[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

Nomenclature

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	
24	char	LDC, STC	UTF24 usage
32	tetra	LDT, STT	
40	penta	LDP, STP	Instruction size
64	octa	LDO, STO	
96		LDN, STN	

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

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$_5$

Programming Model

Register File

Rn – General Purpose Registers

The register file contains 64 96-bit general purpose registers. The register file is *unified*; register may hold integer or floating-point values. The stack pointer is banked with a separate stack pointer for each operation mode.

Regno	ABI	ABI Usage	
0	0	Always zero	
1	A0	First argument / return value register	
2	A1	Second argument / return value register	
3	T0	Temporary register, caller save	
4	T1	Temporary register	
5	T2	Temporary register	
6	T3	Temporary register	
7	T4	Temporary register	
8	T5	Temporary register	
9	T6	Temporary register	
10	T7	Temporary register	
11	T8	Temporary register	
12	T9	Temporary register	
13	S0	Saved register, register variables	
14	S1	Saved register	
15	S2	Saved register	
16	S3	Saved register	
17	S4	Saved register	
18	S5	Saved register	
19	S6	Saved register	
20	S7	Saved register	
21	S8	Saved register	
22	S9	Saved register	
23	A2	Third argument register	
24	A3	Argument register	
25	A4	Argument register	
26	A5	Argument register	
27	A6	Argument register	
28	A7	Argument register	
29	A8	Argument register	
30	A9	Argument register	
31			

32	M0	Vector mask	
33	M1	Vector mask	
34	M2	Vector mask	
35	M3	Vector mask	
36	M4	Vector mask	
37	M5	Vector mask	
38	M6	Vector mask	
39	M7	Vector mask	
40			
41			
42			
43			
44			
45			
46			
47			
48			
49			
50			
51			
52			
53	PC	Program counter / Status Register	
54	SC	Stack canary; a LOAD does CCHK	
55	LC	Loop counter	
56	LR0	Subroutine link register #0; branch subroutine specific	
57	LR1	Subroutine link register #1	
58	LR2	Subroutine link register #2	
59	LR3	Subroutine link register #3	
60	GP1	Global Pointer #1	
61	GP0	Global Pointer #0	
62	FP	Frame Pointer	
63	SP	Stack Pointer	
63	ASP	Application Stack pointer	
63	SSP	System Stack pointer	

	AC	Application Control Register	

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 (m0 to m7)

Mask registers are used to mask off vector operations so that a vector instruction doesn't perform the operation on all elements of the vector. Vector instructions (loads and stores) that don't explicitly specify a mask register assume the use of mask register zero (m0). Mask registers are a subset of the general-purpose register array, allowing instructions that operate on GPRs to operate on the mask registers.

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

Register	Tag	Usage
m0	88	contains all ones by convention
m1	89	
m2	90	
m3	91	
m4	92	
m5	93	
m6	94	
m7	95	

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.

15 8	7 0
0	$\text{Elements}_{7..0}$

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.

LRn – Link Registers

There are four registers in the Thor2023 architecture reserved for subroutine linkage. These registers are used to store the address of 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 may reload the program counter with an offset from the value contained in a link register. Typically, this 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.

LC - Loop Counter

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

SR - Status Register

The processor status register holds bits controlling the overall operation of the processor. The status register is not accessible in user mode.

31 24	23 21	20 16	15		13 12	11	10 8	7	6	5	4	3	2	1	0
CPL	~	~	T		OM		IPL			D	D	AR		RT	

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

T indicates that trace mode is active.

S indicates the processor is in supervisor mode.

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.

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.

Special Purpose Registers

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

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

63 4	3	2	1	0
~	mie	hie	sie	uie

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

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

[U/S/H/M]_SCRATCH – CSR 0x?041

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

S_PTBR (CSR 0x1003)

This register contains the base address of the page table, which must be a multiple of 65536. Also included in this register is table parameters depth and type. Register tag #152.

63 12	11	10 8	7 6	5	4	3 1	0
Page Table Address$_{67..16}$	~	Levels	AL_2	~	S	~	Type

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.

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 eight bits of the register are implemented.

S_KEYS (CSR 0x1020 to 0x1027)

These eight registers contain the collection of keys associated with the process for the memory lot system. Each key is twenty-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.

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.

63 60	59 16	15 0
$State_4$	$\sim_{44}$	$seed_{16}$

$State_4$ Bit	
0	dead
1	test
2	valid, the seed value is valid
3	Busy, the collector is busy collecting a new seed value

M_BADADDR (CSR 0x3007)

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

M_BAD_INSTR (CSR 0x300B)

This register contains a copy of the exceptioned instruction.

M_RGN0* to M_RGN7* (CSR 0x3080 to CSR 0x30FF)

This group of registers contains attributes of the corresponding region.

M_RGNx_START (CSR 0x3080 + x * 16)

This register contains the start address for region number x.

M_RGNx_END (CSR 0x3081 + x * 16)

This register contains the end address for region number x.

M_RGNx_PMT (CSR 0x3082 + x * 16)

This register contains the address of the page management table, PMT, for region number x.

M_RGNx_CTA (CSR 0x3083 + x * 16)

This register contains the address of the card table for region number x.

M_RGNx_ATTR (CSR 0x3084 + x * 16)

This register contains the memory attributes of region number x.

M_RGN_START (CSR 0x3088)

This read-only register contains the start address of the region corresponding to the physical address set in the MN_RGN_PHYS CSR.

M_RGN_END (CSR 0x3088)

This read-only register contains the end address of the region corresponding to the physical address set in the MN_RGN_PHYS CSR.

M_RGN_PMT (CSR 0x3089)

This read-only register contains the address of the PMT for the region corresponding to the physical address set in the MN_RGN_PHYS CSR.

M_RGN_CTA (CSR 0x308A)

This read-only register contains the address of the card table for the region corresponding to the physical address set in the MN_RGN_PHYS CSR.

M_RGN_ATTR (CSR 0x308B)

This read-only register contains the attributes for the region corresponding to the physical address set in the MN_RGN_PHYS CSR.

M_RGN_PHYS (CSR 0x308E)

Set this register to the physical address to lookup the region information. The region number corresponding to the physical address will be updated in the M_RGN_NUM register.

M_RGN_NUM (CSR 0x308F)

This register is set to the region number for the physical address set by the M_RGN_PHYS CSR.

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}

SC - Stack Canary

This special purpose register is available in the general register file as register 54. 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.

