsigned values. The format of the typical immediate mode instruction is shown below:

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



Immediate instructions may have the constant extended via the use of postfixed immediates.

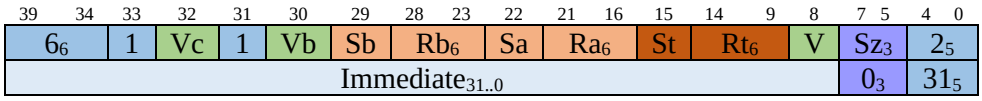
ADD Rt,Ra,Imm$_{15}$



There may seem to be significant wasted space in the instruction when an instruction postfix is used. However, the use of a postfix is the rare case which occurs when a fifteen-bit immediate value is not sufficient. Having the postfix begin with bit 0 to 31 encoded is to allow for instructions that do not have space for an immediate field in the instruction. The postfix is kept consistent between all instructions to make decoding easier to handle and smaller resource wise.

Note that all arithmetic instructions can use an immediate value via a postfix immediate. Not all arithmetic instructions support a fifteen-bit immediate field. Instead, when a postfix is used it will override the value coming from register Rb. The following instruction ignores the Rb register value and multiplies by a postfix immediate.

MULSU Rt, Ra, Rb



There are both signed and unsigned versions of the arithmetic operations. However, note there is no signed or unsigned compare operation as a single compare instruction produces results for both signed and unsigned comparisons. Signed and unsigned ADD and SUB currently work the same way. Two separate versions have been reserved to support the overflow exception in the future.

Logical Operations

Thor logic operations include only the basic ‘and’, ‘or’ and ‘xor’ operations. Other variations of the instructions like ‘nand’, or ‘nor’ are possible by inverting registers. The assembler recognizes some of the possible combinations of register inversion as distinct instructions like ‘nand’ or ‘nor’. Unlike Table888 there are more immediate forms available even for rarely used instructions.

Immediate Operate Functions

Most instructions at the root level are immediate operate instructions, the ones that are not are bordered in red in the table below. Note that in some cases non-immediate format instructions may use an immediate via a postfix instruction.

Major Opcode

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

ABS – Absolute Value

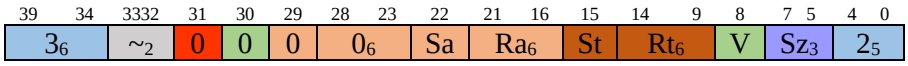
Description:

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

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

Integer Instruction Format: R2

ABS Rt, Ra



Clock Cycles: 4/6

Operation:

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

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

ADD - Addition

Description:

Add two source operands and place the sum in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values unless decimal mode is selected (SZ_3=6) in which case values are treated as BCD numbers. This instruction may be used with the CARRY modifier to perform extended precision addition.

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

Operation:

Rt = Ra + Rb or Rt = Ra + Imm

Clock Cycles:

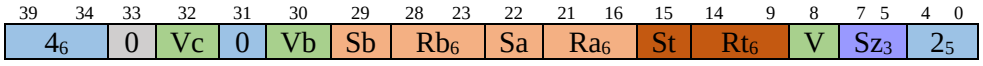
Execution Units: All Integer ALU's

Exceptions: none

Notes:

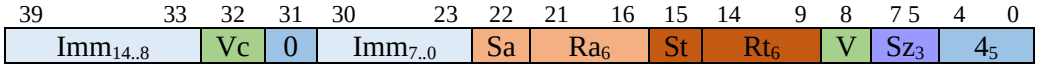
Instruction Formats: R2, RI

ADD Rt, Ra, Rb



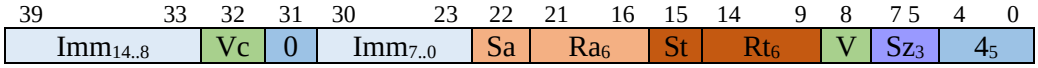
Clock Cycles: 4/6

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



Clock Cycles: 4/6

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



Clock Cycles: 4/6

Cycles	<128	128
IFETCH	*	*
DECODE	*	*
DECODE2		*
OFETCH	*	*
EXECUTE	*	*
WRITEBACK		*
Total	4	6

AND – Bitwise And

Description:

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

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

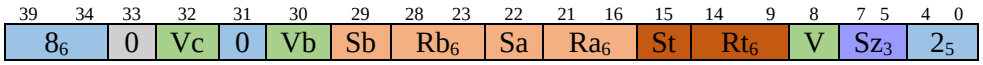
Clock Cycles: 1

Operation:

Rt = Ra & Rb or Rt = Ra & Imm

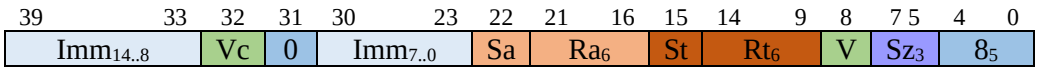
Instruction Formats: R2, RI

AND Rt, Ra, Rb



Clock Cycles: 4/6

AND Rt,Ra,Imm_{15}



Clock Cycles: 4/6

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Cycles	<128	128
IFETCH	*	*
DECODE	*	*
DECODE2		*
OFETCH	*	*
EXECUTE	*	*
WRITEBACK		*
Total	4	6

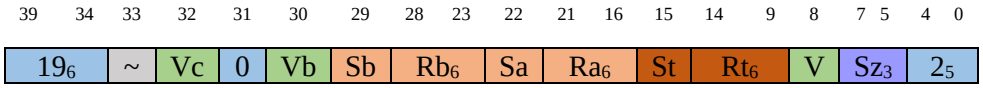
BMAP – Byte Map

Description:

First the target register is cleared, then bytes are mapped from the 16-byte source Ra into bytes in the target register. This instruction may be used to permute the bytes in register Ra and store the result in Rt. This instruction may also pack bytes, wydes or tetras. The map is determined by the low order 64-bits of register Rb or a 64-bit immediate constant. Bytes which are not mapped will end up as zero in the target register.

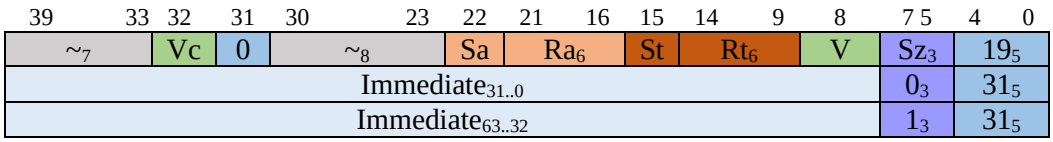
Instruction Formats: R2, RI

BMAP Rt, Ra, Rb



Clock Cycles: 4/6

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



Clock Cycles: 4/6

Operation:

Vector Operation

Execution Units: First Integer ALU

Clock Cycles: 1

Exceptions: none

Notes:

Cycles	<128	128
IFETCH	*	*
DECODE	*	*
DECODE2		*
OFETCH	*	*
EXECUTE	*	*
WRITEBACK		*
Total	4	6

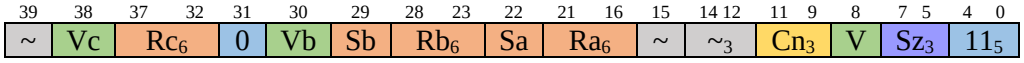
CHK – Check Register Against Bounds

Description:

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

Instruction Formats:

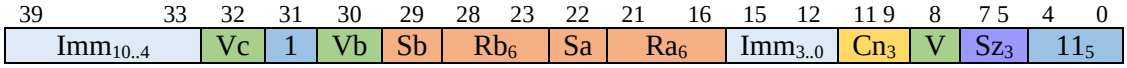
CHK Ra, Rb, Cn – Register direct



Clock Cycles: 1

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

CHKI Ra, Imm, Cn



Clock Cycles: 1

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

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: bounds check

Notes:

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

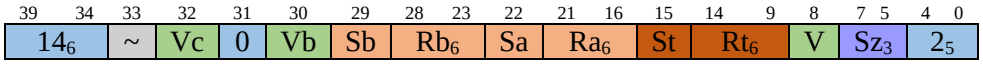
CLMUL – Carry-less Multiply

Description:

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

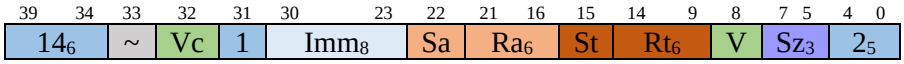
Instruction Formats:

CLMUL Rt, Ra, Rb



Clock Cycles: 7 + n, where n = number of bits

CLMUL Rt,Ra,Imm_8



Clock Cycles: 7 + n, where n = number of bits

Exceptions: none

Execution Units: First Integer ALU

Operations

Rt = Ra * Rb

Vector Operation

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] * Vb[x]

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

Exceptions: none

CMP - Comparison

Description:

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

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

Operation:

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

Clock Cycles: 1

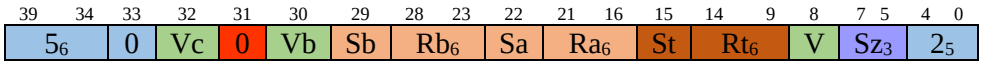
Execution Units: All Integer ALU's

Exceptions: none

Notes:

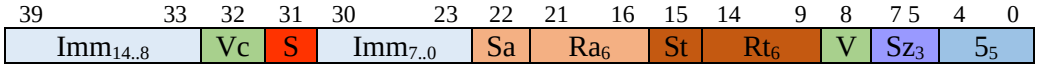
Instruction Formats: R2, RI

CMP Rt, Ra, Rb



Clock Cycles: 4/6

CMP Rt,Ra,Imm_{15}



Clock Cycles: 4/6

Rt Bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]
8			
9			
10	LO / CS	< unsigned less than	a < b
11	LS	<= unsigned less than or equal	a <= b
12	HS / CC	unsigned greater than or equal	a >= b
13	HI	unsigned greater than	a > b
14			
15			

CMPS.B – Signed Byte Comparison

Description:

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

Supported Operand Sizes: .b

Operation:

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

Clock Cycles: 1

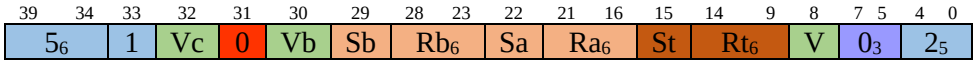
Execution Units: All Integer ALU's

Exceptions: none

Notes:

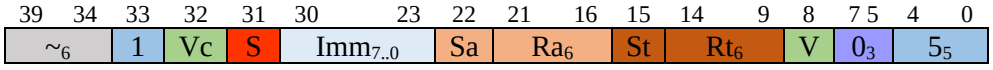
Instruction Formats:

CMPS.B Rt, Ra, Rb – Register direct



Clock Cycles: 4/6

CMPS.B Rt,Ra,Imm$_{15}$



Clock Cycles: 4/6

Rt bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	
1	NE	< > not equal	
2	LT	< less than	
3	LE	<= less than or equal	
4	GE	>= greater than or equal	
5	GT	> greater than	
6			
7			

CMPU.B – Unsigned Byte Comparison

Description:

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

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

Operation:

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

Clock Cycles: 1

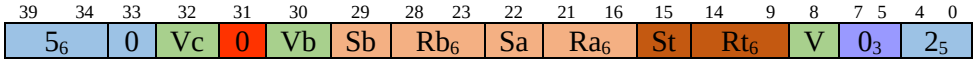
Execution Units: All Integer ALU's

Exceptions: none

Notes:

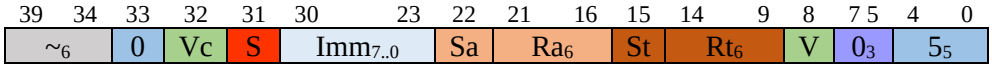
Instruction Formats:

CMPU.B Rt, Ra, Rb – Register direct



Clock Cycles: 1

CMPU.B Rt,Ra,Imm$_8$



Clock Cycles: 1

Rt bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	
1	NE	< > not equal	
2	LTU	< less than	
3	LEU	<= less than or equal	
4	GEU	>= greater than or equal	
5	GTU	> greater than	
6			
7			

CNTLZ – Count Leading Zeros

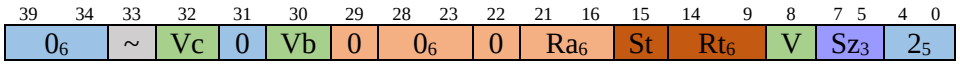
Description:

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

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

Integer Instruction Format: R1

CNTLZ Rt, Ra, Rb – Register direct



Clock Cycles: 1

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTLO – Count Leading Ones

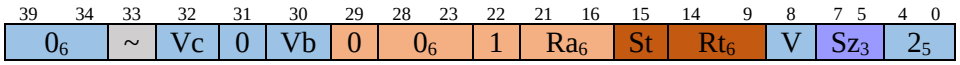
Description:

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

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

Integer Instruction Format: R1

CNTLO Rt, Ra, Rb – Register direct



Clock Cycles: 1

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTPOP – Count Population

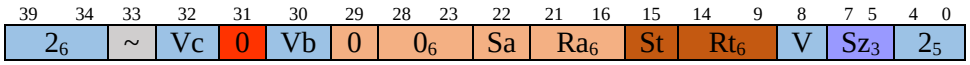
Description:

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

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

Integer Instruction Format: R1

CNTPOP Rt, Ra, Rb – Register direct



Clock Cycles: 1

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CSR – Control and Special Registers Operations

Description:

Perform an operation on a CSR.

Operation	Op_3	
Read CSR	0	
Write CSR	1	
Or to CSR (set bits)	2	
And complement to CSR (clear bits)	3	
Exclusive Or to CSR (flip bits)	4	

Supported Operand Sizes: N/A

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

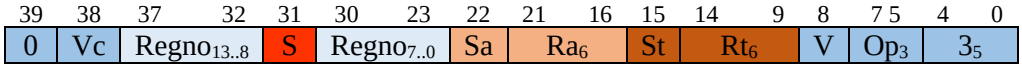
Instruction Formats:

OR Rt, Ra, CSR

ANDC Rt, Ra, CSR

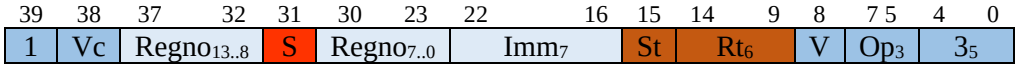
EOR Rt, Ra, CSR

CSR Rt,Ra,#$Regno_{12}$



Clock Cycles: 1

CSR Rt, #imm,#$Regno_{12}$



DIVS – Signed Division

Description:

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

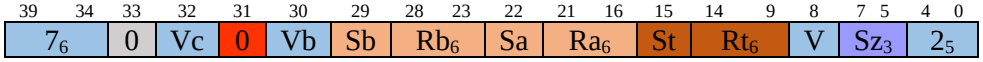
Supported Operand Sizes: .b, .w, .t, .o

Operation:

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

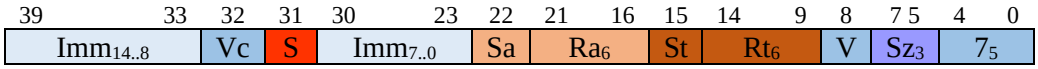
Instruction Formats:

DIVS Rt, Ra, Rb – Register direct



Clock Cycles: 100

DIVS Rt,Ra,Imm_{16}



Clock Cycles: 100

Execution Units: All Integer ALU's

Exceptions: none

Notes:

DIVU – Unsigned Division

Description:

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

15-bit immediate mode is not available for this instruction.

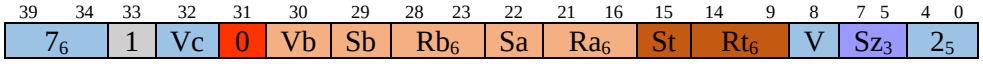
Supported Operand Sizes: .b, .w, .t, .o

Operation:

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

Instruction Formats:

DIVU Rt, Ra, Rb – Register direct



Clock Cycles: 100

Execution Units: All Integer ALU's

Exceptions: none

Notes:

EOR – Bitwise Exclusive Or

Description:

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

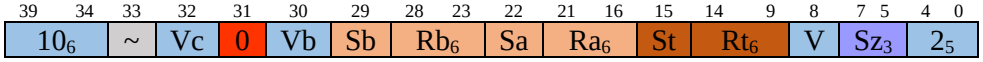
Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Operation:

Rt = Ra ^ Rb or Rt = Ra ^ Imm

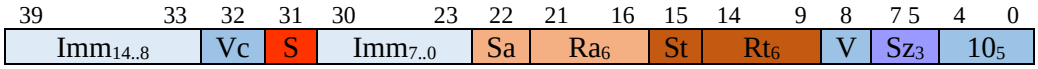
Instruction Formats:

EOR Rt, Ra, Rb – Register direct



Clock Cycles: 1

EOR Rt,Ra,Imm_{16}



Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ENOR – Bitwise Exclusive Nor

Description:

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

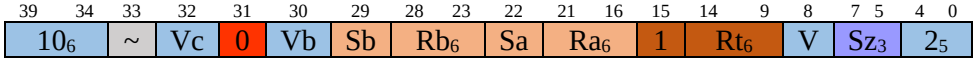
Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Operation:

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

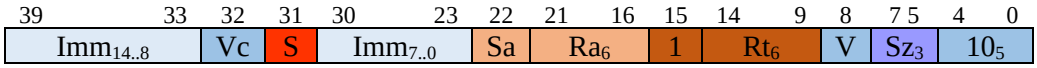
Instruction Formats:

ENOR Rt, Ra, Rb – Register direct



Clock Cycles: 1

ENOR Rt,Ra,Imm_{16}



Clock Cycles: 1

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

PFX – Constant Postfix

Description:

The PFX instruction postfix is used to build large constants for use in the preceding instruction as the immediate constant for the instruction. There are three postfix instructions which extend the constant from different bit locations. They should be used in the order PFX0, PFX1,PFX2. A postfix may be omitted if the omitted bits match what would be included.