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

Tags

<table>
<tr><td>Tag</td><td></td></tr>
<tr><td>0</td><td>Untagged</td></tr>
<tr><td>1</td><td>Address Pointer – 20 bit size + 64 bit pointer
<table>
<tr><td>Subtype</td><td></td></tr>
<tr><td>0</td><td>Unused</td></tr>
<tr><td>1</td><td>Return address</td></tr>
<tr><td>2</td><td>Frame Pointer</td></tr>
<tr><td>3</td><td>Pointer</td></tr>
<tr><td>4 to 7</td><td>Unassigned</td></tr>
<tr><td></td><td></td></tr>
</table>
</td></tr>
<tr><td>2</td><td>Integer 96 bits</td></tr>
<tr><td>3</td><td>Integer 64 - bits</td></tr>
<tr><td>4</td><td>Integer 32 - bits</td></tr>
</table>

5	Integer 16 - bits
6	Integer 8 - bits
8	Float 96 bits
9	Float 64 bits
10	Float 32 bits
11	Float 16-bits
12	Float 8-bits
16	String Descriptor – 24 bit length, 64 bit virtual address pointer
17	Character data, three 32-bit characters
18	Character data, four 24-bit characters
19	Character data, 12 8-bit characters
63	Instructions 40-bit parcels

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, program counter, and predicate group register 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
65 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.

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. This table includes the address range the attributes apply to and the attributes themselves. Address ranges are resolved only to bit four of the address. Meaning the granularity of the check is 16 bytes.

Most of the entries in the table are hard-coded and configured when the system is built. 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 set of CSRs using the CSR instructions.

Region Table Description

CSR	Bits		
0x3080	64	start	start address bits 4 to 67 of the physical address range
0x3081	64	nd	end address bits 4 to 67 of the physical address range
0x3082	64	pmt	associated PMT address
0x3083	64	cta	card table address
0x3084	19	at	memory attributes
0x3085 to 0x308F	…	…	not used
0x3090 to 0x30FF	…	…	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

<table>
<tr><th>Bitno</th><th></th><th></th></tr>
<tr><td>0</td><td>X</td><td>may contain executable code</td></tr>
<tr><td>1</td><td>W</td><td>may be written to</td></tr>
<tr><td>2</td><td>R</td><td>may be read</td></tr>
<tr><td>3</td><td>C</td><td>may be cached</td></tr>
<tr><td>4-6</td><td>G</td><td>granularity
<table><tr><th>G</th><th></th></tr><tr><td>0</td><td>byte accessible</td></tr><tr><td>1</td><td>wyde accessible</td></tr></table></td></tr>
</table>

<table>
<tr><td></td><td></td><td><table><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>7</td><td>~</td><td>reserved</td></tr>
<tr><td>8-15</td><td>T</td><td>device type (rom, dram, eeprom, I/O, etc)</td></tr>
<tr><td>16-18</td><td>S</td><td>number of times to shift address to right and store for telescopic STPTR stores.</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.

The PMT is implemented as a dual-read-write port RAM that allows hardware to update it at high speed during a memory access. The current PMT can provide information for 16384 pages. These pages may be DRAM, ROM, MMIO or other types.

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

<table>
<tr><td>V</td><td>N</td><td>M</td><td colspan="2">~$_{9}$</td><td>C</td><td>E</td><td>AL$_{2}$</td><td>~$_{16}$</td></tr>
<tr><td colspan="8">ACL$_{16}$</td><td>Share Count$_{16}$</td></tr>
<tr><td colspan="9">Access Count$_{32}$</td></tr>
<tr><td colspan="4">PL$_{8}$</td><td colspan="5">Key$_{24}$</td></tr>
</table>

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.

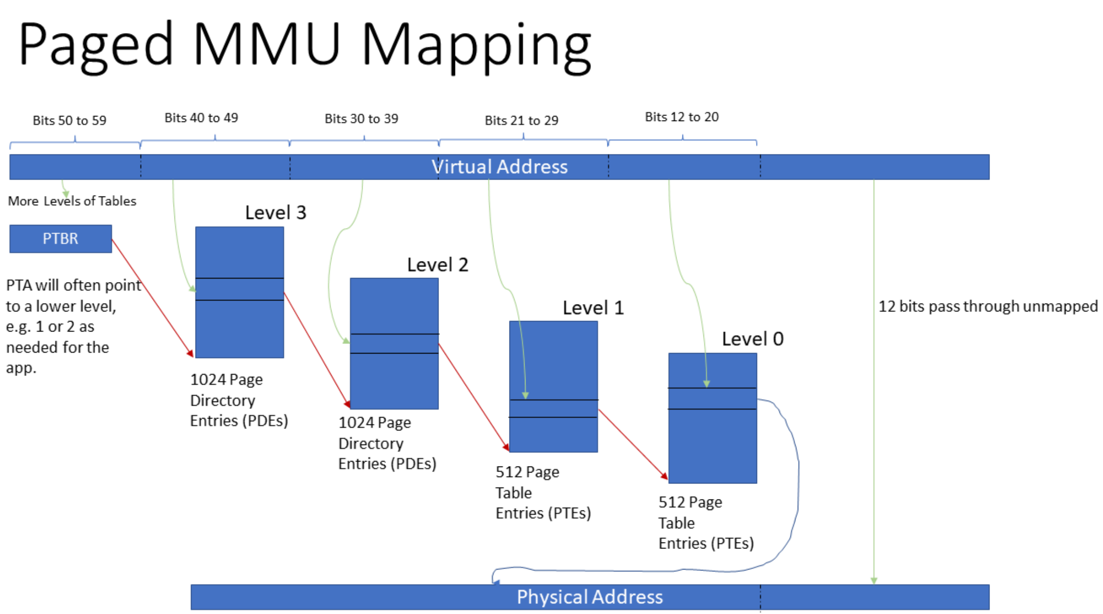
Page Tables

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. 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.

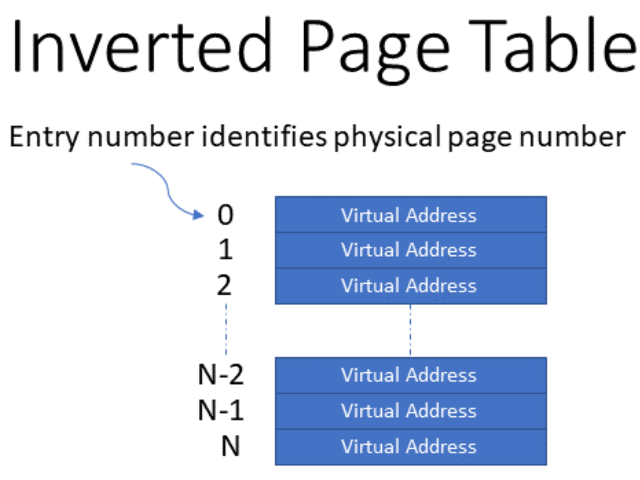


Inverted Page Tables

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 is 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, only addresses that correspond to real pages of memory. 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.