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

Supported Operand Sizes: N/A

Instruction Format: PFX0

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

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

Instruction Format: PFX1

This format extends the previous constant value by 32 bits beginning at bit 32 and sign extends the value to the width of the machine. If this postfix is used without a preceding PFX0 postfix, then the low order 32-bits of the constant will be zero.

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

Instruction Format: PFX2

This format extends the previous constant value by 64 bits beginning at bit 64 and sign extends the value to the width of the machine. Note that the format is always used twice in succession to provide the upper 64-bits of a constant. If this postfix is used without a preceding PFX0, PFX1 postfix, then the low order bits of the constant will be zero.

39 … 8	7 … 5	4 … 0
$Immediate_{95..64}$	2_3	31_5
$Immediate_{127..96}$	2_3	31_5

MODS – Signed Modulus

Description:

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

15-bit Immediate mode is not available for this instruction.

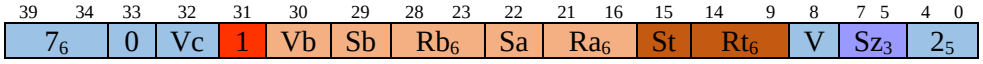
Supported Operand Sizes: .b, .w, .t, .o

Operation:

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

Instruction Formats:

MODS Rt, Ra, Rb – Register direct



Clock Cycles: 100

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MODU – Unsigned Modulus

Description:

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

15-bit Immediate mode is not available for this instruction.

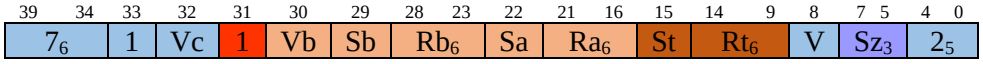
Supported Operand Sizes: .b, .w, .t, .o

Operation:

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

Instruction Formats:

MODU Rt, Ra, Rb – Register direct



Clock Cycles: 100

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MULS – Multiply Signed

Description:

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

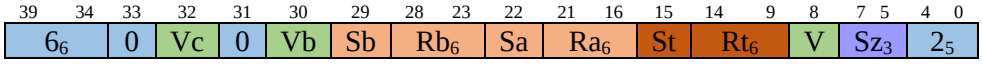
Supported Operand Sizes: .b, .w, .t, .o

Operation:

Rt = Ra * Rb or Rt = Ra * Imm

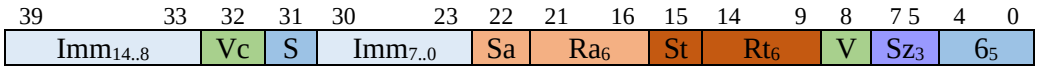
Instruction Formats:

MULS Rt, Ra, Rb



Clock Cycles: 7+n, where n is the number of bits.

MULS Rt,Ra,Imm_{15}



Clock Cycles: 7+n

Clock Cycles: 12

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

MULSU – Signed-Unsigned Multiplication

Description:

Multiply two source operands and place the product in the target register. The first operand is a signed value, the second operand is unsigned. All registers are treated as integer registers. Arithmetic is twos-complement values.

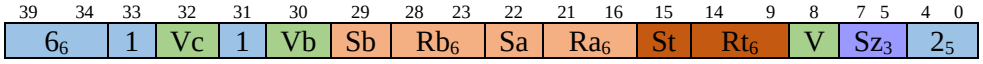
Supported Operand Sizes: .b, .w, .t, .o, .h

Operation:

Rt = Ra * Rb or Rt = Ra * Imm

Instruction Formats:

MULSU Rt, Ra, Rb



Clock Cycles: 7+n, where n is the number of bits.

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MULU – Unsigned Multiplication

Description:

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

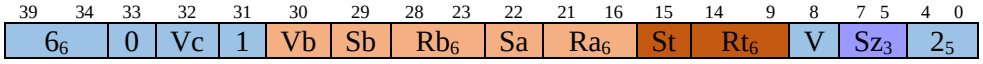
Supported Operand Sizes: .b, .w, .t, .o, .h

Operation:

Rt = Ra * Rb or Rt = Ra * Imm

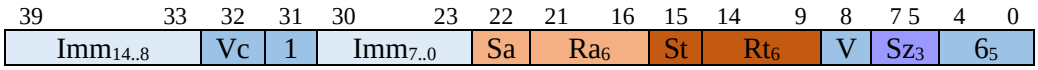
Instruction Formats:

MULU Rt, Ra, Rb – Register direct



Clock Cycles: 7+n, where n is the number of bits.

MULU Rt,Ra,Imm_{16}



Clock Cycles: 7+n

Execution Units: All Integer ALU's

Exceptions: none

Notes:

NAND – Bitwise And and Invert

Description:

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

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

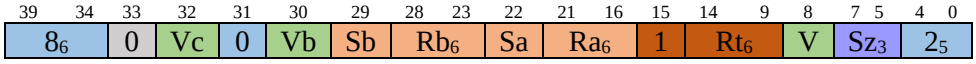
Clock Cycles: 1

Operation:

Rt = ~(Ra & Rb)

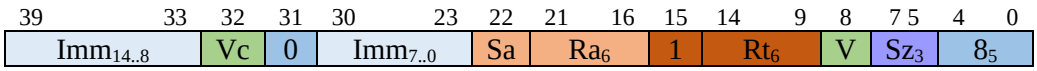
Instruction Formats:

NAND Rt, Ra, Rb – Register direct



Clock Cycles: 1

NAND Rt,Ra,Imm_{15}



Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

NOR – Bitwise Or and Invert

Description:

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

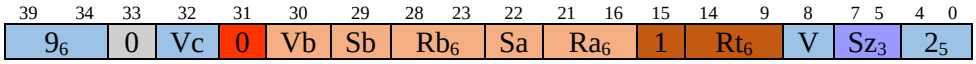
Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Operation:

Rt = ~(Ra | Rb)

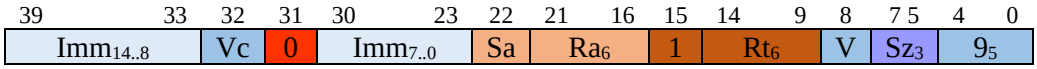
Instruction Formats:

NOR Rt, Ra, Rb – Register direct



Clock Cycles: 1

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



Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

OR – Bitwise Or

Description:

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

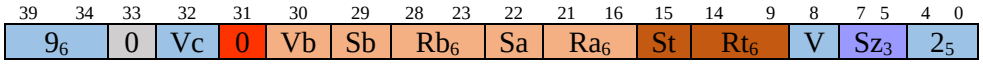
Supported Operand Sizes: .b, .w, .t, .o, .c, .p, .n

Operation:

Rt = Ra | Rb or Rt = Ra | Imm

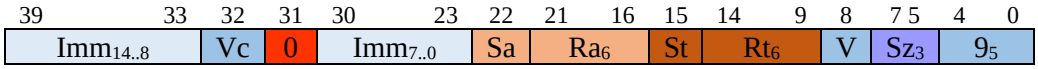
Instruction Formats:

OR Rt, Ra, Rb – Register direct



Clock Cycles: 1

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



Clock Cycles: 1

Clock Cycles: 2

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

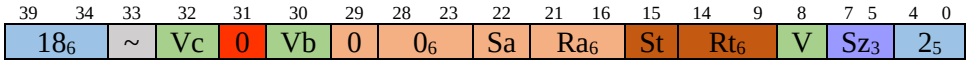
REVBIT – Reverse Bit Order

Description:

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

Integer Instruction Format: R2

REVBIT Rt, Ra – Register direct



Clock Cycles: 1

Operation:

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

SEQ – Set if Equal

Description:

Compare two source operands for equality and place the result in the target register. The result is a Boolean true or false.

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

Operation:

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

Clock Cycles: 1

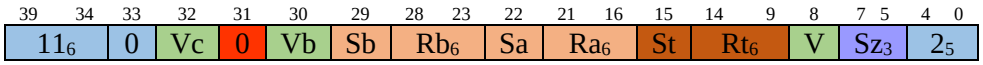
Execution Units: All Integer ALU's

Exceptions: none

Notes:

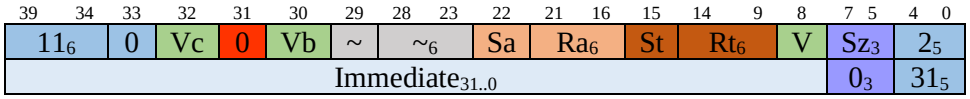
Instruction Formats:

SEQ Rt, Ra, Rb



Clock Cycles: 1

SEQ Rt, Ra, Imm



Clock Cycles: 1

SGE – Set if Greater Than or Equal

Description:

Compare two source operands for greater than or equal and place the result in the target register. The result is a Boolean true or false. This is the same instruction as SLT except that the result is inverted.

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

Operation:

Rt = Ra < Rb or Rt = Ra < Imm

Clock Cycles: 1

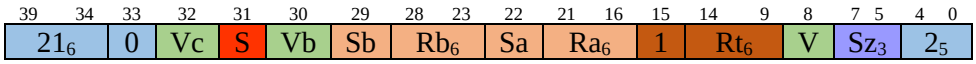
Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

SGE Rt, Ra, Rb – Register direct



Clock Cycles: 1

SGT – Set if Greater Than

Description:

Compare two source operands for greater than and place the result in the target register. The result is a Boolean true or false. This is the same instruction as SLE except that the result is complemented.

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

Operation:

Rt = Ra > Rb or Rt = Ra > Imm

Clock Cycles: 1

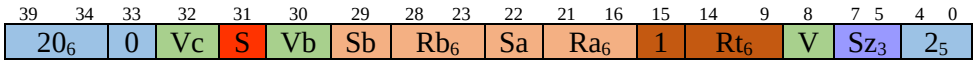
Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

SLE Rt, Ra, Rb – Register direct



Clock Cycles: 1

SLE – Set if Less Than or Equal

Description:

Compare two source operands for less than or equal and place the result in the target register. The result is a Boolean true or false.

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

Operation:

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

Clock Cycles: 1

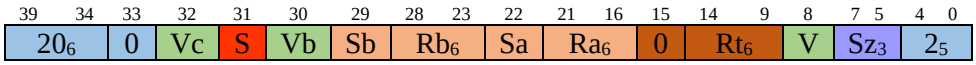
Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

SLE Rt, Ra, Rb – Register direct



Clock Cycles: 1

SLT – Set if Less Than

Description:

Compare two source operands for less than and place the result in the target register. The result is a Boolean true or false.

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

Operation:

Rt = Ra < Rb or Rt = Ra < Imm

Clock Cycles: 1

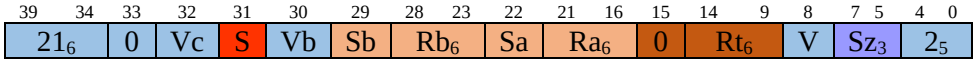
Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

SLT Rt, Ra, Rb – Register direct



Clock Cycles: 1

SNE – Set if Not Equal

Description:

Compare two source operands for inequality and place the result in the target register. The result is a Boolean true or false.

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

Operation:

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

Clock Cycles: 1

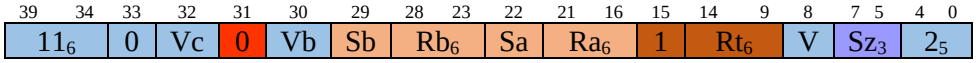
Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

SNE Rt, Ra, Rb – Register direct



Clock Cycles: 1

SQRT – Square Root

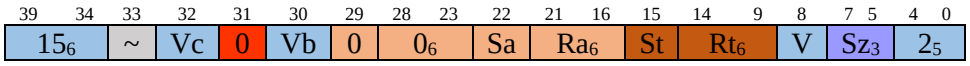
Description:

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

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

Integer Instruction Format: R2

SQRT Rt, Ra – Register direct



Clock Cycles: 1

Operation:

Rt = SQRT(Ra)

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

SUB - Subtraction

Description:

Subtract two source operands and place the difference in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values unless decimal mode is selected (SZ_3=6) in which case values are treated as BCD numbers. This instruction may be used with the CARRY modifier to perform extended precision subtraction.

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

Operation:

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

Clock Cycles:

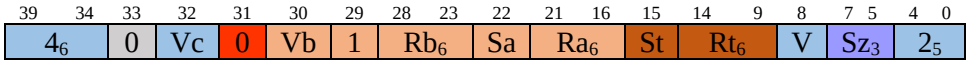
Execution Units: All Integer ALU's

Exceptions: none

Notes:

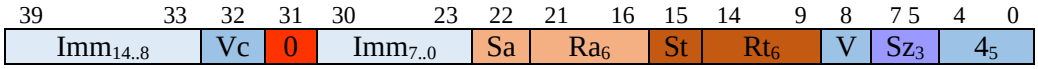
Instruction Formats:

SUB Rt, Ra, Rb – Register direct



Clock Cycles: 1

SUB Rt,Ra,Imm_{15}



Clock Cycles: 1

Vector Arithmetic Operations

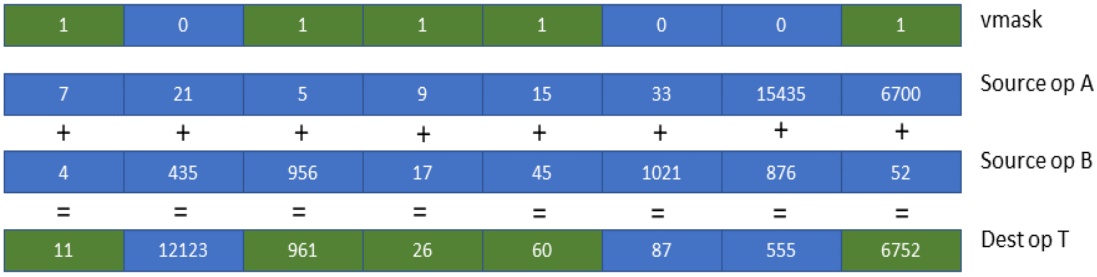
Vector arithmetic operations are identical to scalar ones except that they may operate on vector registers. An extra register specification field may be present in the instruction to allow a mask register to be specified. Vector instructions are either 40 or 48 bits in length depending on the presence of a mask register.

The instruction is prefixed with the letter ‘V’ to indicate a vector form of the instruction. The assembler will recognize a vector instruction if a vector register is specified as one of the operands. The convention of prepending a ‘V’ to the instruction is a visual aid.

VADD - Addition

Description:

Add two source operands and place the sum in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values unless the decimal mode flag is set in which case values are treated as densely packed BCD numbers. This instruction may be used with the CARRY modifier to perform extended precision addition. The following image shows the addition of vectors and how the vector mask register comes into play.



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

Operation:

Rt = Ra + Rb or Rt = Ra + Imm

Clock Cycles:

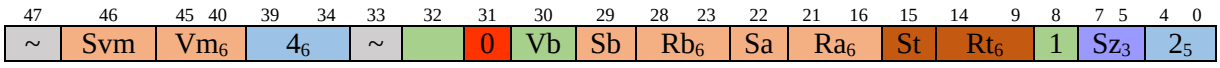
Execution Units: All Integer ALU's

Exceptions: none

Notes:

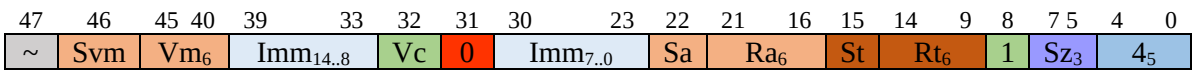
Instruction Formats:

VADD Rt, Ra, Rb, Vm – Register direct



Clock Cycles: 1

VADD Rt,Ra,Imm$_{16}$,Vm



Clock Cycles: 1

VAND – Bitwise And

Description:

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

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

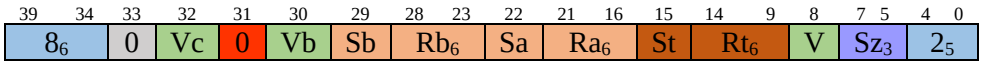
Clock Cycles: 1

Operation:

Rt = Ra & Rb or Rt = Ra & Imm

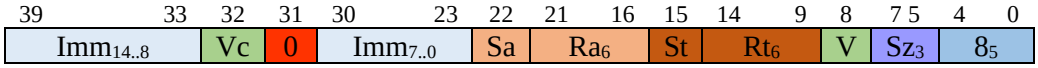
Instruction Formats:

VAND Rt, Ra, Rb – Register direct



Clock Cycles: 1

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



Clock Cycles: 1

Execution Units: All Integer ALU’s

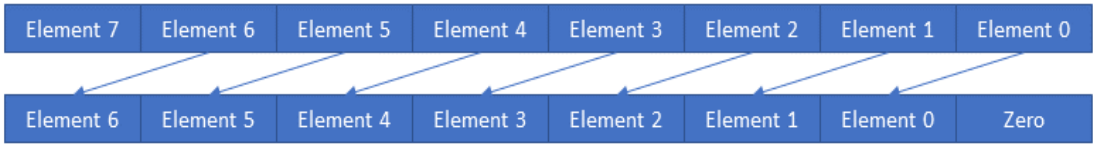
Exceptions: none

Notes:

VSHLV – Shift Vector Left

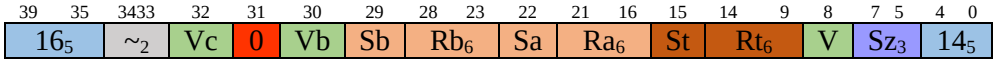
Description

Elements of the vector are transferred upwards to the next element position. The first is loaded with the value zero. The highest element is lost. This is also called a slide operation. Elements may be moved a variable number of elements to the left. The image depicts just a single element shift.



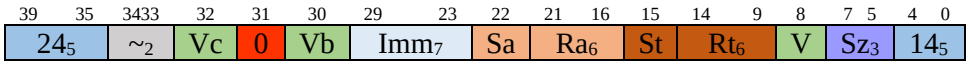
Instruction Formats:

VSHLV Rt, Ra, Rb



Clock Cycles: 1

VSHLV Rt, Ra, Imm_7



Clock Cycles: 1

Operation

Amt = Rb

For x = VL-1 to Amt

    Vt[x] = Va[x-amt]

For x = Amt-1 to 0

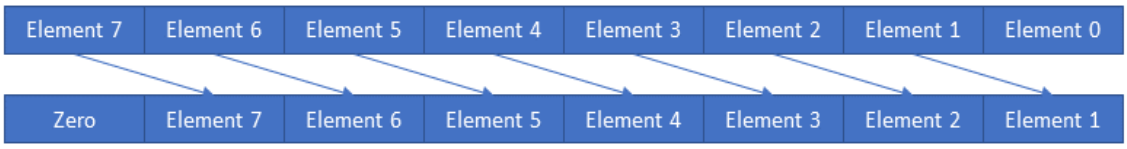
    Vt[x] = 0

Exceptions: none

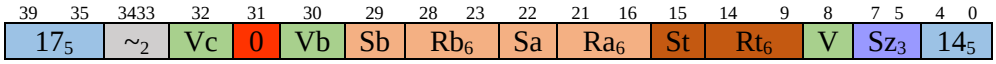
VSHRV – Shift Vector Right

Description

Elements of the vector are transferred downwards to the next element position. The last is loaded with the value zero. This is also called a slide operation. Elements may be moved a variable number of elements to the right. The image depicts just a single element shift.

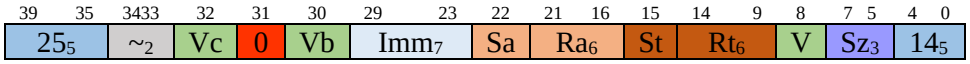


VSHRV Rt, Ra, Rb



Clock Cycles: 1

VSHRV Rt, Ra, Imm_7



Clock Cycles: 1

Operation

Amt = Rb

For x = 0 to VL-Amt

Vt[x] = Va[x+amt]

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

Vt[x] = 0

Exceptions: none

Floating-Point Operations

Precision

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

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

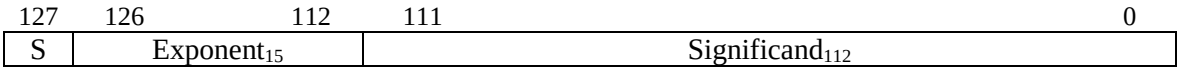
Representations

Binary Floats

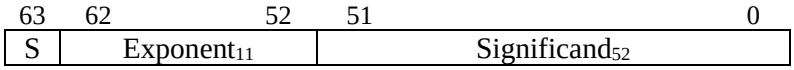
Triple Precision, Float:128

The core uses a 128-bit quad precision binary floating-point representation.

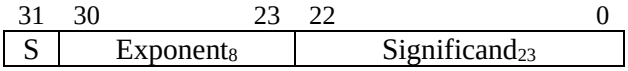
Quad Precision, long double



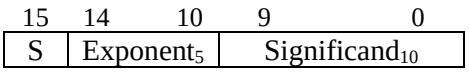
Double Precision, double



Single Precision, float

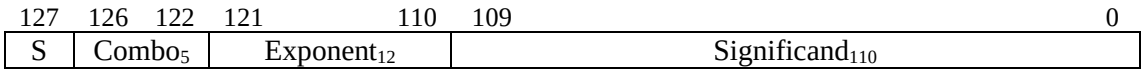


Half Precision, short float



Decimal Floats

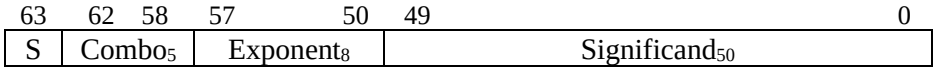
The core uses a 128-bit densely packed decimal triple precision floating-point representation.



The significand stores 34 densely packed decimal digits. One whole digit before the decimal point.

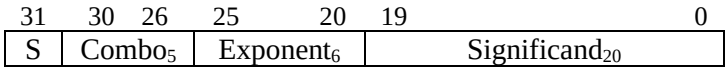
The exponent is a power of ten as a binary number with an offset of 1535. Range is 10^{-1535} to 10^{1536}

64-bit double precision decimal floating point:



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

32-bit single precision decimal floating point:



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

Rounding Modes

Binary Float Rounding Modes

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

Decimal Float Rounding Modes

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

Operand Sizes

Sz_3	Ext.	Operand
0		Reserved
1	.h	16-bit half
2	.s	32-bit single
3	.d	64-bit double
4	.q	128-bit quad
5		reserved
6		128-bit decimal
7		reserved

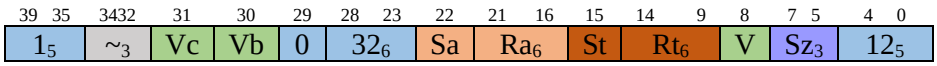
FABS – Absolute Value

Description:

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

Integer Instruction Format: R1

FABS Rt, Ra, Rb – Register direct



Clock Cycles: 1

Operation:

FPt = Abs(FPa)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FADD –Float Addition

Description:

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

Supported Operand Sizes:

Operation:

Rt = Ra + Rb or Rt = Ra + Imm

Clock Cycles: 8

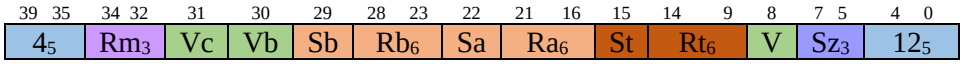
Execution Units: All Integer ALU's

Exceptions: none

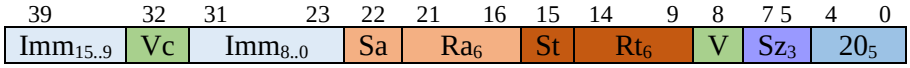
Notes:

Instruction Formats:

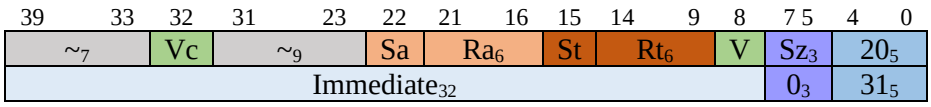
FADD Rt, Ra, Rb – Register direct



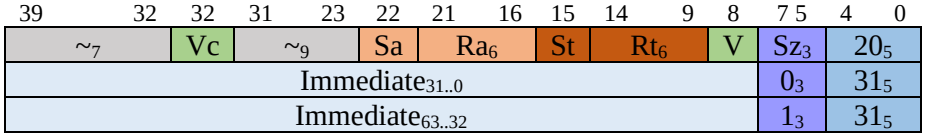
FADD Rt,Ra,Imm$_{16}$



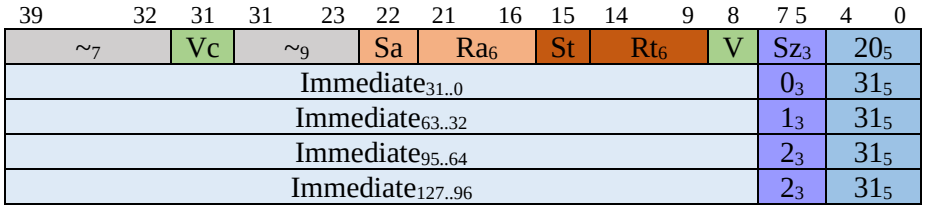
FADD Rt,Ra,Imm$_{32}$



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



FADD Rt,Ra,Imm$_{128}$



FCMP - Comparison

Description:

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as floating-point values. This instruction may compare against lower precision immediate values to conserve code space. Note that result inversion is not available as results are already available in normal and inverted forms in the bit vector. Instead the 'S' bit indicates to swap the operands.

Supported Operand Sizes:

Operation:

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

Clock Cycles: 1

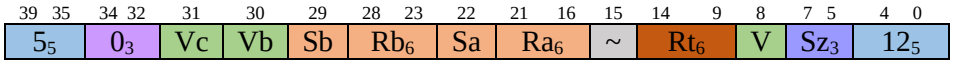
Execution Units: All Integer ALU's

Exceptions: none

Notes:

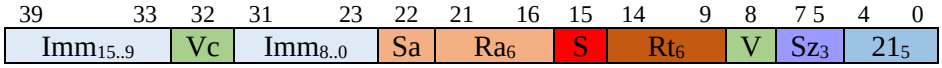
Instruction Formats:

FCMP Rt, Ra, Rb – Register direct



Clock Cycles: 1

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



Clock Cycles: 1

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

14		Reserved	
15		reserved	

FDIV –Float Division

Description:

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

Supported Operand Sizes:

Operation:

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

Clock Cycles:

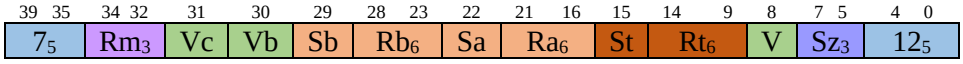
Execution Units: All Integer ALU's

Exceptions: none

Notes:

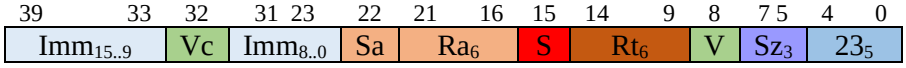
Instruction Formats:

FDIV Rt, Ra, Rb – Register direct



Clock Cycles: 150

FDIV Rt,Ra,Imm_{16}



Clock Cycles: 150

FSEQ – Float Set if Equal

FSNE – Float Set if Not Equal

Description:

Compares two source operands for equality and places the result in the target register. The result is a Boolean true or false. Positive and negative zero are considered equal. This instruction does not support a 16-bit immediate. 32, 64, and 128-bit immediates are supported. For FSEQ is either operand is a NaN zero the result is zero.

Supported Operand Sizes:

Operation:

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

Clock Cycles: 1

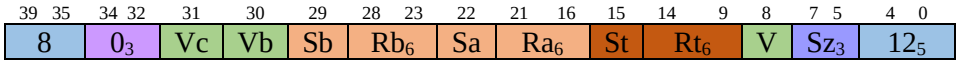
Execution Units: All FPU's

Exceptions: none

Notes:

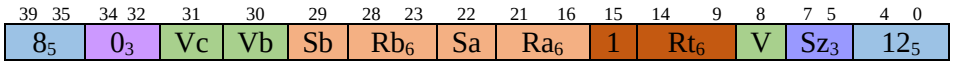
Instruction Formats:

FSEQ Rt, Ra, Rb



Clock Cycles: 1

FSNE Rt, Ra, Rb



Clock Cycles: 1

FSGE – Float Set if Greater Than or Equal

Description:

Compares two source operands for greater than or equal and places the result in the target register. The result is a Boolean true or false.

Supported Operand Sizes:

Operation:

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

Clock Cycles: 1

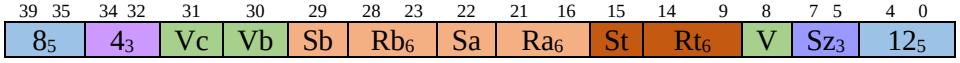
Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSGE Rt, Ra, Rb



FSGT – Float Set if Greater Than

Description:

Compares two source operands for greater than and places the result in the target register. The result is a Boolean true or false.

Supported Operand Sizes:

Operation:

Rt = Ra > Rb or Rt = Ra > Imm

Clock Cycles: 1

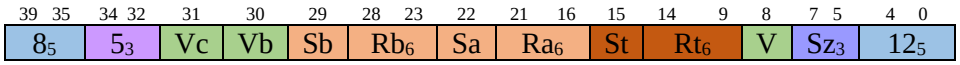
Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSLT Rt, Ra, Rb



Clock Cycles: 1

FSLE – Float Set if Less Than or Equal

Description:

Compares two source operands for less than or equal and places the result in the target register. The result is a Boolean true or false.

Supported Operand Sizes:

Operation:

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

Clock Cycles: 1

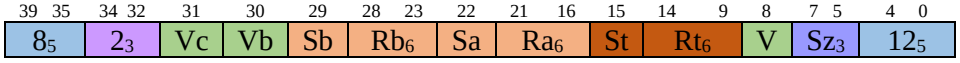
Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSLE Rt, Ra, Rb



FSLT – Float Set if Less Than

Description:

Compares two source operands for less than and places the result in the target register. The result is a Boolean true or false.

Supported Operand Sizes:

Operation:

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

Clock Cycles: 1

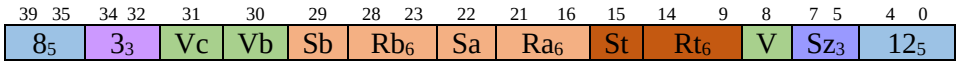
Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

FSLT Rt, Ra, Rb



Clock Cycles: 1

FMUL –Float Multiplication

Description:

Multiply two source operands and place the product in the target register. All registers values are treated as 128-bit floating-point values.

Supported Operand Sizes:

Operation:

Rt = Ra * Rb or Rt = Ra * Imm

Clock Cycles:

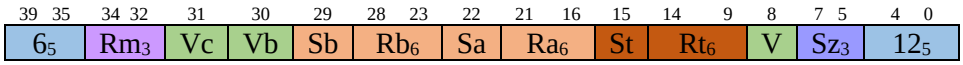
Execution Units: All FPU's

Exceptions: none

Notes:

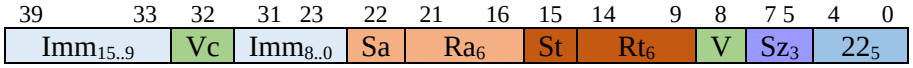
Instruction Formats:

FMUL Rt, Ra, Rb



Clock Cycles: 8

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



Clock Cycles: 8

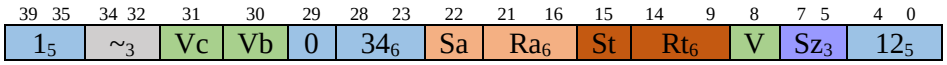
FNEG – Negate Value

Description:

This instruction computes the negative value of the contents of the source operand and places the result in Rt. The sign bit of the value is inverted. No rounding occurs. Note that in most cases a FNEG operation can be absorbed into a previous instruction by negating the result.

Integer Instruction Format: R1

FNEG Rt, Ra



Clock Cycles: 1

Operation:

Rt = -Ra

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FSCALEB –Scale Exponent

Description:

Add the source operand to the exponent. The second source operand is an integer value.

Supported Operand Sizes:

Operation:

Clock Cycles:

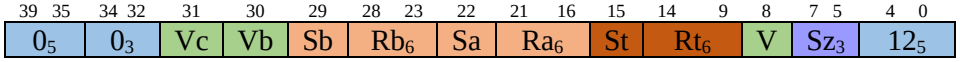
Execution Units: All Integer ALU's

Exceptions: none

Notes:

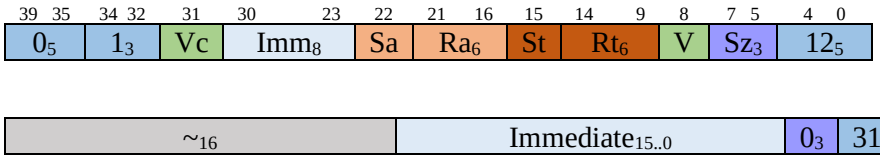
Instruction Formats:

FSCALEB Rt, Ra, Rb



Clock Cycles: 1

FSCALEB Rt, Ra, #Imm



Clock Cycles: 1

FSUB –Float Subtraction

Description:

Subtract two source operands and place the difference in the target register. All registers values are treated as 128-bit floating-point values. This is an alternate mnemonic for the FADD instruction where the second source operand, Rb is assumed negated.

Supported Operand Sizes:

Operation:

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

Clock Cycles: 8

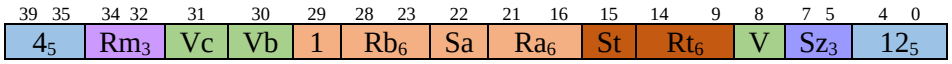
Execution Units: All Integer ALU's

Exceptions: none

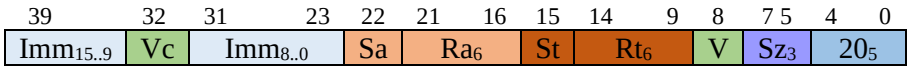
Notes:

Instruction Formats:

FSUB Rt, Ra, Rb – Register direct



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



Clock Cycles: 8

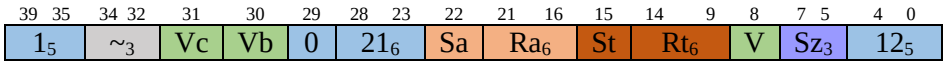
FTRUNC – Truncate Fraction

Description:

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

Integer Instruction Format: R1

FTRUNC Rt, Ra



Clock Cycles: 1

Operation:

Rt = Trunc(Ra)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

ORF – Bitwise Or to Float

Description:

Convert the immediate constant to quad precision format and bitwise ‘or’ with source operand Ra and place the result in the target register. The immediate constant may be a half, single, double, or quad precision value. This instruction is provided mainly for loading a floating-point value into a register. The value may be compressed into the minimum size format for representation without loss of precision. FADD could also be used to load a float constant into a register but it has a longer latency.