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. 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 128-bit HPTE format.

<table>
<tr><td>31</td><td>30 26</td><td>25</td><td>24</td><td>23 21</td><td>20</td><td>19 15 0</td></tr>
<tr><td>V</td><td>BC_5</td><td>G</td><td>~</td><td>RWX_3</td><td>A</td><td>$PPN_{19..0}$</td></tr>
<tr><td colspan="5">$ASID_{15..0}$</td><td colspan="2">$VPN_{15..0}$</td></tr>
<tr><td colspan="7">$PPN_{51..20}$</td></tr>
<tr><td colspan="7">$VPN_{47..16}$</td></tr>
</table>

Fields Description

V	translation Valid
G	global translation
PPN	Physical page number
VPN	Virtual page number
RWX	readable, writeable, executable
ASID	address space identifier
BC	bounce count
M	modified
A	accessed

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.

The virtual to physical address mapping is for an 64kB page.

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 eight page-table 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 page table is small enough to fit into the block RAM resources available in the FPGA. So, 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 / 32 = 16384 entries. 16384 / 4 = 4096 physical pages. Since the RAM size is 512MB, each page would be 512MB/4096 = 128kB. Since half the pages may be in secondary storage, 1GB of address range is available.

Since there are 16,384 entries in the table and they are grouped into groups of eight, there are 2048 PTGs. To get to a page table group fast a hash function is needed then that returns a 11-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 eight 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.

Location of Page Table

Thor2023's hash page table is in the physical address space at $FFAxxxxx. It is a specially dedicated block RAM memory which has two sides. One side is updateable and readable via the load hexi-byte pair and store hexi-byte pair LDHP, STHP instructions. The other side is updateable and readable in terms of page groups by the hash page table control logic.

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 or the ASID. We know the virtual address because it is what is being translated and there is no chance of collisions unlike the hash table. Since there is a separate page table for each process the ASID does not need to be stored in it. The structure is 64 bits in size. This allows 8192 PTEs to fit into an 64kB page.

There are three types of page table entries. The first type, 0, is a pointer to a 64kB page of memory, the second type, 1, is a pointer to a 256B page of I/O, and the third type, 2, is an entry that points to lower-level page tables. The type is determined by the 'Ty' bits in the PTE. PTE's that point to lower-level page tables are sometimes called page table pointers, PTPs.

Page Table Entry Format – PTE

V	LVL$_3$	~	Ty	M	RWX$_3$	A	PPN$_{19..0}$
PPN$_{51..20}$							

Type 0 – memory page pointer

V	LVL$_3$	~	0	M	RWX$_3$	A	PPN$_{19..0}$
PPN$_{51..20}$							

Type 1 – I/O page pointer

V	LVL$_3$	~	1	~	RWX$_3$	~	IOPN$_{27..8}$
IOPN$_{7..0}$				IOPN$_{51..28}$			

Type 1- page table pointer

V	LVL$_3$	~	2	~	~$_3$	~	PPN$_{19..0}$
PPN$_{51..20}$							

Field	Size	Purpose
PPN	52	Physical page number
A	1	1=accessed
X	1	1=executable
W	1	1=writeable
R	1	1=readable
M	1	1=modified
V	1	1 if entry is valid, otherwise 0
T	1	0=page pointer,1 = table pointer
LVL	3	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 a large number of pages. If the page level is greater than zero and the PTE is marked as a pointer to a memory page then it is pointing to a super page. 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:

1	1$_3$	~$_2$	0	M	5$_3$	A	0x1FF$_9$	0x00f$_{11}$
0xFFFFFFFF$_{32}$								

512MB pages

V	1$_3$	~$_2$	0	M	RWX$_3$	A	PPN$_{19..11}$	PC$_{11}$
PPN$_{51..20}$								

64GB pages

V	2_3	$\sim_2$	0	M	RWX_3	A	$PC_{19..0}$	
$PPN_{51..22}$								$PC_{21..20}$

MMU Cache

To improve the performance of PDE lookups. The MMU has a small fully associative cache for PDE lookups.

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 five 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 fifth 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 five-way set associative cache. The fifth way may only be updated by software. The fifth way allows translations to be stored that will not be overwritten.

TLB Entries - TLBE

Closely related to page table entries are translation look-aside buffer, TLB, entries. TLB entries have more fields to provide access counting and keyed access. The additional field are populated from the page management table, PMT.

<table>
<tr><td>V</td><td>LVL$_3$</td><td>~$_2$</td><td>T</td><td>M</td><td>RWX$_3$</td><td>A</td><td colspan="5">PPN$_{19..0}$</td></tr>
<tr><td colspan="12">PPN$_{51..20}$</td></tr>
<tr><td colspan="8">AS$_{15..0}$</td><td>Virtual Page Number$_{17..10}$</td><td>~$_2$</td><td>G</td><td>BC$_5$</td></tr>
<tr><td colspan="12">VPN$_{49..18}$</td></tr>
</table>

<table>
<tr><td>V</td><td>N</td><td>M</td><td colspan="2">~$_9$</td><td>C</td><td>E</td><td>AL$_2$</td><td>~$_{16}$</td></tr>
<tr><td colspan="8">ACL$_{16}$</td><td>Share Count$_{16}$</td></tr>
<tr><td colspan="9">Access Count$_{32}$</td></tr>
<tr><td colspan="4">PL$_8$</td><td colspan="5">Key$_{24}$</td></tr>
</table>

<table>
<tr><td>PTE Address$_{31..0}$</td></tr>
<tr><td></td></tr>
</table>

The TLB entry also contains pointers to the PTE and PMT entries used to update the TLB. The TLB needs this information to be able to update those structures in memory.

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 64kB. For a 512MB system (the size of the memory in the test system) there are 8192 64kB 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 seven buckets which must be filled with TLB info using STO instructions. Then address $F…FE00038 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 or LRU way algorithm. Otherwise, a way to update will be selected randomly. The data is octa-byte aligned. When the LRU algorithm is active the most recently used entry is placed in way #0. It may be desirable to bump out entry #3 and replace it with the new entry for LRU operation.