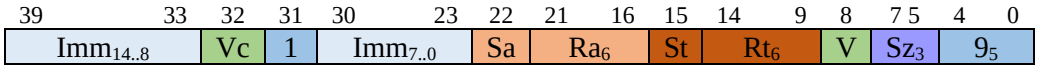
Supported Operand Sizes: .b, .w, .t, .o, .h

Operation:

Rt = Ra | |Convert(Imm)

Instruction Formats:

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



Clock Cycles: 1

Clock Cycles: 1

Execution Units: All Integer ALU’s

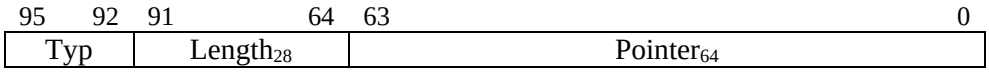
Exceptions: none

Notes:

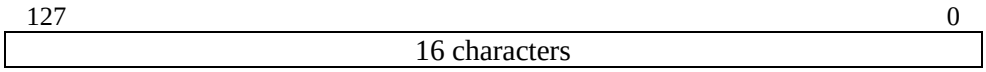
String Operations

Representations

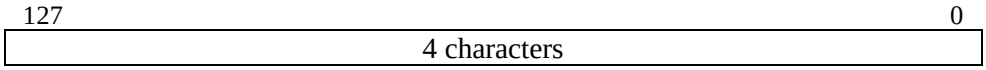
Strings



UTF8 Chars



UTF32 Chars



CHRNDX – Character Index

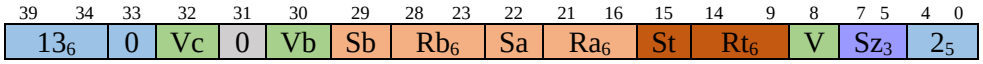
Description:

This instruction searches Ra, which is treated as an array of characters, for a character value specified by Rb and places the index of the character into the target register Rt. If the character is not found -1 is placed in the target register. A common use would be to search for a null byte. The index result may vary from -1 to +15 for UTF8 characters or -1 to +3 for UTF32 characters. The index of the first found byte is returned (closest to zero).

Supported Operand Sizes: .b, .t

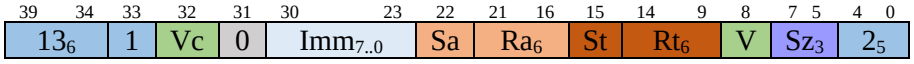
Instruction Formats:

CHRNDX Rt, Ra, Rb



Clock Cycles: 1

CHRNDX Rt, Ra, Imm



Clock Cycles: 1

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Bit Manipulation Operations

Many CPUs do not have direct support for bit-field manipulation. Instead, they rely on ordinary logical and shift operations. The benefit of having bit-field operations is that they are more code dense then performing the operations using other ALU ops.

Bitfield operations repurpose the size field for use as an opcode extension.

The beginning and end of a bitfield may be specified as either a pair of immediate constants or in a pair of registers.

CLR – Clear Bit Field

Description:

A bit field in the source operand is cleared and the result placed in the target register. The specified bit to clear is modulo the operand size.

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

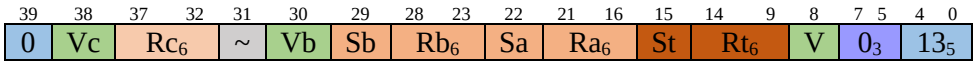
Flag Updates: none

Operation:

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

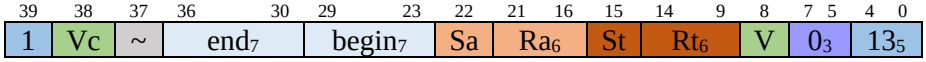
Instruction Formats:

CLR Rt, Ra, Rb, Rc



Clock Cycles: 4

CLR Rt, Ra,Offs$_7$,Wid$_7$



Clock Cycles: 4

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

COM – Complement Bit Field

Description:

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

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

Flag Updates: none

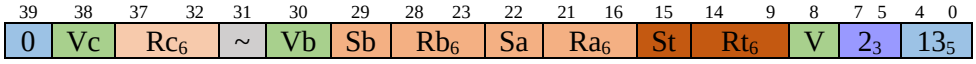
Operation:

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

Instruction Formats:

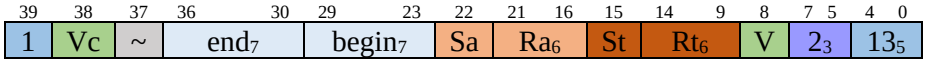
Instruction Formats:

CLR Rt, Ra, Rb



Clock Cycles: 4

CLR Rt, Ra,Offs$_7$,Wid$_7$



Clock Cycles: 4

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

DEP – Deposit Bit Field

Description:

A source operand is transferred to a bitfield in the target register.

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

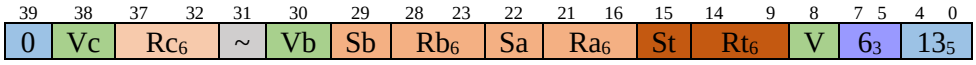
Flag Updates: none

Operation:

MB = offset
ME = offset + width
Rt[ME:MB] = Ra

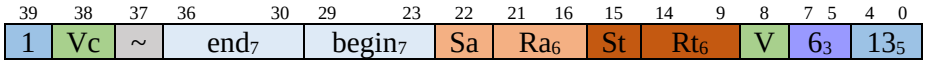
Instruction Formats:

DEP Rt, Ra, Rb, Rc



Clock Cycles: 1

DEP Rt, Ra,Offs$_7$,Wid$_7$



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

EXTS – Extract Signed Bit Field

Description:

Extract a bit field from the source operand and place the bit field in the target register. The field is sign extended.

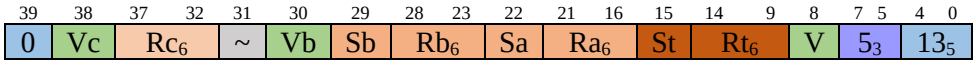
Supported Operand Sizes: .b, .w, .t, .o, .h

Operation:

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

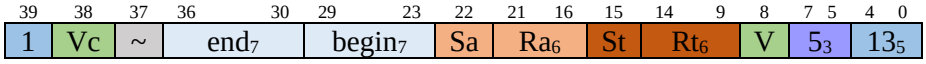
Instruction Formats:

EXTS Rt, Ra, Rb



Clock Cycles: 1

EXTS Rt, Ra,Offs$_7$,Wid$_7$



Clock Cycles: 1

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

EXTU – Extract Bit Field

Description:

Extract a bit field from the source operand and place the bit field in the target register. The field is zero extended.

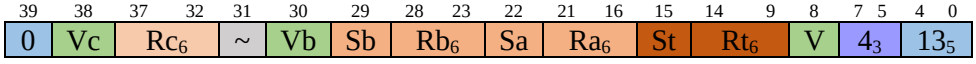
Supported Operand Sizes: .b, .w, .t, .o

Operation:

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

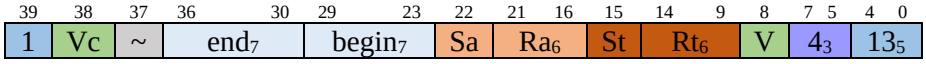
Instruction Formats:

EXTU Rt, Ra, Rb, Rc



Clock Cycles: 1

EXTU Rt, Ra,Offs$_7$,Wid$_7$



Clock Cycles: 1

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SBX – Sign Bit Extend

Description:

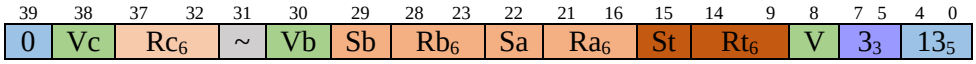
Sign extend a value beginning at a specified bit to the width specified and place the result in the target register. All registers are integer registers.

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

Operation:

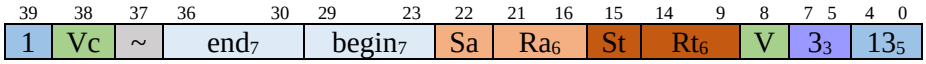
Instruction Formats:

SBX Rt, Ra, Rb, Rc



Clock Cycles: 1

SBX Rt, Ra,Offs$_7$,Wid$_7$



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

SET – Set Bit Field

Description:

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

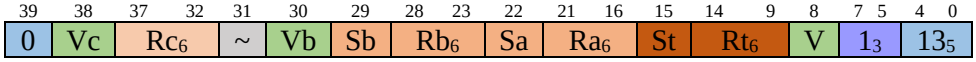
Supported Operand Sizes: .b, .w, .t, .o

Operation:

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

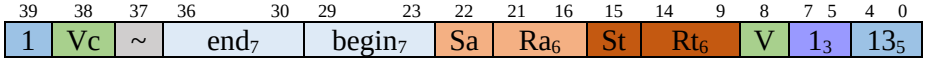
Instruction Formats:

SET Rt, Ra, Rb, Rc



Clock Cycles: 1

SET Rt, Ra,Offs$_7$,Wid$_7$



Clock Cycles: 1

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Shift and Rotate Operations

Shift instructions can take the place of some multiplication and division instructions. Some architectures provide shifts that shift only by a single bit. Others use counted shifts, the original 80x88 used multiple clock cycles to shift by an amount stored in the CX register. Table888 and Thor use a barrel shifter to allow shifting by an arbitrary amount in a single clock cycle. Shifts are infrequently used, and a barrel (or funnel) shifter is relatively expensive in terms of hardware resources.

ASL – Arithmetic Shift Left

Description:

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

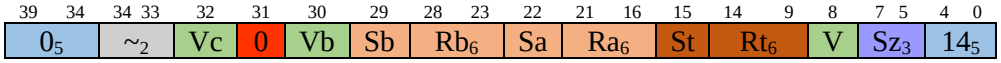
Supported Operand Sizes: .b, .w, .t, .o

Operation:

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

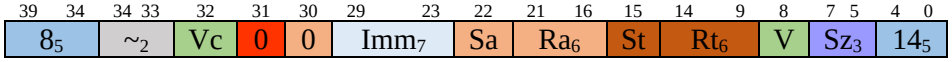
Instruction Formats:

ASL Rt, Ra, Rb



Clock Cycles: 1

ASL Rt, Ra, Imm_7



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ASR – Arithmetic Shift Right

Description:

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

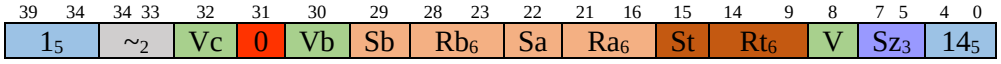
Supported Operand Sizes: .b, .w, .l

Operation:

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

Instruction Formats:

ASR Rt, Ra, Rb



Clock Cycles: 1

ASR Rt, Ra, Imm_7

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

LSL – Logical Shift Left

Description:

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

This instruction may be used to generate a bitmask by setting N to one, and shifting a zero.

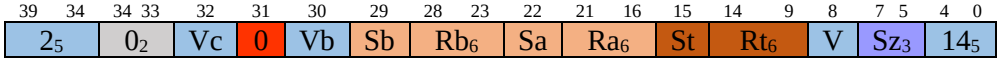
Supported Operand Sizes: .b, .w, .l

Operation:

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

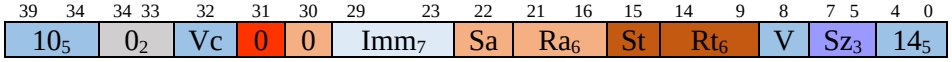
Instruction Formats:

LSL Rt, Ra, Rb



Clock Cycles: 1

LSL Rt, Ra, Imm_7



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

LSLAND – Logical Shift Left and And

Description:

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

This instruction may be used to isolate a bitfield in a target register.

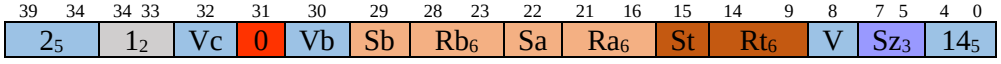
Supported Operand Sizes: .b, .w, .l

Operation:

Rt = Rt & (Ra << Rb) or Rt = Rt & (Ra << Imm)

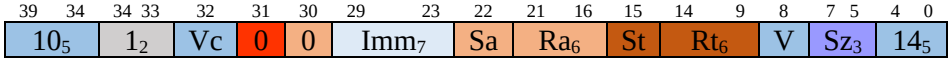
Instruction Formats:

LSL.AND Rt, Ra, Rb



Clock Cycles: 1

LSL.AND Rt, Ra, Imm$_7$



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

LSLOR – Logical Shift Left and Or

Description:

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

This instruction may be used to insert a bitfield into a target register.

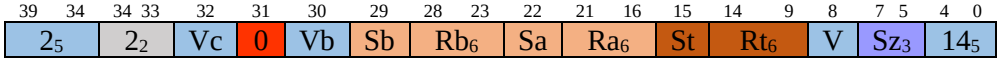
Supported Operand Sizes: .b, .w, .l

Operation:

Rt = Rt | (Ra << Rb) or Rt = Rt | (Ra << Imm)

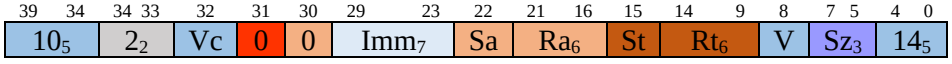
Instruction Formats:

LSL.OR Rt, Ra, Rb



Clock Cycles: 1

LSL.OR Rt, Ra, Imm$_7$



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

LSLXOR – Logical Shift Left and Exclusive Or

Description:

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

This instruction may be used to insert or invert a bitfield in a target register.

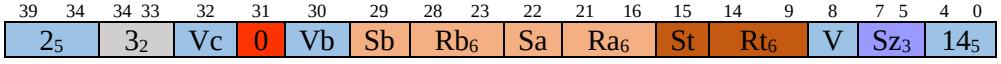
Supported Operand Sizes: .b, .w, .l

Operation:

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

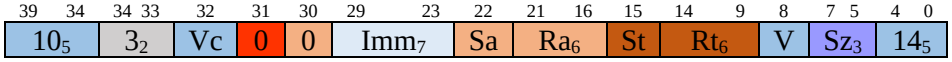
Instruction Formats:

LSL.XOR Rt, Ra, Rb



Clock Cycles: 1

LSL.XOR Rt, Ra, Imm_7



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

LSR – Logical Shift Right

Description:

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

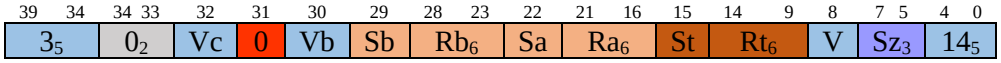
Supported Operand Sizes: .b, .w, .t, .o

Operation:

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

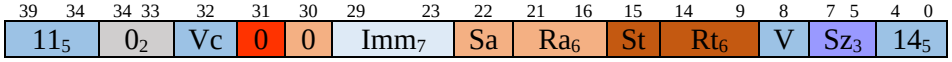
Instruction Formats:

LSL Rt, Ra, Rb



Clock Cycles: 1

LSL Rt, Ra, Imm_7



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ROL – Rotate Left

Description:

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

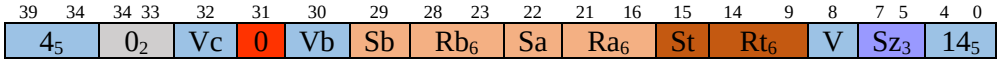
Supported Operand Sizes: .b, .w, .t, .o

Operation:

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

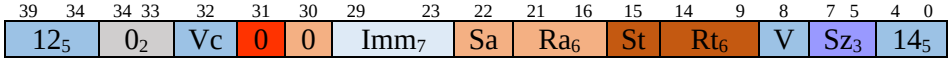
Instruction Formats:

ROL Rt, Ra, Rb



Clock Cycles: 1

ROL Rt, Ra, Imm_7



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ROR – Rotate Right

Description:

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

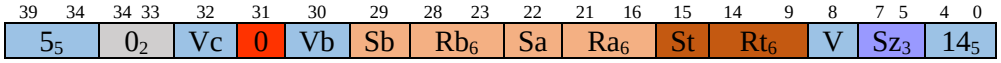
Supported Operand Sizes: .b, .w, .l

Operation:

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

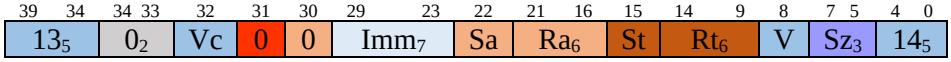
Instruction Formats:

ROR Rt, Ra, Rb



Clock Cycles: 1

ROR Rt, Ra, Imm_7



Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Flow Control Instructions

The Branch Set

One of the first things the author looks at when evaluating an ISA is the branch set. Is it semi-sensible or non-sense ? Branches may represent up to one quarter of instruction executed. Branches are one item that should be well done in an architecture. What conditions will the processor branch on ? Is it a simple branch on zero / non-zero test or are there more complex conditions available ? What the branch set supports impacts what other instructions need to be available in the architecture. If branching only supports a zero / non-zero test, then other instructions must be present to setup the branch test. In the DLX architecture for instance, there are a set of 'set' instructions that set a register to a one or zero based on a condition. After a set instruction is done, then a conditional branch may occur. Many architectures include a compare instruction(s). For instance, the MMIX architecture includes both signed (CMP) and unsigned compare (CMPU) instructions that set the value of a register to -1, 0, or 1 for less than, equal, or greater than another register. The same paradigm was used for the Raptor64 processor. For the Thor processor there is a fairly standard set of branches. Because the instruction set is wide enough, two registers may be compared during the branch.

Bcc – Conditional Branch

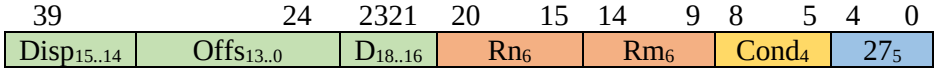
Bcc Rm, Rn, label

Description:

Branch if the condition is met. The condition is a relationship between either two registers or a register and an immediate value. The displacement is relative to the address of the branch instruction. The branch range is +/- 256kB.

A postfix instruction containing an immediate value may follow the branch instruction, in which case the immediate is used instead of Rn. Rn should be set to zero.

Instruction Format: B



Cond4	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	a == b
1	NE	< > not equal	a <> b
2	LT	< less than	a < b
3	LE	<= less than or equal	a <= b
4	GE	>= greater than or equal	a >= b
5	GT	> greater than	a > b
6	BC	Bit clear	!a[b]
7	BS	Bit set	a[b]
8	BCI	Bit clear immediate	!a[b]
9	BSI	Bit set immediate	a[b]
10	LO / CS	< unsigned less than	a < b
11	LS	<= unsigned less than or equal	a <= b
12	HS / CC	unsigned greater than or equal	a >= b
13	HI	unsigned greater than	a > b
14	RA	Branch always	1
15	SR	Branch subroutine	1

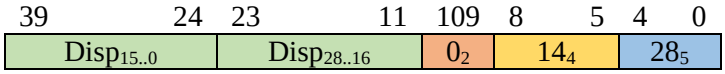
Clock Cycles: 4

BRA – Unconditional Branch

Description:

Unconditionally branch to a new program address. The displacement is relative to the address of the branch instruction. The branch range is +/- 256MB.

Instruction Format: BL2



Instruction Format: JSR

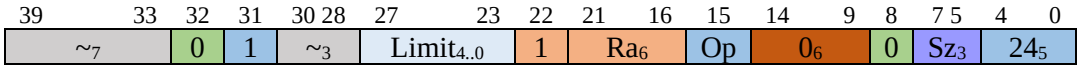
BRA [(Ra)],Limit

This form of the instruction branches to an address formed by adding the program counter to a value loaded from a table following the instruction inline in memory. The address loaded from must be no more than limit bytes after the address of the instruction or an exception will occur.

Operation:

table address = next pc + (Ra * scale)

if table address > next pc + limit then table limit exception



Clock Cycles: 1

Op	Operation
0	Load PC LSBs
1	Add to PC

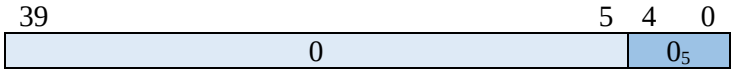
Clock Cycles: 3

BRK – Breakpoint

Description:

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

Instruction Format:



Operation:

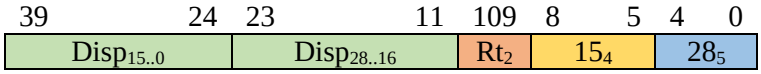
The program counter and the status register are pushed on an internal stack. Next the BRK vector is fetched from the exception vector table and jumped to.

BSR – Branch to Subroutine

Description:

Branch to a subroutine placing the address of the next instruction in a register. The displacement is relative to the address of the branch instruction. The branch range is +/- 256MB.

Instruction Format: BL2



Instruction Format: JSR

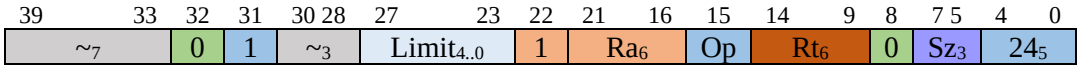
BSR [(Ra)],Limit

This form of the instruction branches to an address formed by adding the program counter to a value loaded from a table following the instruction inline in memory. The address loaded from must be no more than limit bytes after the address of the instruction or an exception will occur.

Operation:

table address = next pc + (Ra * scale)

if table address > next pc + limit then table limit exception



Clock Cycles: 4

Op	Operation
0	Load PC LSBs
1	Add to PC

Clock Cycles: 3

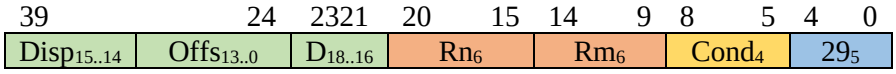
DBcc – Decrement and Branch

DBcc Rm, Rn, label

Description:

Decrement the loop counter and branch if the condition is false and the loop counter is not equal to minus one. The displacement is relative to the address of the branch instruction. The branch range is +/- 256kB.

Instruction Format:



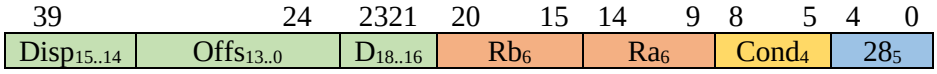
FBcc – Conditional Branch

FBcc Rm, Rn, label

Description:

Branch if the condition is met. The condition is a relationship between either two registers or a register and an immediate value. The displacement is relative to the address of the branch instruction. The branch range is +/- 256kB.

Instruction Format: RR



Cond$_4$	Mnem.	Meaning	Test
Cond$_4$	Mnem.	Meaning	Test
		Float Compare Results	
0	FBEQ	equal	!nan & eq
1	FBNE	not equal	!eq
2	FBGT	greater than	!nan & !eq & !lt & !inf
3	FBUGT	Unordered or greater than	Nan || (!eq & !lt & !inf)
4	FBGE	greater than or equal	Eq || (!nan & !lt & !inf)
5	FBUGE	Unordered or greater than or equal	Nan || (!lt || eq)
6	FBLT	Less than	Lt & (!nan & !inf & !eq)
7	FBULT	Unordered or less than	Nan | (!eq & lt)
8	FBLE	Less than or equal	Eq | (lt & !nan)
9	FBULE	unordered less than or equal	Nan | (eq | lt)
10	FBGL	Greater than or less than	!nan & (!eq & !inf)
11	FBUGL	Unordered or greater than or less than	Nan | !eq
12	FBORD	Greater than less than or equal / ordered	!nan
13	FBUN	Unordered	Nan
Cond$_5$	Mnem.	Meaning	Test
14			
15			

Clock Cycles: 4

JMP – Jump to Address

Description:

Compute the effective address and jump to it. If Ra=53 then the program counter is used. If the indirection bit 'I' of the instruction is set then load the address from memory specified by the effective address and jump to it.

Operation:

PC = Ra + Rb or PC = Ra + Imm

Clock Cycles:

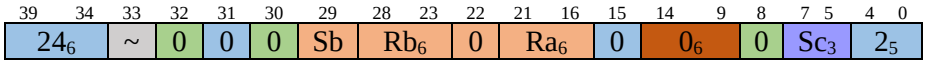
Execution Units: All Integer ALU's

Exceptions: none

Notes:

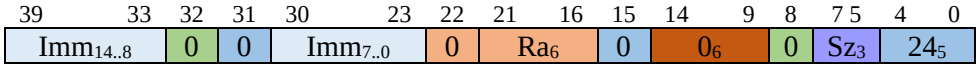
Instruction Formats:

JMP d(Ra, Rb)



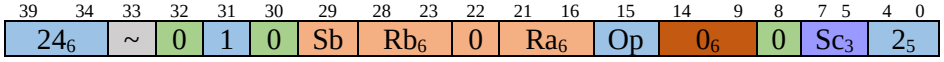
Clock Cycles: 1

JMP Imm$_{15}$ (Ra)



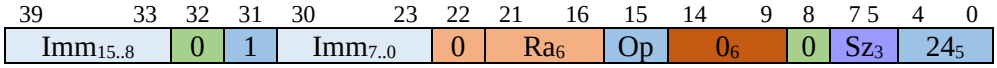
Clock Cycles: 1

JMP [d(Ra, Rb)]



Clock Cycles: 1

JMP [Imm$_{15}$ (Ra)]



Clock Cycles: 1

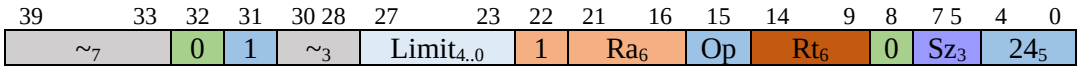
BRA [(Ra)],Limit

This form of the instruction branches to an address formed by adding the program counter to a value loaded from a table following the instruction inline in memory. The address loaded from must be no more than limit bytes after the address of the instruction or an exception will occur.

Operation:

table address = next pc + (Ra * scale)

if table address > next pc + limit then table limit exception



Clock Cycles: 1

Op	Operation
0	Load PC LSBs
1	Add to PC

JSR – Jump to Subroutine

Description:

Compute the effective address and jump to it. The address of the instruction is stored in a register. If Ra=53 then the program counter is used. If the indirection bit 'I' of the instruction is set then load the address from memory specified by the effective address and jump to it.

Flag Updates:

None.

Operation:

Rt = PC

PC = Ra + Rb or PC = Ra + Imm

Clock Cycles:

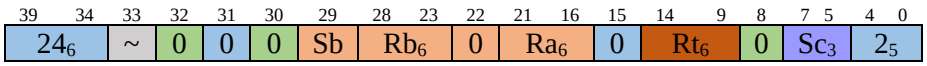
Execution Units: All Integer ALU's

Exceptions: none

Notes:

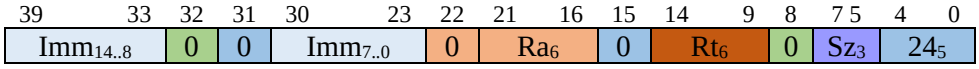
Instruction Formats:

JSR d(Ra, Rb)



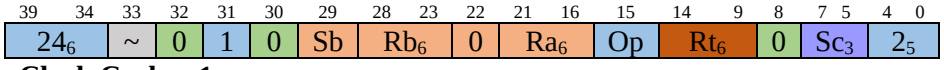
Clock Cycles: 1

JSR Imm$_{15}$ (Ra)



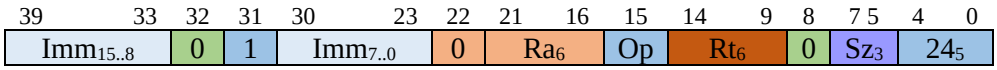
Clock Cycles: 1

JSR [d(Ra, Rb)]



Clock Cycles: 1

JSR [Imm$_{15}$ (Ra)]



Clock Cycles: 1

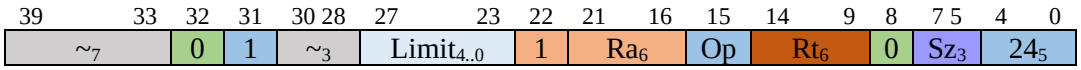
BSR [(Ra)],Limit

This form of the instruction branches to an address formed by adding the program counter to a value loaded from a table following the instruction inline in memory. The address loaded from must be no more than limit bytes after the address of the instruction or an exception will occur.

Operation:

table address = next pc + (Ra * scale)

if table address > next pc + limit then table limit exception



Clock Cycles: 1

Op	Operation
0	Load PC LSBs
1	Add to PC

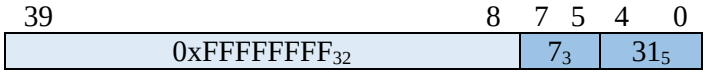
NOP – No Operation

NOP

Description:

This instruction does not perform any operation. Ty_3 0 to 2 indicates a postfix instruction, and these codes should not be used for other NOPs. The value 3 to 6 for Ty_3 are reserved. The NOP operation is an opcode of all ones.

Instruction Format:



RTD – Return from Subroutine, Deallocate

Description:

Return from subroutine and deallocate stack. Add two source operands and place the sum in the target register. All registers are treated as integer registers. Arithmetic is signed twos-complement values. The program counter is loaded with the value of the specified link register.

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

Operation:

Rt = Ra + Rb or Rt = Ra + Imm

PC = Lr

Clock Cycles:

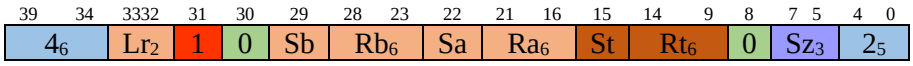
Execution Units: All Integer ALU's

Exceptions: none

Notes:

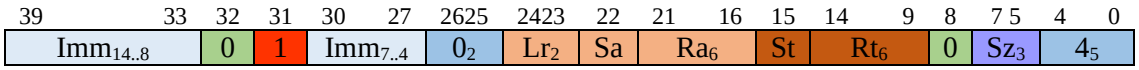
Instruction Formats:

RTD Rt, Ra, Rb – Register direct



Clock Cycles: 1

RTD Rt,Ra,Imm_{16}



Clock Cycles: 1

RTS – Return from Subroutine

Description:

Return from subroutine. Load the program counter with the contents of the specified link register.

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

Operation:

Rt = Ra + Rb or Rt = Ra + Imm

Clock Cycles:

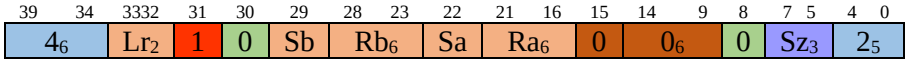
Execution Units: All Integer ALU's

Exceptions: none

Notes:

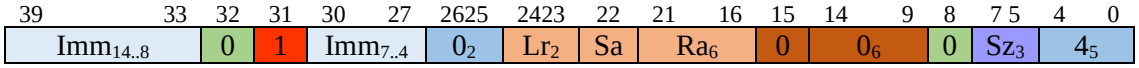
Instruction Formats:

RTS Rt



Clock Cycles: 1

RTS Rt

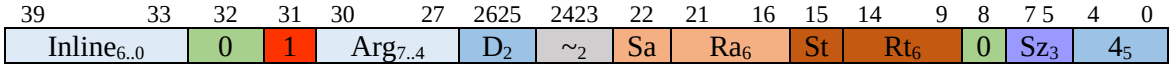


Clock Cycles: 1

RTE – Return From Exception

Instruction Formats:

RTE Imm$_7$



Clock Cycles: 1

Field Description:

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

D_2 specifies the number of internal stack entries to unstack. It may be used to perform a multi-level return. Legal values for D are 1 or 2. (0 is the RTD instruction). In most cases a single entry is unstacked. If two entries are unstacked a two-up level return will occur.

Operation:

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

TRAP – Trap

Description:

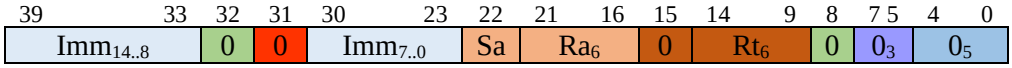
Execute trap. The data field is loaded into the specified target register, Rt. The trap number to execute comes from the contents of register Ra or an immediate value encoded in the instruction. The trap number must be between 1 and 511. Trap numbers below 64 are reserved for the system. Trap numbers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

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

Trap routines should return using an RTE instruction.

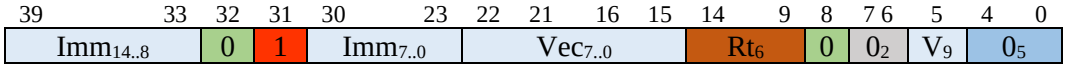
Instruction Format:

TRAP Rt,Ra,#Imm$_{15}$



Clock Cycles: 1

TRAP Rt, #Vec, #Data



Clock Cycles: 1

Operation:

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

Memory Operations

AMADD - Addition

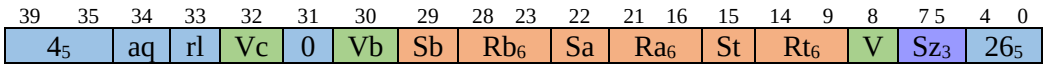
Description:

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

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

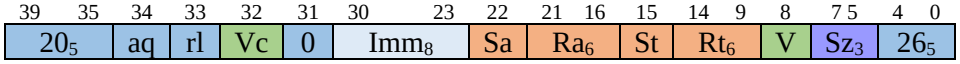
Instruction Formats: AMO

AMADD Rt, Rb, [Ra]



Clock Cycles:

AMADD Rt, imm, [Ra]



Clock Cycles:

AMAND – Bitwise And

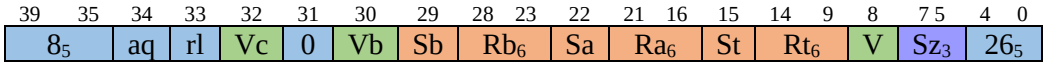
Description:

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

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

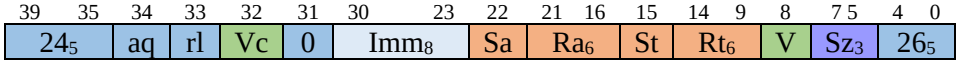
Instruction Formats: AMO

AMAND Rt, Rb, [Ra]



Clock Cycles:

AMAND Rt, imm, [Ra]



Clock Cycles:

AMASL – Arithmetic Shift Left

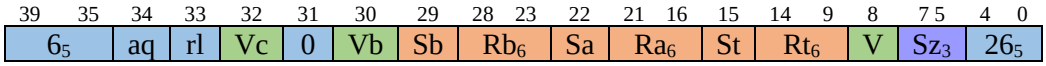
Description:

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

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

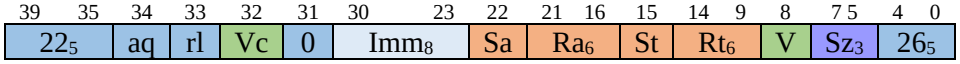
Instruction Formats: AMO

AMASL Rt, Rb, [Ra]