Bucket	63 32	31 16	15	14 5	4 3	2 0
00	$PTE_{63..0}$					
08	$PTE_{127..64}$					
10	$PMT_{63..0}$					
18	$PMT_{127..64}$					
20		PTE Address				
28		PMT Address				
30			0	Entry Num_{10}	$\sim_2$	way_3
38						

Example TLB Update Routine

```
_TLBMap:
        ldo     a0,0[sp]
        ldo     a1,8[sp]
        ldo     a2,16[sp]
        ldo     a3,24[sp]
        ldo     a4,32[sp]
        ldo     a5,40[sp]
        ldo     a6,48[sp]
        ; <lock TLB update semaphore>
        sto     a1,0xFFE00000           # PTE value
        sto     a2,0xFFE00008           # PTE value
        sto     a3,0xFFE00010           # PMT value
        sto     a4,0xFFE00018           # PMT value
        sto     a5,0xFFE00020           # PTE address
        sto     a6,0xFFE00028           # PMT address
        sto     a0,0xFFE00030           # entry number
        sto     r0,0xFFE00038           # triggers a TLB update
        ; <unlock TLB update semaphore>
        add     sp,sp,56
```

```
        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, rotates the most recent address translation to the first way and updates by over-writing the value in the third way. 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.

IO Address Resolver

The IO address resolver resolves a virtual address down to a finer resolution than memory pages for access to IO devices. IO pages are 256B in size compared to a memory page of 64kB.

The IOAR contains 4096 entries which contain the lower 32-bits of the physical address for IO devices. The low order eight bits of the address pass through the MMU unchanged. Virtual address bits 8 to 19 index a 4096-entry table used to lookup the physical address.

IOAR Entry Format – IOARE

V	RWX_3	$\sim_4$	$IOPN_{23..0}$

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.

Instruction Descriptions

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	28 Bcc	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	12	13 CHRNDX	14 CLMUL	15 SQRT
1x	16 DIF	17 PTRDIF	18 REVBIT	19 BMAP	20	21	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 VMASK	35 ATOM	36 ROUND	37	38	39
	40 V2BITS	41 BITS2V	42 VEX	43 VEINS	44 VGNDX	45	46	47
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

{SHIFT}

	0	1	2	3	4	5	6	7
0x	0 ASL	1 ASR	2 LSL	3 LSR	4 ROL	5 ROR	6	7
	8 ZXB	9 SXB	10	11	12	13	14	15
1x	16 VSHLV	17 VSHRV	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
2x	32 ASLI	33 ASRI	34 LSLI	35 LSRI	36 ROLI	37 RORI	38	39
	40 ZXBI	41 SXBI	42	43	44	45	46	47
3x	48 VSHLVI	49 VSHRVI	50	51	52	53	54	55
	56	57	58	59	60	61	62	63

{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 FSEQ	9 FSLT	10 FSLE	11 FSNE	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 FS2T	11 FD2T	12	13	14 ISNAN	15 FINITE
1x	16	17	18	19	20	21 FTRUNC	22	23 FRES
	24	25 FD2S	26 FT2S	27 FT2D	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 four different operand sizes: byte, wyde, tetra and octa. The operand size is selected by suffixing the mnemonic with ‘b’ for byte, ‘w’ for wyde, ‘t’ for tetra and ‘o’ for octa.

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 Penta
6	.n	96-bit
7		reserved

Arithmetic Operations

Representations

Int:96

95	0
96 bits	

Int:64

63	0
64 bits	

Int:40

39	0
40 bits	

Int:32

31	0
32 bits	

Int:16

15	0
16 bits	

Int:8

7	0
8 bits	

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

Integer Instruction Format: R2

ABS Rt, Ra – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
3_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

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 the decimal mode flag is set 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

Operation:

Rt = Ra + Rb or Rt = Ra + Imm

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

ADD Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	4_5

Clock Cycles: 1

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, .c, .p, .n

Clock Cycles: 1

Operation:

Rt = Ra & Rb or Rt = Ra & Imm

Instruction Formats:

AND Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
8_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	8_5

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

BMAP – Byte Map

Description:

First the target register is cleared, then bytes are mapped from the 12-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 48-bits of register Rb or a 48-bit immediate constant. Bytes which are not mapped will end up as zero in the target register.

Instruction Formats:

BMAP Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
19_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

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

39 33	32	31 23	22	21 16	15	14 9	8	7 5	4 0
$\sim_7$	S	$\sim_9$	Sa	Ra_6	St	Rt_6	V	Sz_3	19_5
Immediate$_{31..0}$								0_3	31_5
$\sim_{16}$			Imm$_{47..32}$					1_3	31_5

Clock Cycles: 1

Operation:

Vector Operation

Execution Units: First Integer ALU

Clock Cycles: 1

Exceptions: none

Notes:

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

39	38	37 32	31	30	29	28 23	22	21 16	15	14 12	11 9	8	7 5	4 0
~	Vc	Rc$_6$	0	Vb	Sb	Rb$_6$	Sa	Ra$_6$	~	~$_3$	Cn$_3$	V	Sz$_3$	11$_5$

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

39 32	31	30	29	28 23	22	21 16	15 12	11 9	8	7 5	4 0
Imm$_{11..4}$	1	Vb	Sb	Rb$_6$	Sa	Ra$_6$	Imm$_{3..0}$	Cn$_3$	V	Sz$_3$	11$_5$

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 4

CLMUL Rt,Ra,Imm$_8$

39 34	33 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
4_6	$\sim_2$	1	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 4

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, .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:

ADD Rt, Ra, Rb – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
5_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	5_5

Clock Cycles: 1

Rt bit	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	
1	LT	< less than	
2	LE	<= less than or equal	
3	LO / CS	< unsigned less than	
4	LS	<= unsigned less than or equal	
5	AND	And	
6	OR	Or	
7	T	1	
8	NE	< > not equal	
9	GE	>= greater than or equal	
10	GT	> greater than	
11	HS / CC	unsigned greater than or equal	
12	HI	unsigned greater than	
13	NAND	nand	
14	NOR	Nor	
15	SR	Branch subroutine	

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
0_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
2_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

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,#$Regno_{12}$

3938	37 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
0	$Regno_{13..8}$	S	$Regno_{7..0}$	Sa	Ra_6	St	Rt_6	V	Op_3	3_5

Clock Cycles: 1

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

3938	37 32	31	30 23	22 16	15	14 9	8	7 5	4 0
1	$Regno_{13..8}$	S	$Regno_{7..0}$	Imm_7	St	Rt_6	V	Op_3	3_5

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

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
7_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 100

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	7_5

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.

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

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
7_6	1_2	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

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

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
10_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	10_5

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
10_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	1	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	1	Rt_6	V	Sz_3	10_5

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. 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:

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:

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.

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

Instruction Format:

This format extends the previous constant value by 32 bits beginning at bit 64 and sign extends the value to the width of the machine.

39 8	7 5	4 0
Immediate_{32}	2_3	31_5

MULS – Multiply Signed

Description:

Multiply 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 ‘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 – Register direct

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
6_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 12

MULS Rt,Ra,Imm16

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	6_5

Clock Cycles: 12

Clock Cycles: 12

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

Operation:

Rt = Ra * Rb or Rt = Ra * Imm

Instruction Formats:

MULU Rt, Ra, Rb – Register direct

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
6_6	$\sim_2$	1	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 12

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	1	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	6_5

Clock Cycles: 12

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
8_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	1	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	1	Rt_6	V	Sz_3	8_5

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
9_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	1	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

NOR Rt,Ra,Imm$_{16}$

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	1	Rt_6	V	Sz_3	9_5

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

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
9_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	9_5

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
18_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

Operation:

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
15_6	$\sim_2$	0	0	0	0_6	Sa	Ra_6	St	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

Operation:

Rt = SQRT(Ra)

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

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:96

The core uses a 96-bit triple precision binary floating-point representation.

96-bit values are more compact than 128-bit ones which reduces the amount of hardware required and data being transferred. They have enough significant digits for a wide variety of applications. 64-bit values are not sufficient for some applications. The question then is how much larger of a representation to use. 80-bits is popular, offering about 19 significant digits which is good for a wide variety of applications. 96-bit floats offer about 24 significant digits.

95	94 80	79 0
S	Exponent_{15}	Significand_{80}

Double Precision, Float:64

63	62 52	51 0
S	Exponent_{11}	Significand_{52}

Single Precision, Float:32

31	30 23	22 0
S	Exponent_{8}	Significand_{23}

Half Precision, Float:16

15	14 10	9 0
S	Exponent_{5}	Significand_{10}

Decimal Floats

The core uses a 96-bit densely packed decimal triple precision floating-point representation.

95	94 90	89 80	79 0
S	$Combo_5$	$Exponent_{10}$	$Significand_{80}$

The significand stores 25 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:

63	62 58	57 50	49 0
S	$Combo_5$	$Exponent_8$	$Significand_{50}$

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

32-bit single precision decimal floating point:

31	30 26	25 20	19 0
S	$Combo_5$	$Exponent_6$	$Significand_{20}$

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		Reserved
5		Reserved
6	.t	96-bit triple
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

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_5	$\sim_3$	0	0	0	32_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

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 96-bit floating-point values. An immediate value is converted to 96-bit triple precision 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

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_5	Rm_3	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$\sim_8$	S	$\sim_8$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5
$Immediate_{32}$								0_3	31_5

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$\sim_8$	S	$\sim_8$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5
$Immediate_{31..0}$								0_3	31_5
$Immediate_{63..32}$								1_3	31_5

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$\sim_8$	S	$\sim_8$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5
$Immediate_{31..0}$								0_3	31_5
$Immediate_{63..32}$								1_3	31_5
$Immediate_{95..64}$								2_3	31_5

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.

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

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
5_5	$\sim_3$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 1

FCMP Rt,Ra,Imm_{13}

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	21_5

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			
15			

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

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
7_5	Rm_3	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 150

FDIV Rt,Ra,Imm_{16}

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	23_5

Clock Cycles: 150

FMUL –Float Multiplication

Description:

Multiply two source operands and place the product 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

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Formats:

FDIV Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
6_5	Rm_3	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 8

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

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	22_5

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.

Integer Instruction Format: R1

FNEG Rt, Ra, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_5	$\sim_3$	0	0	0	34_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

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.

Supported Operand Sizes:

Operation:

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Formats:

FSCALEB Rt, Ra, Rb – Register direct

39 36	35 33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
0_4	0_3	0	Vb	Sb	Rb_6	Va	Sa	Ra_6	Vt	St	Rt_6	Sz_3	12_5

Clock Cycles: 1

FSCALEB Rt, Ra, #Imm – Immediate

39 36	35 33	32	31 24	23	22	21 16	15	14	13 8	7 5	4 0
0_4	1_3	0	Imm_8	Va	Sa	Ra_6	Vt	St	Rt_6	Sz_3	12_5

$\sim_{16}$	$Immediate_{15..0}$	0_3	31_5

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 88-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

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_5	Rm_3	0	Vb	1	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 8

FSUB Rt,Ra,Imm_{13}

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	20_5

Clock Cycles: 8

s

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, Rb – Register direct

39 35	34 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_5	$\sim_3$	0	0	0	21_6	Sa	Ra_6	St	Rt_6	V	Sz_3	12_5

Clock Cycles: 1

Operation:

Rt = Trunc(Ra)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

String Operations

Representations

Strings

95 … 92	91 … 64	63 … 0
Typ	Length$_{28}$	Pointer$_{64}$

UTF8 Chars

95 … 0
12 characters

UTF24 Chars

95 … 0
4 characters

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 +11 for UTF8 characters or -1 to +3 for UTF24 characters. The index of the first found byte is returned (closest to zero).

Supported Operand Sizes: .b, .c

Instruction Formats:

CHRNDX Rt, Ra, Rb – Register direct

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
13_6	~	0	Vb	Sb	Rb_6	Va	Sa	Ra_6	Vt	St	Rt_6	Sz_3	2_5

Clock Cycles: 1

CHRNDX Rt,Ra,Imm_{15}

39 34	33	32	31 24	23	22	21 16	15	14	13 8	7 5	4 0
13_6	~	1	Imm_8	Va	Sa	Ra_6	Vt	St	Rt_6	Sz_3	2_5

Clock Cycles: 1

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Bit Manipulation Operations

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

Flag Updates: none

Operation:

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

Instruction Formats:

CLR Rt, Ra, Rb

39 36	35 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
0_4	$\sim_5$	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

CLR Rt, Ra,Offs$_7$,Wid$_4$

39 36	35 30	29 23	22	21 16	15	14 9	8	7 5	4 0
8_4	Wid_6	$Offs_7$	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

CLR Rt, Ra,Imm$_8$,Imm$_8$

39 36	35 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
8_4	$\sim_4$	~	~	$\sim_7$	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

$\sim_{16}$	$Width_8$	$Offset_8$	0_3	31_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