Clock Cycles:

AMASL Rt, imm, [Ra]



Clock Cycles:

AMEOR – Bitwise Exclusive Or

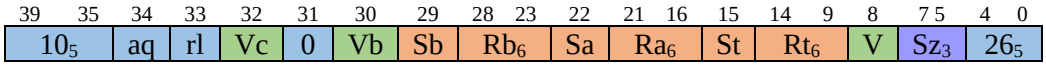
Description:

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

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

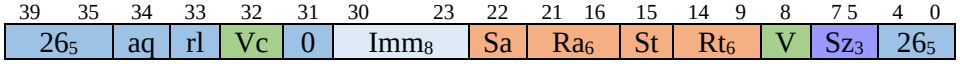
Instruction Formats: AMO

AMEOR Rt, Rb, [Ra]



Clock Cycles:

AMEOR Rt, imm, [Ra]



Clock Cycles:

AMLSR – Logical Shift Right

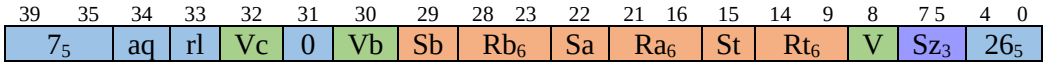
Description:

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

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

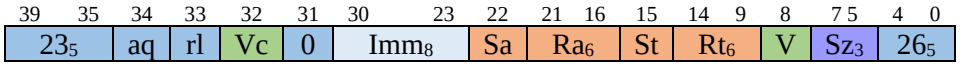
Instruction Formats: AMO

AMLSR Rt, Rb, [Ra]



Clock Cycles:

AMLSR Rt, imm, [Ra]



Clock Cycles:

AMMIN - Minimum

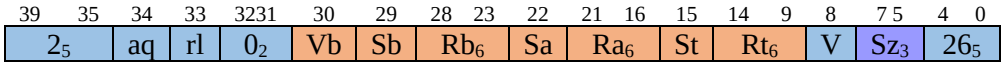
Description:

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

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

Instruction Formats: AMO

AMMIN Rt, Rb, [Ra]



Clock Cycles:

AMMINU - Minimum

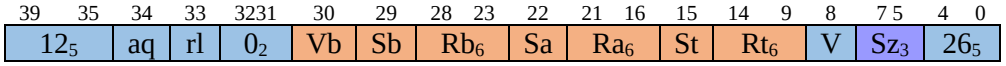
Description:

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

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

Instruction Formats: AMO

AMMINU Rt, Rb, [Ra]



Clock Cycles:

AMOR – Bitwise Or

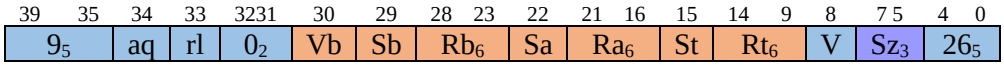
Description:

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

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

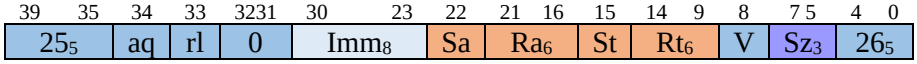
Instruction Formats: AMO

AMOR Rt, Rb, [Ra]



Clock Cycles:

AMOR Rt, imm, [Ra]



Clock Cycles:

CACHE <cmd>,<ea>

Description:

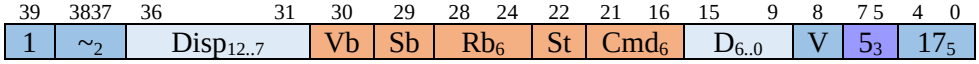
Issue command to cache controller.

Supported Operand Sizes: N/A

Sz_3	Ext.	Operand
0		LEA
1		
2	.t	STPTR
3	.o	STPTR
4	.h	STPTR
5		CACHE
6		
7		Indexed

Instruction Formats: RINDS

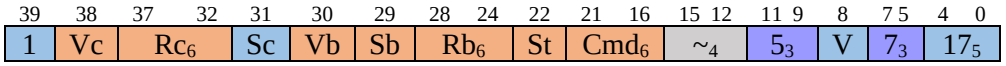
CACHE cmd, d(Rb)



Clock Cycles:

Instruction Formats: NDXS

CACHE cmd, d(Rb,Rc*Sc)



Clock Cycles:

Notes:

Cmd_6	Cache	
???000	Ins.	Invalidate cache
???001	Ins.	Invalidate line
???100	TLB	Invalidate TLB
???101	TLB	Invalidate TLB entry
000???	Data	Invalidate cache
001???	Data	Invalidate line
010???	Data	Turn cache off
011???	Data	Turn cache on

CMPXCHG – Compare and Exchange

Description:

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

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

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

Instruction Formats: CMPXCHG

CMPXCHG Rt, Rb, Rc, [Ra]

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

Clock Cycles:

Notes:

FLOAD Rn,<ea>

Description:

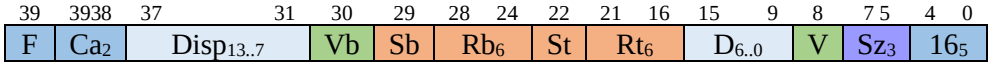
Load register Rt from floating-point source. The source value is converted to the machine width; 128-bit quad precision. No rounding needs to take place; the smaller source can always be guaranteed to fit into the target register.

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

Sz_3	Ext.	Operand
0		reserved
1	.h	16-bit half
2	.s	32-bit single
3	.d	64-bit double
4	.q	128-bit quad
5		reserved
6		reserved
7		reserved

Instruction Formats: RINDL

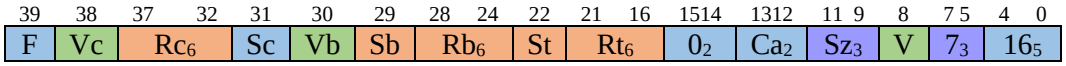
FLOAD Rt, d(Rb)



Clock Cycles:

Instruction Formats: NDXL

FLOAD Rt, d(Rb,Rc*Sc)



Clock Cycles:

Notes:

F	Load operand type
0	Integer
1	Floating point

Ca_2	Policy	Qualifier	Comment
0	none	.io	Always read from main memory or I/O
1	Read	.rd	Read from cache if in cache, otherwise read main memory
2	Read, allocate	.rda	Allocate storage in cache, read from cache
3			Reserved

FSTORE Ra,<ea>

Description:

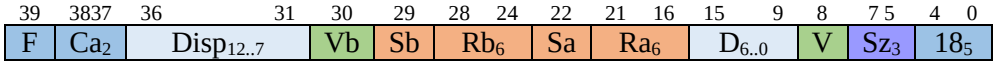
Store register Ra to destination. The register is converted from quad precision to the storage precision.

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

Sz_3	Ext.	Operand
0		Reserved
1	.h	16-bit half
2	.s	32-bit single
3	.d	64-bit double
4	.q	128-bit quad
5		
6		
7		Indexed

Instruction Formats: RINDS

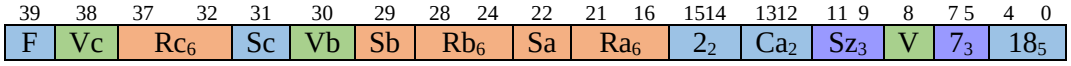
FSTORE Ra, d(Rb)



Clock Cycles:

Instruction Formats: NDXS

FSTORE Ra, d(Rb,Rc*Sc)



Clock Cycles:

Notes:

F	Store operand type
0	Integer
1	Floating point

Ca_2	Policy	Comment
0	Write through	Always write through to main memory
1	Writeback	Store to main memory only when data not in cache
2	Write through, write allocate	Write to main memory, and allocate in cache
3	Write back, write allocate	Allocate in cache, write to cache

LA Ra,<ea>

Description:

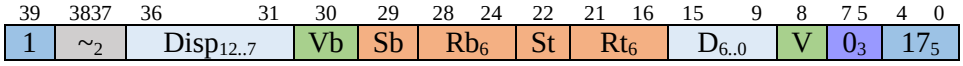
Load address into target register. The address is calculated as if a memory operation were occurring, then it is loaded into the target register.

Supported Operand Sizes: N/A

Sz_3	Ext.	Operand
0		LA
1		
2	.t	STPTR
3	.o	STPTR
4	.h	STPTR
5		CACHE
6		
7		Indexed

Instruction Formats: RINDS

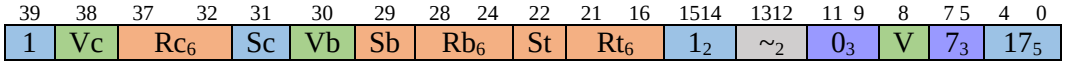
LA Rt, d(Rb)



Clock Cycles:

Instruction Formats: NDXS

LA Rt, d(Rb,Rc*Sc)



Clock Cycles:

Notes:

LOAD Rn,<ea>

Description:

Load register Rt from source. The source value is sign extended to the machine width. Loading register r53, the stack canary placeholder, will cause a check trap if the value loaded is not equal to the current value of the stack canary register.

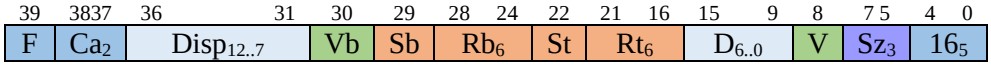
The default cache policy is read-allocate, .rda, if not specified.

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

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa
4	.h	128-bit Hexi
5		256-bit
6	.g	512-bit group
7		Indexed/group

Instruction Formats: RINDL

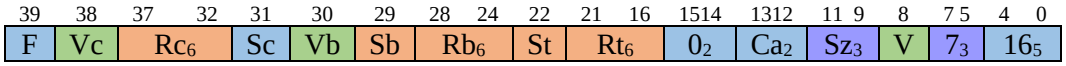
LOAD Rt, d(Rb)



Clock Cycles:

Instruction Formats: NDXL

LOAD Rt, d(Rb,Rc*Sc)



Clock Cycles:

Notes:

F	Load operand type
0	Integer
1	Floating point

Ca_2	Policy	Qualifier	Comment
0	none	.io	Always read from main memory or I/O
1	Read	.rd	Read from cache if in cache, otherwise read main memory
2	Read, allocate	.rda	Allocate storage in cache, read from cache
3			Reserved

LOADG Gn,<ea>

Description:

Load a group of eight scalar registers from source. The load operation may be masked so that only specific registers of the group are loaded. The instruction will always read 512-bits from memory.

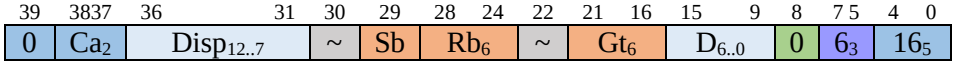
Lower Group (bits 0 to 63)		
Gn	Group	Registers
0	AG0	R0 to R7
1	TG0	R8 to R15
2	SG0	R16 to R23
3	SG1	R24 to R31
4	VMG	R32 to R39
5	G5	R40 to R47
6	G6	R48 to R55
7	G7	R56 to R63

Upper Group (bits 64 to 127)		
Gn	Group	Registers
8	UAG0	R0 to R7
9	UTG0	R8 to R15
10	USG0	R16 to R23
11	USG1	R24 to R31
12	VMG	R32 to R39
13	UG5	R40 to R47
13	UG6	R48 to R55
14	UG7	R56 to R63

Supported Operand Sizes: .g

Instruction Formats: RINDL

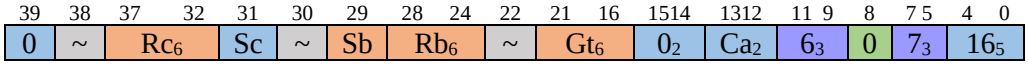
LOADG Rt, d(Rb)



Clock Cycles:

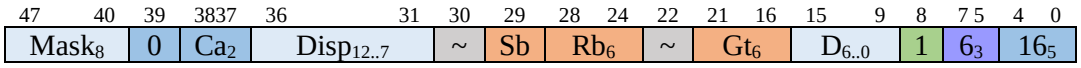
Instruction Formats: NDXL

LOADG Rt, d(Rb,Rc*Sc)



Clock Cycles:

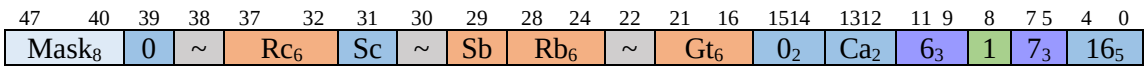
LOADG Rt, d(Rb)



Clock Cycles:

Instruction Formats: NDXL

LOADG Rt, d(Rb,Rc*Sc)



Clock Cycles:

Notes:

LOADZ Rn,<ea>

Description:

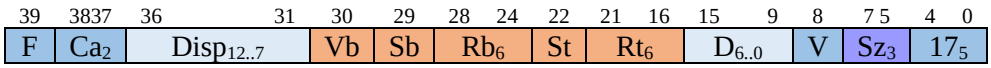
Load register Rt from source. The source value is zero extended to the machine width. Loading register r53, the stack canary placeholder, will cause a check trap if the value loaded is not equal to the current value of the stack canary register.

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

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa
4	.h	128-bit Hexi
5		
6		
7		indexed

Instruction Formats: RINDL

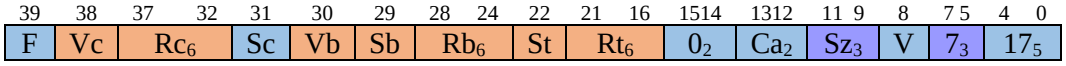
LOADZ Rt, d(Rb)



Clock Cycles:

Instruction Formats: NDXL

LOADZ Rt, d(Rb,Rc*Sc)



Clock Cycles:

Notes:

F	Store operand type
0	Integer
1	LEA, CACHE, STPTR

Ca_2	Policy	Qualifier	Comment
0	none	.io	Always read from main memory or I/O
1	Read	.rd	Read from cache if in cache, otherwise read main memory
2	Read, allocate	.rda	Allocate storage in cache, read from cache
3			Reserved

STORE Ra,<ea>

Description:

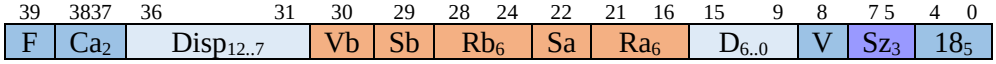
Store register Ra to destination. The default cache policy is write-through, .wt.

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

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa
4	.h	128-bit
5		256-bit
6	.g	512-bit group
7		indexed

Instruction Formats: RINDS

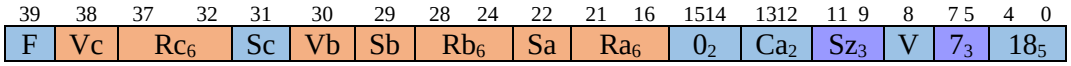
STORE Ra, d(Rb)



Clock Cycles:

Instruction Formats: NDXS

STORE Ra, d(Rb,Rc*Sc)



Clock Cycles:

Notes:

F	Store operand type
0	Integer
1	Floating point

Ca_2	Policy	Qualifier	Comment
0	Write through	.wt	Always write through to main memory
1	Writeback	.wb	Store to main memory only when data not in cache
2	Write through, write allocate	.wta	Write to main memory, and allocate in cache
3	Write back, write allocate	.wba	Allocate in cache, write to cache

Stores using write through will always write through to main memory, and will also update the cache if the data is in the cache. If allocating is specified, the write operation will allocate storage in the cache for data.

Stores using writeback will update memory only when there is a cache collision and new data needs to be stored in the cache. Otherwise, references will be to and from the cache.

STOREPTR Ra,<ea>

Description:

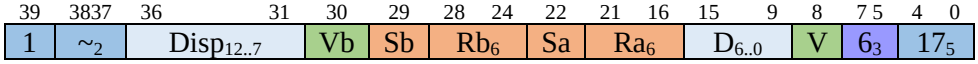
Store a pointer contained in register Ra to destination.

Supported Operand Sizes: N/A

Sz_3	Ext.	Operand
0		LEA
1		
2	.t	STPTR
3	.o	STPTR
4	.h	STPTR
5		CACHE
6		
7		indexed

Instruction Formats: RINDS

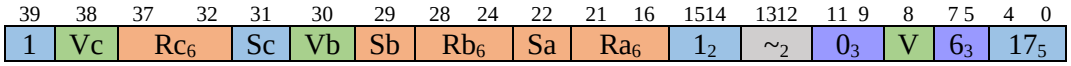
STOREPTR Ra, d(Rb)



Clock Cycles:

Instruction Formats: NDXS

STOREPTR Ra, d(Rb,Rc*Sc)



Clock Cycles:

Notes:

STOREG Gt,<ea>

Description:

Store a register group to destination. 512-bits are always stored. Zeros may be written instead of specific registers via a mask. One use is to initialize large blocks of memory to zero.