COM – Complement Bit Field

Description:

A bit 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:

COM Rt, Ra, Rb

39 34	35 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
2_4	$\sim_5$	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

COM Rt, Ra,Offs$_7$,Wid$_6$

39 34	35 30	29 23	22	21 16	15	14 9	8	7 5	4 0
10_4	Wid_6	$Offs_7$	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

COM Rt, Ra,Imm$_8$,Imm$_8$

<table>
<tr><td>39 34</td><td>3332</td><td>31</td><td>30</td><td>29 23</td><td>22</td><td>21 16</td><td>15</td><td>14 9</td><td>8</td><td>7 5</td><td>4 0</td></tr>
<tr><td>10_4</td><td>$\sim_2$</td><td>~</td><td>~</td><td>$\sim_7$</td><td>Sa</td><td>Ra_6</td><td>St</td><td>Rt_6</td><td>V</td><td>Sz_3</td><td>13_5</td></tr>
<tr><td colspan="5">$\sim_{16}$</td><td colspan="3">$Width_8$</td><td colspan="2">$Offset_8$</td><td>0_3</td><td>31_5</td></tr>
</table>

Clock Cycles: 1

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

39 36	35 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_4	$\sim_5$	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

SET Rt, Ra,Offs$_7$,Wid$_6$

39 36	35 30	29 23	22	21 16	15	14 9	8	7 5	4 0
9_4	Wid_6	$Offs_7$	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

SET Rt, Ra,Offset$_8$,Width$_8$

39 36	3532	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
9_4	$\sim_4$	~	~	$\sim_7$	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

$\sim_{16}$	$Width_8$	$Offset_8$	0_3	31_5

Clock Cycles: 1

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

Operation:

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

Instruction Formats:

EXTS Rt, Ra, Rb

39 36	35 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
5_4	$\sim_5$	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

EXTS Rt, Ra,Offs$_7$,Wid$_6$

39 36	35 30	29 23	22	21 16	15	14 9	8	7 5	4 0
13_4	Wid_6	$Offs_7$	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

EXTS Rt, Ra,Offset$_8$,Width$_8$

39 36	35 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
13_4	$\sim_4$	~	~	$\sim_7$	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5
$\sim_{16}$					$Width_8$		$Offset_8$			0_3	31_5

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

39 36	35 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_4	$\sim_5$	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

Clock Cycles: 1

EXTU Rt, Ra,Offs$_7$,Wid$_6$

39 36	35 30	29 23	22	21 16	15	14 9	8	7 5	4 0
12_4	Wid_6	$Offs_7$	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

EXTU Rt, Ra,Offset$_8$,Width$_8$

39 36	3532	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
12_4	$\sim_4$	~	~	$\sim_7$	Sa	Ra_6	St	Rt_6	V	Sz_3	13_5

$\sim_{16}$	$Width_8$	$Offset_8$	0_3	31_5

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Shift and Rotate Operations

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
0_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ASL Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
32_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

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

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
1_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ASR Rt, Ra, Imm_7

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
33_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

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 of the register and place the result in the target register. All registers are integer registers.

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

Operation:

Instruction Formats:

SXB Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
9_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

SXB Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
41_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

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.

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

Operation:

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

Instruction Formats:

LSL Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
2_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

LSL Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
34_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

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:

LSR Rt, Ra, Rb

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
3_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

LSR Rt, Ra, Imm_7

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
35_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ROL Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
36_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

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

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
5_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ROR Rt, Ra, Imm$_7$

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
37_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

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. This is also called a slide operation.

Instruction Formats:

VSHLV Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
16_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

VSHLV Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
48_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

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.

VSHLR Rt, Ra, Rb

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
17_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

VSHLR Rt, Ra, Imm_7

39 34	3332	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
49_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

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

ZBX – Zero Bit Extend

Description:

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

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

Operation:

Rt = Zero Extend(Ra)

Instruction Formats:

ZXB Rt, Ra, Rb

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
8_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

ZXB Rt, Ra, Imm_7

39 34	33 32	31	30	29 23	22	21 16	15	14 9	8	7 5	4 0
40_6	$\sim_2$	0	0	Imm_7	Sa	Ra_6	St	Rt_6	V	Sz_3	14_5

Clock Cycles: 1

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Flow Control Instructions

Bcc – Conditional Branch

Bcc Pn, label

Description:

Branch if the predicate condition is met. The displacement is relative to the address of the branch instruction. The branch range is +/- 8MB.

Instruction Format:

39 ... 16	15 ... 10	9 ... 5	4 ... 0
$Disp_{23..0}$	Rn_6	$Cond_5$	28_5

$Cond_5$	Mnem.	Meaning	Test
		Integer Compare Results	
0	EQ	= equal	
1	LT	< less than	
2	LE	<= less than or equal	
3	LO / CS	< unsigned less than	
4	LS	<= unsigned less than or equal	
5	LEZ	<= 0	
6	Z	0	
7	LTZ / MI	< 0	
8	NE	< > not equal	
9	GE	>= greater than or equal	
10	GT	> greater than	
11	HS / CC	unsigned greater than or equal	
12	HI	unsigned greater than	
13	GEZ	>= 0	
14	NZ	Not 0	
15	GTZ	>0	
$Cond_5$	Mnem.	Meaning	Test
		Float Compare Results	
16	EQ	equal	!nan & eq
17	NE	not equal	!eq
18	GT	greater than	!nan & !eq & !lt & !inf
19	UGT	Unordered or greater than	Nan || (!eq & !lt & !inf)
20	GE	greater than or equal	Eq || (!nan & !lt & !inf)
21	UGE	Unordered or greater than or equal	Nan || (!lt || eq)
22	LT	Less than	Lt & (!nan & !inf & !eq)
23	ULT	Unordered or less than	Nan | (!eq & lt)
24	LE	Less than or equal	Eq | (lt & !nan)
25	ULE	unordered less than or equal	Nan | (eq | lt)
26	GL	Greater than or less than	!nan & (!eq & !inf)
27	UGL	Unordered or greater than or less than	Nan | !eq
28	ORD	Greater than less than or equal / ordered	!nan
29	UN	Unordered	Nan
$Cond_5$	Mnem.	Meaning	Test
30	TW	Three-way branch	
31	SR	Branch subroutine	

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 +/- 8MB.

Instruction Format:

39 … 16	15 … 10	9 … 5	4 … 0
$Disp_{23..0}$	0_6	6_5	28_5

Clock Cycles: 3

BRK – Breakpoint

Description:

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

Instruction Format:

39 … 5	4 … 0
0	0_5

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 +/- 8MB.

Instruction Format:

39 … 16	15 … 10	9 … 5	4 … 0
$Disp_{23..0}$	Rt_6	31_5	28_5

Clock Cycles: 3

BTW – Branch Three-way

BTW Pn, labelLT, labelEQ, labelGT

Description:

Branch one of three ways. If the EQ bit is set, branch to the EQ label, otherwise if the LT bit is set, branch to the LT label, otherwise branch to the GT label. The displacement is relative to the address of the branch instruction. The branch range is +/- 128B, about 25 instructions.

Instruction Format:

39 … 32	31 … 24	23 … 16	15 … 10	9 … 5	4 … 0
$DispLT_{7..0}$	$DispEQ_{7..0}$	$DispGT_{7..0}$	Rn_6	30_5	28_5

DBcc – Decrement and Branch

DBcc 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 +/- 8MB.

Instruction Format:

39 16	15 10	9 5	4 0
Disp$_{23..0}$	Rn$_6$	Cond$_5$	29$_5$

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)

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
24_6	Op_2	0	0	Sb	Rb_6	Sa	Ra_6	0	0_6	0	Sc_3	2_5

Clock Cycles: 1

JMP Imm_{16} (Ra)

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	0	$Imm_{7..0}$	Sa	Ra_6	0	0_6	0	Sz_3	24_5

Clock Cycles: 1

JMP [d(Ra, Rb)]

39 34	33 32	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
24_6	Op_2	1	0	Sb	Rb_6	Sa	Ra_6	0	0_6	0	Sc_3	2_5

Clock Cycles: 1

JMP [Imm_{16} (Ra)]

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	1	$Imm_{7..0}$	Sa	Ra_6	0	0_6	0	Sz_3	24_5

Clock Cycles: 1

Op_2	
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.

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 (Ra, Rb)

39 34	3332	31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
24_6	$\sim_2$	0	Vb	Sb	Rb_6	Sa	Ra_6	0	Rt_6	V	Sz_3	2_5

Clock Cycles: 1

JSR Imm_{16} (Ra)

39 32	31	30 23	22	21 16	15	14 9	8	7 5	4 0
$Imm_{15..8}$	S	$Imm_{7..0}$	Sa	Ra_6	St	Rt_6	V	Sz_3	24_5

Clock Cycles: 1

NOP – No Operation

NOP

Description:

This instruction does not perform any operation. Ty_3 0 to 3 indicates a postfix instruction, and these codes should not be used for other NOPs.

Instruction Format:

39 8	7 5	4 0
$Payload_{32}$	Ty_3	31_5

RTE – Return From Exception

Instruction Formats:

RTE #Rpt

39 34	33 31	3029	28	27	26 21	20	19 14	13	12 7	6 5	4 0
4_6	$\sim_3$	$\sim_2$	0	~	$\sim_6$	~	$\sim_6$	R_6	Rpt_6	D_2	1_5

Field Description:

Rpt_7 is 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 1,2 or 3. In most cases a single entry is unstacked. If two entries are unstack 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 Rpt tetras to the program counter, and Arg tetras 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.

Instruction Format:

TRAP Rt, Ra, #Data

39 33	32	31 24	23	22	21 16	15	14	13 8	7 5	4 0
$Imm_{14..8}$	0	$Imm_{7..0}$	Va	Sa	Ra_6	0	0	Rt_6	0_3	0_5

TRAP Rt, #Vec, #Data

39 33	32	31 24	23 16	15	14	13 8	7 6	5	4 0
$Imm_{14..8}$	1	$Imm_{7..0}$	Vec_8	0	0	Rt_6	$\sim_2$	V_9	0_5

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, .n

Instruction Formats: AMO

AMADD Rt, Rb, [Ra]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
4_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMADD Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
20_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

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]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
8_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMAND Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
24_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

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]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
6_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMASL Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
22_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

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]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
10_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMEOR Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
26_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

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]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
7_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMLSR Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
23_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

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]

39 35	34	33	32 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
2_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

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]

39 35	34	33	32 31	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
12_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

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]

39 35	34	33	3231	30	29	28 23	22	21 16	15	14 9	8	7 5	4 0
9_5	aq	rl	0_2	Vb	Sb	Rb_6	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

AMOR Rt, imm, [Ra]

39 35	34	33	3231	30 23	22	21 16	15	14 9	8	7 5	4 0
25_5	aq	rl	0	Imm_8	Sa	Ra_6	St	Rt_6	V	Sz_3	26_5

Clock Cycles:

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 is 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
~	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; 96-bit triple precision.

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

Sz_3	Ext.	Operand
0		reserved
1	.h	16-bit half
2	.s	32-bit single
3	.d	64-bit double
4		Reserved
5		Reserved
6	.t	96-bit triple
7		reserved

Instruction Formats: NDXL

FLOAD Rt, d(Rb,Rc*Sc) – indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
1	Vc	Rc_6	Sc	Vb	Sb	Rb_6	St	Rt_6	$D_{6..0}$	V	Sz_3	16_5

Clock Cycles:

Notes:

FSTORE Ra,<ea>

Description:

Store register Ra to destination. The register is converted from triple 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		Reserved
5		reserved
6	.t	96-bit triple
7		reserved

Instruction Formats: NDXS

STORE Ra, d(Rb, Rc*Sc) – Indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
1	Vc	Rc_6	Sc	Vb	Sb	Rb_6	Sa	Ra_6	$D_{6..0}$	V	Sz_3	18_5

Clock Cycles:

Notes:

LOAD Rn,<ea>

Description:

Load register Rt from source. The source value is sign extended to the machine width. Loading register r54, 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	.c	24-bit
5	.p	40-bit
6	.n	96-bit
7		group

Instruction Formats: NDXL

LOAD Rt, d(Rb,Rc*Sc) – indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
0	Vc	Rc_6	Sc	Vb	Sb	Rb_6	St	Rt_6	$D_{6..0}$	V	Sz_3	16_5

Clock Cycles:

Notes:

LOADG Gn,<ea>

Description:

Load group of five registers from source.

Gn	Registers
0	R0 to R4
1	R5 to R9
2	R10 to R14
3	R15 to R19
4	R20 to R24
5	R25 to R29
6	R30 to R34
7	R35 to R39

Gn	Registers
8	R40 to R44
9	R45 to R49
10	R50 to R54
11	R55 to R59
12	R60 to R64
13	ASP, SSP, HSP, MSP

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

Instruction Formats: NDXL

LOADG Gt, d(Rb,Rc*Sc) – indexed

39	38 33	32	31	30	29 24	23	22	21 20	19 16	15 8	7 5	4 0
Vc	Rc_6	Sc	Vb	Sb	Rb_6	Vt	~	$\sim_2$	Gt_4	$D_{7..0}$	7_3	16_5

Clock Cycles:

Notes:

LOADZ Rn,<ea>

Description:

Load register Rt from source. The source value is zero extended to the machine width. Loading register r54, 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	.c	24-bit
5	.p	40-bit
6	.n	96-bit
7		reserved

Instruction Formats: NDXL

LOAD Rt, d(Rb,Rc*Sc) – indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
~	Vc	Rc_6	Sc	Vb	Sb	Rb_6	St	Rt_6	$D_{6..0}$	V	Sz_3	17_5

Clock Cycles:

Notes:

STORE Ra,<ea>

Description:

Store register Ra 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: NDXS

STORE Ra, d(Rb, Rc*Sc) – Indexed

39	38	37 32	31	30	29	28 24	22	21 16	15 9	8	7 5	4 0
0	Vc	Rc_6	Sc	Vb	Sb	Rb_6	Sa	Ra_6	$D_{6..0}$	V	Sz_3	18_5

Clock Cycles:

Notes:

STOREG Gt,<ea>

Description:

Store register group to destination. The destination is a 512 bit / 64 byte aligned region of memory.

Gn	Registers
0	R0 to R4
1	R5 to R9
2	R10 to R14
3	R15 to R19
4	R20 to R24
5	R25 to R29
6	R30 to R34
7	R35 to R39

Gn	Registers
8	R40 to R44
9	R45 to R49
10	R50 to R54
11	R55 to R59
12	R60 to R64
13	ASP, SSP, HSP, MSP

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

Instruction Formats: NDXS

STOREG Ga, d(Rb, Rc*Sc) – Indexed

39	38 33	32	31	30	29 24	23	22	21 20	19 16	15 8	7 5	4 0
Vc	Rc_6	Sc	Vb	Sb	Rb_6	Va	~	$\sim_2$	Ga_4	$D_{7..0}$	7_3	18_5

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]
```