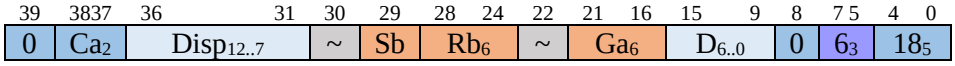
Lower Group (bits 0 to 63)		
Gn	Group	Registers
0	AG0	R0 to R7
1	TG0	R8 to R15
2	SG0	R16 to R23
3	SG1	R24 to R31
4	VMG	R32 to R39
5	G5	R40 to R47
6	G6	R48 to R55
7	G7	R56 to R63

Upper Group (bits 64 to 127)		
Gn	Group	Registers
8	UAG0	R0 to R7
9	UTG0	R8 to R15
10	USG0	R16 to R23
11	USG1	R24 to R31
12	VMG	R32 to R39
13	UG5	R40 to R47
14	UG6	R48 to R55
15	UG7	R56 to R63

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

Instruction Formats: RINDL

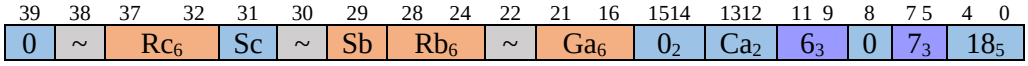
STOREG Ra, d(Rb)



Clock Cycles:

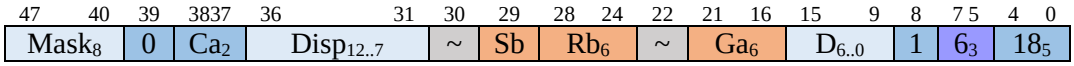
Instruction Formats: NDXS

STOREG Ra, d(Rb,Rc*Sc)



Clock Cycles:

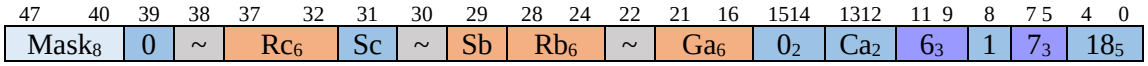
STOREG Ra, d(Rb), Mask



Clock Cycles:

Instruction Formats: NDXS

STOREG Ra, d(Rb,Rc*Sc), Mask



Clock Cycles:

Notes:

Compare and Exchange

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

Load add and store:

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

Load or and store

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

Load and complement and store

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

STORE_PAIR Rb, Rc, d[Ra]

Description:

Store register pair to destination.

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

Sz_3	Ext.	Operand
0	.b	8-bit Byte
1	.w	16-bit Wyde
2	.t	32-bit Tetra
3	.o	64-bit Octa

4	.c	24-bit
5	.p	40-bit
6	.n	96-bit
7		group

Instruction Formats: dRa

STORE Rb, Rc, d(Ra)

Clock Cycles:

Notes:

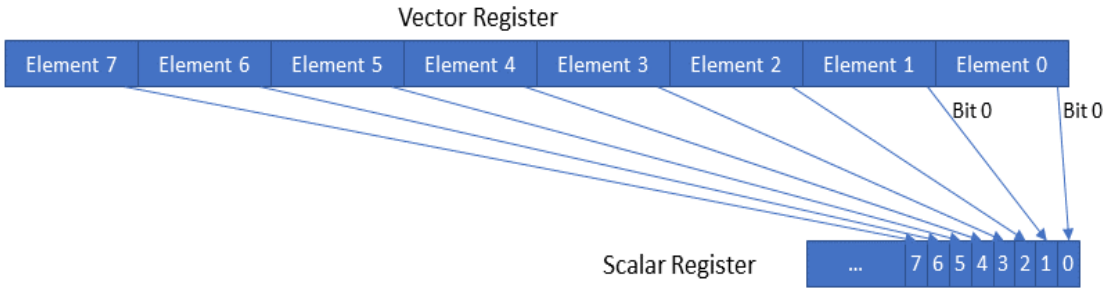
Vector Specific Instructions

V2BITS

Description

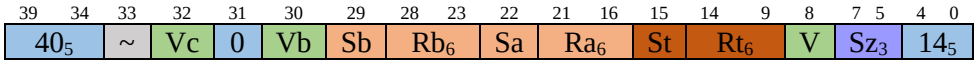
Convert Boolean vector to bits. A bit specified by Rb or an immediate of each vector element is copied to the bit corresponding to the vector element in the target register. The target register is a scalar register. Usually, Rb would be zero so that the least significant bit of the vector is copied.

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



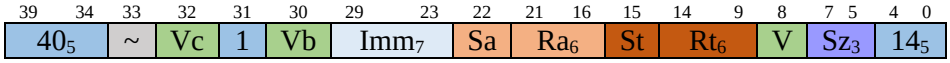
Instruction Format: R2

V2BITS Rt, Ra, Rb



Clock Cycles: 1

V2BITS Rt, Ra, Imm$_7$



Clock Cycles: 1

Operation

For x = 0 to VL-1

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

Exceptions: none

Example:

```
cmp v1,v2,v3       ; compare vectors v2 and v3
v2bits vm1,v1,#8   ; move NE status to bits in m1
vadd v4,v5,v6,vm1        ; perform some masked vector operations
vmuls v7,v8,v9,vm1
vadd v7,v7,v4,vm1
```

Cryptographic Accelerator Instructions

AES64DS – Final Round Decryption

Description:

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

Integer Instruction Format: R3

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

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

Operation:

Rt = Ra & Rb

Exceptions: none

AES64DSM – Middle Round Decryption

Description:

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

Integer Instruction Format: R3

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

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

Operation:

Rt = Ra & Rb

Exceptions: none

AES64ES – Final Round Encryption

Description:

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

Integer Instruction Format: R3

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

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

Operation:

Rt = Ra & Rb

Exceptions: none

AES64ESM – Middle Round Encryption

Description:

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

Integer Instruction Format: R3

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

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

Operation:

Rt = Ra & Rb

Exceptions: none

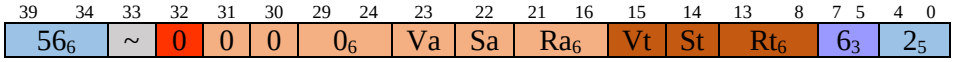
SHA256SIG0

Description:

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

Instruction Format: R2

SHA256SIG0 Rt, Ra – Register direct



Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

Exceptions: none

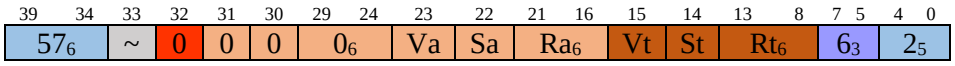
SHA256SIG1

Description:

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

Instruction Format: R2

SHA256SIG1 Rt, Ra – Register direct



Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

Exceptions: none

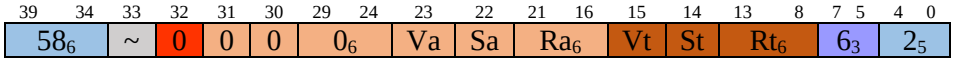
SHA256SUM0

Description:

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

Instruction Format: R2

SHA256SUM0 Rt, Ra – Register direct



Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

Exceptions: none

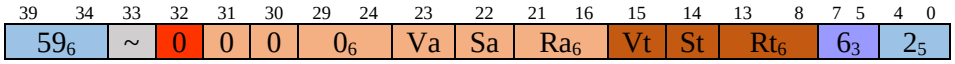
SHA256SUM1

Description:

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

Instruction Format: R2

SHA256SUM1 Rt, Ra – Register direct



Operation:

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

Execution Units: ALU #0

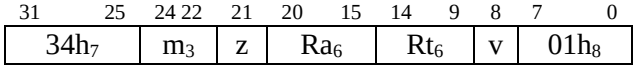
Exceptions: none

SHA512SIG0

Description:

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

Instruction Format: R1



Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

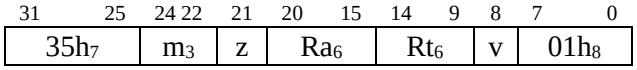
Exceptions: none

SHA512SIG1

Description:

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

Instruction Format: R1



Clock Cycles: 1

Operation:

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

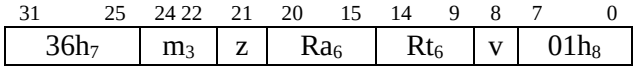
Execution Units: ALU #0

Exceptions: none

SHA512SUM0

Description:

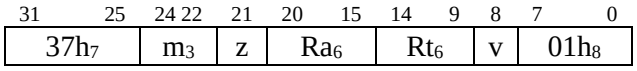
Instruction Format: R1



SHA512SUM1

Description:

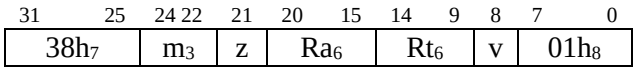
Instruction Format: R1



SM3P0

Description:

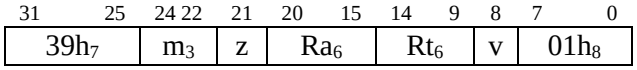
Instruction Format: R1



SM3P1

Description:

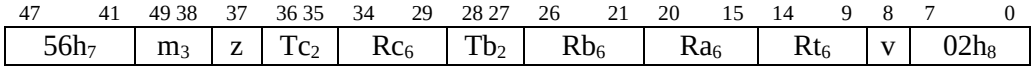
Instruction Format: R1



SM4ED

Description:

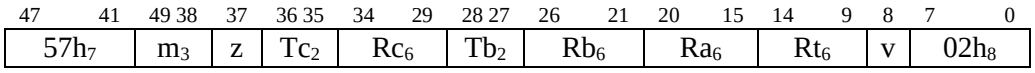
Instruction Format: R3



SM4KS

Description:

Instruction Format: R3



Modifiers

ATOM

Description:

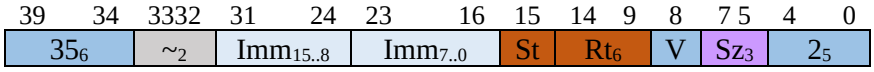
Treat the following sequence of instructions as an “atom”. Rt specifies the register results are to be written to.

Disable interrupts for the following instructions.

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

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

Instruction Format:



Assembler Syntax:

Example:

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

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

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

CARRY

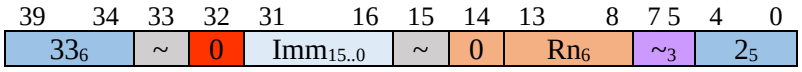
Description:

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

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

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

Instruction Format:



Assembler Syntax:

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

Example:

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

VMASK

Description:

Apply the vector masking to following instructions according to a bit mask. Note that postfixes do not count as instructions. The mask register for the next four instructions may be specified. Note that the value in the mask register may be inverted. To have all mask bits enabled specify an inverted r0 as the mask register.

MASK Modifier Scope	Mask Bit	
	0 to 6	Instruction one
	7 to 13	Instruction two
	14 to 20	Instruction three
	21 to 27	Instruction four

Instruction Format:

39 34	33	32	31 26	25	24 19	18	17 12	11	10 5	4 0
34_6	~	S3	Msk3	S2	Msk2	S1	Msk1	S0	Msk0	2_5

Assembler Syntax:

sExample:

```
VMASK s0,s1,s2,s3
ADD v6,v3,v7          ; vector mask reg s0
ADD v6,v6,#1234       ; vector mask reg s1
ADD v6,v2,v1          ; vector mask reg s2
ADD v6,v6,#456        ; vector mask reg s3
VMASK t0,t1,t2
ADD v7,v6,#456        ; vector mask reg t0
ADD v8,v7,#987        ; vector mask reg t1
MUL v8,v9,v10         ; vector mask reg t2
```

PRED

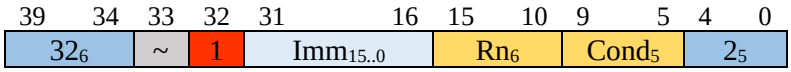
Description:

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

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

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

Instruction Format:



Assembler Syntax:

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

Example:

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

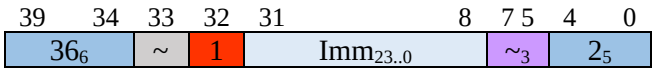
ROUND

Description:

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

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

Instruction Format:



Assembler Syntax:

Example:

MPU Hardware
PIC – Programmable Interrupt Controller

Overview

The programmable interrupt controller manages interrupt sources in the system and presents an interrupt signal to the cpu. The PIC may be used in a multi-CPU system as a shared interrupt controller. The PIC can guide the interrupt to the specified core. If two interrupts occur at the same time the controller resolves which interrupt the cpu sees. While the CPU's interrupt input is only level sensitive the PIC may process interrupts that are either level or edge sensitive. the PIC is a 32-bit I/O device.

System Usage

There is just a single interrupt controller in the system. It supports 31 different interrupt sources plus a non-maskable interrupt source.

The PIC is located at an address determined by BAR0 in the configuration space.

Priority Resolution

Interrupts have a fixed priority relationship with interrupt #1 having the highest priority and interrupt #31 the lowest. Note that interrupt priorities are only effective when two interrupts occur at the same time.

Config Space

A 256-byte config space is supported. Most of the config space is unused. The only configuration is for the I/O address of the register set.

Regno	Width	R/W	Moniker	Description		
000	32	RO	REG_ID	Vendor and device ID		
004	32	R/W				
008	32	RO				
00C	32	R/W				
010	32	R/W	REG_BAR0	Base Address Register		
014	32	R/W	REG_BAR1	Base Address Register		
018	32	R/W	REG_BAR2	Base Address Register		
01C	32	R/W	REG_BAR3	Base Address Register		
020	32	R/W	REG_BAR4	Base Address Register		
024	32	R/W	REG_BAR5	Base Address Register		
028	32	R/W				
02C	32	RO		Subsystem ID		
030	32	R/W		Expansion ROM address		
034	32	RO				
038	32	R/W		Reserved		
03C	32	R/W		Interrupt		
040 to 0FF	32	R/W		Capabilities area		

REG_BAR0 defaults to $FEE20001 which is used to specify the address of the controller's registers in the I/O address space.

The controller will respond with a memory size request of 0MB (0xFFFFFFFF) when BAR0 is written with all ones. The controller contains its own dedicated memory and does not require memory allocated from the system.

Parameters

CFG_BUS defaults to zero
CFG_DEVICE defaults to six
CFG_FUNC defaults to zero
Config parameters must be set correctly. CFG device and vendors default to zero.

Registers

The PIC contains 40 registers spread out through a 256 byte I/O region. All registers are 32-bit and only 32-bit accessible. There are two different means to control interrupt sources. One is a set of registers that works with bit masks enabling control of multiple interrupt sources at the same time using single I/O accesses. The other is a set of control registers, one for each interrupt source, allowing control of interrupts on a source-by-source basis.

Regno	Access	Moniker	Purpose
00	R	CAUSE	interrupt cause code for currently interrupting source
04	RW	RE	request enable, a 1 bit indicates interrupt requesting is enabled for that interrupt, a 0 bit indicates the interrupt request is disabled.
08	W	ID	Disables interrupt identified by low order five data bits.
0C	W	IE	enables interrupt identified by low order five data bits
10			reserved
14	W	RSTE	resets the edge-sense circuit for edge sensitive interrupts, 1 bit for each interrupt source. This register has no effect on level sensitive sources. This register automatically resets to zero.
18	W	TRIG	software trigger of the interrupt specified by the low order five data bits.
20	W	ESL	The low bit for edge sensitivity selection. ESL and ESH combine to form a two bit select of the edge sensitivity.
24	W	ESH	The high bit for edge sensitivity selection
80	RW	CTRL0	control register for interrupt #0
84	RW	CTRL1	control register for interrupt #1
…		…	
FC	RW	CTRL31	control register for interrupt #31

ESH,EHL	Sensitivity
00	level sensitive interrupt
01	positive edge sensitive
10	negative edge sensitive
11	either edge sensitive

Control Register

All the control registers are identical for all interrupt sources, so only the first control register is described here.

Bits		
0 to 7	CAUSE	The cause code associated with the interrupt; this register is copied to the cause register when the interrupt is selected.
8 to 10	IRQ	This register determines which signal lines of the cpu are activated for the interrupt. Signal lines are typically used to resolve priority.
16	IE	This is the interrupt enable bit, 1 enables the interrupt, 0 disables it. This is the same bit reflected in the RE register.
17	ES	This bit controls edge sensitivity for the interrupt 0 = level, 1 = pos. edge sensitive. This same bit is present in the ESL register.
18		reserved
19	IRQAR	Respond to an IRQ Ack cycle
20 to 23		reserved
24 to 29	CORE	Core number to select for interrupt processing
30 to 31		reserved

PIT – Programmable Interval Timer

Overview

Many systems have at least one timer. The timing device may be built into the cpu, but it is frequently a separate component on its own. The programmable interval timer has many potential uses in the system. It can perform several different timing operations including pulse and waveform generation, along with measurements. While it is possible to manage timing events strictly through software it is quite challenging to perform in that manner. A hardware timer comes into play for the difficult to manage timing events. A hardware timer can supply precise timing. In the test system there are two groups of four timers. Timers are often grouped together in a single component. The PIT is a 64-bit peripheral. The PIT while powerful turns out to be one of the simpler peripherals in the system.

System Usage

One programmable timer component, which may include up 32 timers, is used to generate the system time slice interrupt and timing controls for system garbage collection. The second timer component is used to aid the paged memory management unit. There are free timing channels on the second timer component.

Each PIT is given a 64kB-byte memory range to respond to for I/O access. As is typical for I/O devices part of the address range is not decoded to conserve hardware.

PIT#1 is located at $FFFFFFFFFFEE4xxxx

PIT#2 is located at $FFFFFFFFFFEE5xxxx

Config Space

A 256-byte config space is supported. Most of the config space is unused. The only configuration is for the I/O address of the register set and the interrupt line used.

Regno	Width	R/W	Moniker	Description		
000	32	RO	REG_ID	Vendor and device ID		
004	32	R/W				
008	32	RO				
00C	32	R/W				
010	32	R/W	REG_BAR0	Base Address Register		
014	32	R/W	REG_BAR1	Base Address Register		
018	32	R/W	REG_BAR2	Base Address Register		
01C	32	R/W	REG_BAR3	Base Address Register		
020	32	R/W	REG_BAR4	Base Address Register		
024	32	R/W	REG_BAR5	Base Address Register		
028	32	R/W				
02C	32	RO		Subsystem ID		
030	32	R/W		Expansion ROM address		
034	32	RO				
038	32	R/W		Reserved		
03C	32	R/W		Interrupt		
040 to	32	R/W		Capabilities area		

0FF						

REG_BAR0 defaults to $FEE40001 which is used to specify the address of the controller's registers in the I/O address space. Note for additional groups of timers the REG_BAR0 must be changed to point to a different I/O address range. Note the core uses only bits determined by the address mask in the address range comparison. It is assumed that the I/O address select input, cs_io, will have bits 24 and above in its decode and that a 64kB page is required for the device, matching the MMU page size.

The controller will respond with a mask of 0x00FF0000 when BAR0 is written with all ones.

Parameters

CFG_BUS defaults to zero
CFG_DEVICE defaults to four
CFG_FUNC defaults to zero
CFG_ADDR_MASK defaults to 0x00FF0000
CFG_IRQ_LINE defaults to 29
Config parameters must be set correctly. CFG device and vendors default to zero.

Parameters

NTIMER: This parameter controls the number of timers present. The default is eight. The maximum is 32.

BITS: This parameter controls the number of bits in the counters. The default is 48 bits. The maximum is 64.

PIT_ADDR: This parameter sets the I/O address that the PIT responds to. The default is $FEE40001.

PIT_ADDR_ALLOC: This parameter determines which bits of the address are significant during decoding. The default is $00FF0000 for an allocation of 64kB. To compute the address range allocation required, 'or' the value from the register with $FF000000, complement it then add 1.

Registers

The PIT has 134 registers addressed as 64-bit I/O cells. It occupies 2048 consecutive I/O locations. All registers are read-write except for the current counts which are read-only. All registers all 64-bit accessible; all 64 bits must be read or written. Values written to registers do not take effect until the synchronization register is written.

Note the core may be configured to implement fewer timers in which case timers that are not implemented will read as zero and ignore writes. The core may also be configured to support fewer bits per count register in which case the unimplemented bits will read as zero and ignore writes.

Regno	Access	Moniker	Purpose
00	R	CC0	Current Count
08	RW	MC0	Max count
10	RW	OT0	On Time
18	RW	CTRL0	Control
20 to 7F8	…	…	Groups of four registers for timer #1 to #63
800	RW	USTAT	Underflow status
808	RZW	SYNC	Synchronization register
810	RW	IE	Interrupt enable
818	RW	TMP	Temporary register
820	RO	OSTAT	Output status
828	RW	GATE	Gate register
830	RZW	GATEON	Gate on register
838	RZW	GATEOFF	Gate off register

Control Register

This register contains bits controlling the overall operation of the timer.

Bit		Purpose
0	LD	setting this bit will load max count into current count, this bit automatically resets to zero.
1	CE	count enable, if 1 counting will be enabled, if 0 counting is disabled and the current count register holds its value. On counter underflow this bit will be reset to zero causing the count to halt unless auto-reload is set.
2	AR	auto-reload, if 1 the max count will automatically be reloaded into the current count register when it underflows.
3	XC	external clock, if 1 the counter is clocked by an external clock source. The external clock source must be of lower frequency than the clock supplied to the PIT. The PIT contains edge detectors on the external clock source and counting occurs on the detection of a positive edge on the clock source. This bit is forced to 0 for timers 4 to 31.
4	GE	gating enable, if 1 an external gate signal will also be required to be active high for the counter to count, otherwise if 0 the external gate is ignored. Gating the counter using the external gate may allow pulse-width measurement. This bit is forced to 0 for timers 4 to 31.
5 to 63	~	not used, reserved

Current Count

This register reflects the current count value for the timer. The value in this register will change by counting downwards whenever a count signal is active. The current count may be automatically reloaded at underflow if the auto reload bit (bit #2) of the control byte is set. The current count may also be force loaded to the max count by setting the load bit (bit #0) of the counter control byte.

Max Count

This register holds onto the maximum count for the timer. It is loaded by software and otherwise does not change. When the counter underflows the current count may be automatically reloaded from the max count register.

On Time

The on-time register determines the output pulse width of the timer. The timer output is low until the on-time value is reached, at which point the timer output switches high. The timer output remains high until the counter reaches zero at which point the timer output is reset back to zero. So, the on time reflects the length of time the timer output is high. The timer output is low for max count minus the on-time clock cycles.

Underflow Status

The underflow status register contains a record of which timers underflowed.

Writing the underflow register clears the underflows and disable further interrupts where bits are set in the incoming data. Interrupt processing should read the underflow register to determine which timers underflowed, then write back the value to the underflow register.

Synchronization Register

The synchronization register allows all the timers to be updated simultaneously. Values written to timer registers do not take effect until the synchronization register is written. The synchronization register must be written with a '1' bit in the bit position corresponding to the timer to update. For instance, writing all one's to the sync register will cause all timers to be updated. The synchronization register is write-only and reads as zero.

Interrupt Enable Register

Each bit of the interrupt enable register enables the interrupt for the corresponding timer. Interrupts must also be globally enabled by the interrupt enable bit in the config space for interrupts to occur. A '1' bit enables the interrupt, a '0' bit value disables it.

Temporary Register

This is merely a register that may be used to hold values temporarily.

Output Status

The output status register reflects the current status of the timers output (high or low). This register is read-only.

Gate Register

The internal gate register is used to temporarily halt or resume counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn on all timers where there is a '1' bit in the value and turn off all timers where there is a '0' bit in the value.

Gate On Register

The internal gate 'on' register is used to resume counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn on all timers where there is a '1' bit in the value. Where there is a '0' in the value the timer will not be affected. This register reads as zero.

Gate Off Register

The internal gate 'off' register is used to halt counting for the timer corresponding to the bit position of this register. Writing a value to this register will turn off all timers where there is a '1' bit in the value. Where there is a '0' in the value the timer will not be affected. This register reads as zero.

Programming

The PIT is a memory mapped i/o device. The PIT is programmed using 64-bit load and store instructions (LDO and STO). Byte loads and stores (LDB, STB) may be used for control register access. It must reside in the non-cached address space of the system.

Interrupts

The core is configured use interrupt signal #29 by default. This may be changed with the CFG_IRQ_LINE parameter. Interrupts may be globally disabled by writing the interrupt disable

bit in the config space with a '1'. Individual interrupts may be enabled or disabled by the setting of the interrupt enable register in the I/O space.

Testing and Debugging

This section seems short for the amount of testing I do. 90% of the work is in the testing. But this is a book about implementing or developing a processor, not a book about testing. Whole books could easily be written about testing. The key to avoiding backtracking and wasted time down the road is lots of testing along the way. Every bug fix is a test. When one bug is fixed, the next one shows up. Sometimes they seem like a two-headed hydra. Good testing skills are a requirement for developing and debugging a processor. Once you've managed to get such a thing working you're probably an ace at testing. Sometimes the processor and programming cannot help you to find a bug in the processor itself. You must be able to think in terms of 'what test can I do ?' to fix the bug. There are usually a least several wow-zzy bugs. For example, I had a bug where a register exchange instruction only failed on a cache miss, when the instruction was at the end of a cache line. Many programs worked fine, and the processor seemed not to work intermittently. It took quite a while to find. I finally noticed the instruction failed when the cache was turned off. So, one thing to try for testing is turning the cache on or off.

Test Benches

If you're going to build it there must be some way to perform testing. I'd recommend writing a test-bench first and trying the code in a simulator before trying out the code in an FPGA. A test bench is an artificial environment setup specifically to test a component. Inputs simulating a real environment are sent to the component then the output of the component is monitored for correctness. In the test bench usually so-called corner cases are tested, which are cases testing the extremes to which the component should work. If the component works in the extremes of the test bench it'll certainly work when it's put to real use is the general idea. A simulator is a tool built specifically for running test benches. The simulator has features to aid in debugging logic. One may set breakpoints, points which force the logic to stop at a particular place, and view the outputs of a component.

A simple test bench for the Thor divider circuit is shown below. Note that most test bench files don't have any input or output ports. Instead, signals are selected in the simulator for viewing.

In this case parameters for the divider were manually altered in the test bench to check for specific cases.

```
module Thor_divider_tb();
parameter WID=64;
reg rst;
reg clk;
reg ld;
wire done;
wire [WID-1:0] qo,ro;

initial begin
        clk = 1;
        rst = 0;
        #100 rst = 1;
        #100 rst = 0;
        #100 ld = 1;
        #150 ld = 0;
end

always #10 clk = ~clk;   //  50 MHz


Thor_divider #(WID) u1
(
        .rst(rst),
        .clk(clk),
        .ld(ld),
        .sgn(1'b1),
        .isDivi(1'b0),
        .a(64'd10005),
        .b(64'd27),
        .imm(64'd123),
        .qo(qo),
        .ro(ro),
        .dvByZr(),
        .done(done)
);

endmodule
```

Note that it is possible to automate test cases and even use file I/O in some tools. Test benches can become quite complex. Test benches for the float components often use a test input file containing the operands for the design under test, DUT, and output the results along with the input operands in a results output file. The output file can then be studied at leisure for issues to correct. Having a file output allows different revisions of the core to be compared and may make regression testing easier.

It is extremely unlikely that one would get the HDL code perfect the first time. The processor is not likely to be working, so how do you fix it up ? One needs debugging dumps of course, and those are only available from a simulator. Judiciously placed debug output can be real aid to

getting the cpu working. Unless a fix-up is minor and well-known, I run simulator traces before attempting to run the code in an FPGA.

As a first test running software code in the FPGA try something simple like turning an LED on or off. One of the first lines of code Table888 executes is:

```
start
       sei                                    ; disable interrupts
       ld            r1,#$FF
       st            r1,LEDS
```

which turns on all the LEDs on the board.

This idea is popular for debugging hardware. The IBM PC had a "post-code" which was a byte value periodically written to an I/O port during startup for debugging. Depending on the display of the byte one could tell where in start-up it failed. Something like a missing or bad display adapter would end up with a specific code.

Another suggestion for test-benches is to use the actual system being loaded into the FPGA device as a component of the test-bench. If one keeps the system simple enough to start with then it's possible to debug using the test-bench.

Emulators

An invaluable tool for debugging software prior to the processor being finished is the software emulator. A software emulator is an emulation of the device or system written as a software program to run on a workstation. Software emulators are often significantly slower than the real hardware. It's also a tool where events applied to the system can be generated by user input. The code for the software emulation of a system mirrors the code for processor implementation itself. The code is just written in a different language. Having an emulator available allows for consistency checks between the emulation and the "real" device. Ideally the emulator should produce the same results as the real device would, except that it's in a virtual environment of the emulator. The emulator can help resolve software problems that would be too difficult to do using the logic simulator.

Emulators can be cycle-exact, meaning they emulate what happens during each cycle of the processor's clock. Cycle-exact emulators are often slower than non-cycle exact ones. An emulator that is not cycle exact may only emulate running software, interpreting object code, rather than performing all the internal operations that the CPU does.

Bootstrap Code vs the "Real Code"

The next thing to do after getting simpler I/O tests working is more complex I/O like a video display. Being able to display things on-screen can be invaluable (a character LCD display or LED display works well too). Many low-cost FPGA boards come with a numeric LED displays for output and buttons for input. It's slightly more challenging to drive a numeric display and may make a good second test. Also being able to get a keystroke can be valuable too. One of the first routines my processors execute is the clear-screen routine. If it can't clear the screen I know something's seriously wrong in the start-up. While the blue screen-of-death may be a bad sign, it's a good sign at least the processor is working that much. When setting the processor software up (bootstrapping) don't go for the most complex algorithms to begin with. Go with simple things. I have two versions of keyboard routines. The one that 'works the right way' and the one I use for bootstrapping. The bootstrapping routine goes directly to the keyboard port to read a character. It's very simple, and pauses the whole machine waiting for a character.

Data Alignment

Are your variables mysteriously getting over-written ? There could be a problem with address generation in the processor, or perhaps a problem with the external address decoding.

One approach to aligning data structures in memory is to ensure that the structures don't have partially overlapping addresses. This may help if there are memory addressing problems. For instance, if data structure addresses all end in xxx000, then if there is an address decoding problem, all the structures may get overwritten by values intended for other variables. If the variable addresses are somewhat mangled for example 0xxxx004,xx1018, xx2036 (ending in different LSB's) then it may be less likely for data to be corrupted. This is a temporary debugging approach. One would want to have the var's properly listed in a program.

Get Rid of Complexity

One of the best ways to be able to debug something is to get rid of all the extra complexities involved with it. Many is the time that the author has backtracked on a project and removed features in favor of getting something to work. Add one feature at a time, make it a component that can be easily disabled or removed from the design. Disable the complex features of the design. It's great to be able to do a complex design. But all the complicated stuff started out small and simple. One doesn't need caches, interrupts, branch predictors, and so on to have a working design. It's very rewarding to have even the simplest design working.

Disabling Interrupts

This bit only applies if you've managed to get some sort of interrupt facility working. Several smaller, simpler systems don't make use of interrupts. The original Apple computer did not use interrupts. Interrupts aren't something that one must get working right away. They would be part of a longer-term project goal (if at all). Start small and simple and expand from there. There are alternatives to interrupts the main one being polling in a loop.

When working with the real hardware having a set of switches available can be invaluable. The switches can be wired to key signals in the design to offer a manual override option. There may be times when one desires to disable a feature under development while other aspects of the project are taking place. For instance, eventually at some point in time one might want to venture into the world of interrupt processing. Interrupts are a challenge to get working. It's nice to be able to disable interrupts using an external switch. Also, there are times when one wants to know if the processor is capable of executing a linear sequence of instructions, without the interference of interrupts. Debugging the processor with interrupts enabled can be tricky. Development of an interrupt system is something for a later stage of development. Get the processor running longer sequences of code successfully first before trying to deal with interrupts.

IRQ Live Indicator

The IRQ live indicator is one of the first debug techniques the author uses once the core can run some code. An indicator that IRQ's are happening seems like a friendly image. It can be useful to see that IRQ's are happening on a regular basis. An IRQ indicator can let one know if the machine is just busy, or really, really stuck. This can be accomplished by incrementing a character at a fixed location on-screen. If that character stops flipping around one knows there's real trouble. Another common approach is to use an LED to indicate the presence of IRQ's. Turning a LED on and off at a low frequency can be handy to visually detect the presence of IRQs.

Disable Caching

This tip applies only if a cache is present. Implementing a cache isn't priority number one. The first few projects I did, did not include any caching. It was too complex to add a cache to begin with. As mentioned before, it sometimes necessary to disable the cache. Nice-to-have instructions are a cache-on and cache-off instruction. The processor should end up with the same results regardless of whether caching is enabled. If results seem flaky try disabling the cache.

Clock Frequency

Be conservative when choosing a clock frequency. Don't try to run at the fastest possible frequency until the design is thoroughly debugged. Sometimes changing the clock frequency will provide clues to timing or synchronization problems. If the problem varies with a change in clock frequency, then maybe it's a timing problem. If the problem is consistent regardless of the clock frequency, it's likely some other problem. Note we are dealing with debugging probabilities here. Just because a problem is consistent at different clock frequencies doesn't mean it's not a timing problem.

Another nice aspect of a conservative clock frequency is that the tools used for building the system often work much faster if it's easy for the tools to meet the timing requirements. A conservative clock frequency is a way to speed up the development cycle.

More Advanced Debugging Options

The following debugging mechanisms fall under the category of being more sophisticated in nature and more difficult to do, but they can sometime prove invaluable. They require interrupts or exceptions.

Debug Registers

One option that aids primarily software debugging is the presence and use of debug registers. Adding debug registers to the core may make software debugging easier to do. Typically, there are one or more address matching registers that cause an interrupt or exception when the processor's program counter or data address matches the one in the debug register. One must have a working interrupt system for this to be usable.

Trace / Program Counter History

One of the debug facilities that I've added to cores is the capability to capture the history of the program counter. While the processor is running at full speed, the program counter is stored in a small history table which is usually some sort of shift register. When an exceptional condition occurs in the processor core the history capture is turned off. In the exception processing routine, the program counter history can then be dumped to the screen showing where the program went awry.

The technique is called "trace". A good trace history will often be able to be triggered perhaps at a specific address or via debug match register. The trace may record all instructions, but it is

common to record only the branch history, and then a few of the instruction addresses for synchronization purposes. Since branches are either taken or not taken a single bit can be used to record the history making trace very compact. With only a couple of block RAMs a trace history of thousands of instructions is possible.

Stuck on a Bug ?

This is a brain trick. Try changing the code around in the area of the bug. Sometimes just by changing the code, refactoring without really changing operation, you will be able to spot a bug that wasn't readily apparent. It's a bit like moving your eyes around on the horizon to try and spot an enemy. The action of changing or simply moving the code causes a bug to pop out, out of the shadows.

The Rare Chance

There is a rare chance that it's a problem in the toolset. A problem like this can make things really difficult, especially if it's a free toolset with no technical support. In about 20 years or so, of using toolsets I've found a few bugs. The toolsets, generally speaking are superb, so the chance of it being a bug in a toolset is extremely remote but not impossible. The one bug I ran into was in extending a complement of a single bit value. The toolset returned a binary "10" the value two when a single bit was being inverted. It should have returned a zero. I was able to work around this problem by zero extending the value manually. I found the bug by tracking the location of it down and dumping values using debug outputs.

Bugs in toolsets are often obvious. The most recent one caused the toolset to crash and quit running depending on how simulation was started. There was a work-around by restarting the simulation fresh every time which takes longer than the usual restart.

If you suspect a bug in the toolset try searching the web for information on it. If it's a common problem it's bound to be posted on the web somewhere. There are also usually forums on the web where one can post about problems, and even sometimes get replies.