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 compare operation into a mask register.

Instruction Format: R2

V2BITS Rt, Ra, Rb – Register direct

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
40_6	0	0	0	0	Rb_6	Va	Sa	Ra_6	0	St	Rt_6	Sz_3	2_5

Clock Cycles: 1

V2BITS Rt, Ra, #bit – Register direct

39 34	33	32	31	30 24	23	22	21 16	15	14	13 8	7 5	4 0
40_6	1	0	0	Imm_7	Va	Sa	Ra_6	0	St	Rt_6	Sz_3	2_5

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 m1,v1,#8     ; move NE status to bits in m1
vmask “11100000”
add v4,v5,v6        ; perform some masked vector operations
muls v7,v8,v9
add v7,v7,v4
```

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	~$_2$	~$_6$	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	~$_2$	~$_6$	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

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
56_6	~	0	0	0	0_6	Va	Sa	Ra_6	Vt	St	Rt_6	6_3	2_5

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

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
57_6	~	0	0	0	0_6	Va	Sa	Ra_6	Vt	St	Rt_6	6_3	2_5

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

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
58_6	~	0	0	0	0_6	Va	Sa	Ra_6	Vt	St	Rt_6	6_3	2_5

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

39 34	33	32	31	30	29 24	23	22	21 16	15	14	13 8	7 5	4 0
59_6	~	0	0	0	0_6	Va	Sa	Ra_6	Vt	St	Rt_6	6_3	2_5

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

31 25	24 22	21	20 15	14 9	8	7 0
$34h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

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

31 25	24 22	21	20 15	14 9	8	7 0
$35h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

Exceptions: none

SHA512SUM0

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$36h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

SHA512SUM1

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$37h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

SM3P0

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$38h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

SM3P1

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$39h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

SM4ED

Description:

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$56h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

SM4KS

Description:

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$57h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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 Bit	
MASK Modifier Scope	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:

39 34	33 32	31 24	23 16	15	14 9	8	7 5	4 0
35_6	$\sim_2$	$Imm_{15..8}$	$Imm_{7..0}$	St	Rt_6	V	Sz_3	2_5

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.

	Mask Bit	
Carry Modifier Scope	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:

39 34	33	32	31 16	15	14	13 8	7 5	4 0
33_6	~	0	$Imm_{15..0}$	~	0	Rn_6	$\sim_3$	2_5

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.

MASK 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:

39 34	33	32	31 8	7 5	4 0
34_6	~	1	$Imm_{23..0}$	$\sim_3$	2_5

Assembler Syntax:

Specifying the mask register for following instructions consists of a map using single digit numeric characters between ‘0’ and ‘7’. A character is present in a string for each following instruction in sequence.

Example:

```
VMASK ”12345000”
ADD v6,v3,v7          ; vector mask reg #1
ADD v6,v6,#1234       ; vector mask reg #2
ADD v6,v2,v1          ; vector mask reg #3
ADD v6,v6,#456        ; vector mask reg #4
ADD v7,v6,#456        ; vector mask reg #5
ADD v8,v7,#987        ; vector mask reg #0
MUL v8,v9,v10         ; vector mask reg #0
```

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.

	Mask Bit	
Pred Modifier Scope	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:

39 34	33	32	31 16	15 10	9 5	4 0
32_6	~	1	$Imm_{15..0}$	Rn_6	$Cond_5$	2_5

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:

39 34	33	32	31 8	7 5	4 0
36_6	~	1	$Imm_{23..0}$	$\sim_3$	2_5

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.

<table>
<tr><th>Regno</th><th>Access</th><th>Moniker</th><th>Purpose</th></tr>
<tr><td>00</td><td>R</td><td>CAUSE</td><td>interrupt cause code for currently interrupting source</td></tr>
<tr><td>04</td><td>RW</td><td>RE</td><td>request enable, a 1 bit indicates interrupt requesting is enabled for that interrupt, a 0 bit indicates the interrupt request is disabled.</td></tr>
<tr><td>08</td><td>W</td><td>ID</td><td>Disables interrupt identified by low order five data bits.</td></tr>
<tr><td>0C</td><td>W</td><td>IE</td><td>enables interrupt identified by low order five data bits</td></tr>
<tr><td>10</td><td></td><td></td><td>reserved</td></tr>
<tr><td>14</td><td>W</td><td>RSTE</td><td>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.</td></tr>
<tr><td>18</td><td>W</td><td>TRIG</td><td>software trigger of the interrupt specified by the low order five data bits.</td></tr>
<tr><td>20</td><td>W</td><td>ESL</td><td>The low bit for edge sensitivity selection. ESL and ESH combine to form a two bit select of the edge sensitivity.
<table>
<tr><td>ESH,EHL</td><td>Sensitivity</td></tr>
<tr><td>00</td><td>level sensitive interrupt</td></tr>
<tr><td>01</td><td>positive edge sensitive</td></tr>
<tr><td>10</td><td>negative edge sensitive</td></tr>
<tr><td>11</td><td>either edge sensitive</td></tr>
</table></td></tr>
<tr><td>24</td><td>W</td><td>ESH</td><td>The high bit for edge sensitivity selection</td></tr>
<tr><td>80</td><td>RW</td><td>CTRL0</td><td>control register for interrupt #0</td></tr>
<tr><td>84</td><td>RW</td><td>CTRL1</td><td>control register for interrupt #1</td></tr>
<tr><td>…</td><td></td><td>…</td><td></td></tr>
<tr><td>FC</td><td>RW</td><td>CTRL31</td><td>control register for interrupt #31</td></tr>
</table>

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.