Preface

Who This Book is For

This book describes the Thor2024 ISA. It is for anyone interested in instruction set architectures.

About the Author

First a warning: I'm an enthusiastic hobbyist like yourself, with a ton of experience. I've spent a lot of time at home doing research and implementing several soft-core processors, almost maniacally. One of the first cores I worked on was a 6502 emulation. I then went on to develop the Butterfly32 core. Later the Raptor64. I have about 25 years professional experience working on banking applications at a variety of language levels including assembler. So, I have some real-world experience developing complex applications. I also have a diploma in electronics engineering technology. Some of the cores I work on these days are too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come. To some extent larger boards have arrived. The author is a bit wary of larger boards. Larger FPGAs increase build times by their nature.

Motivation

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

Having worked on Thor2023 for several months, the author finally realized that it did not have very good code density. Thor2022 was better in that regard. So, Thor2024 is a mix of the best from previous designs. Thor2024 aims to improve code density over earlier versions.

Some efficiency is being traded off for design simplicity. Some of the most efficient designs are 32-bit.

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

Nomenclature

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

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

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

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

Little Endian vs big Endian

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

Endian

Thor2024 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

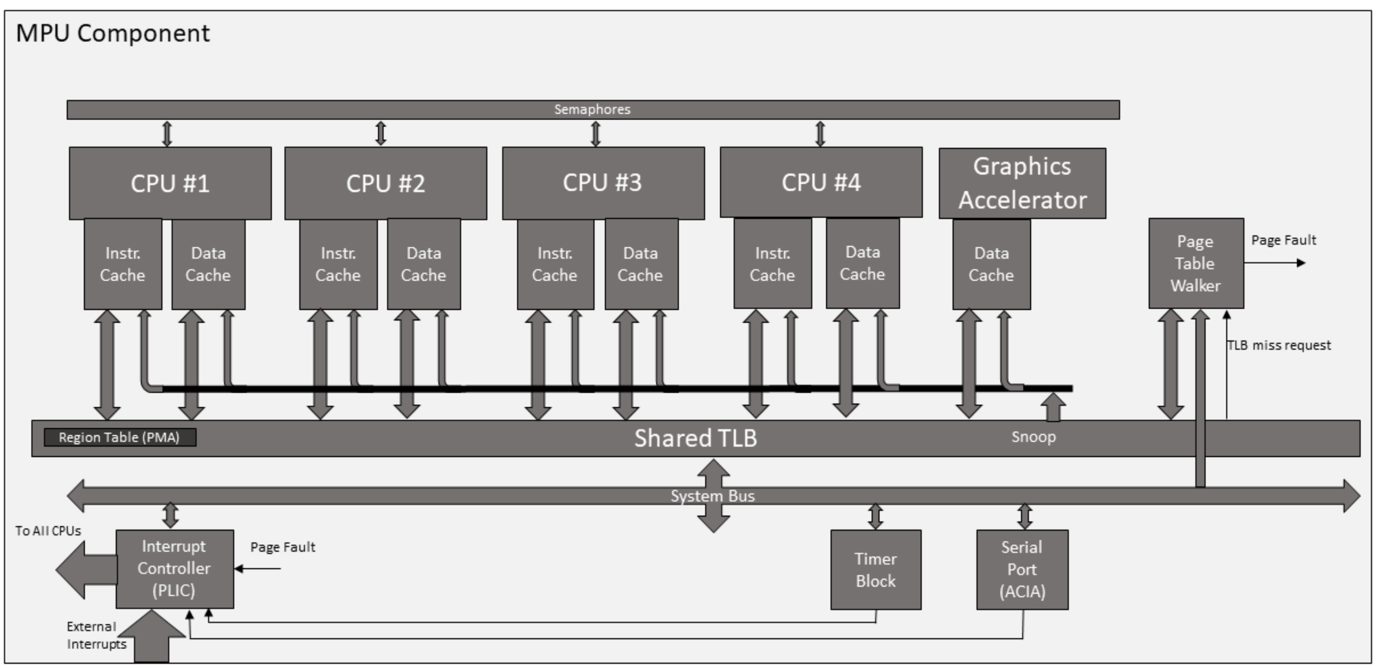
Big Endian:

Address	3	2	1	0
Byte	0	1	2	3

For Thor2024 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	$Opcode_5$

MPU Block Diagram



Programming Model

Register File

Rn – General Purpose Registers

The register file contains 63 128-bit general purpose registers.

The register file is *unified* and may hold either integer or floating-point values. The stack pointer, register 62, is banked with a separate stack pointer for each operation mode.

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

Register r0 is special in that it always reads as a zero.

Register code 63 is used to indicate the use of a postfix immediate in place of the register operand. R63 is also acts as PR15 used to indicate a predicate value of all ones.

Register ABI

Regno	ABI	Group Reg	ABI Usage
0	0	AG0	Always zero
1	A0		First argument / return value register
2	A1		Second argument / return value register
3	A2		Third argument register
4 to 15	T0 to T11	TG0 to TG2	Temporary register, caller save
16 to 31	S0 to S15	SG0 to SG3	Saved register, register variables
32 to 39	PR0 to PR7	PG0, PG1	Predicate registers
40 to 43	A3 to A6 or PR8 to PR11	AG2	Argument register
44 to 47	A7 to A10 or PR12 to PR14		Argument register
48 to 51	MC0 to MC3	MCG	Micro-code
52	TS		Thread State
53	PC , SC		Program counter / Stack Canary
54	CTA		Card table address
55	LC		Loop Counter
56	LR0	LG0	Link register #0
57	LR1		Link register #1
58	LR2		Micro-code link register
59	GP1		Global Pointer #1
60	GP		Global Pointer
61	FP		Frame Pointer
62	ASP		Application / User stack pointer
62	SSP		Supervisor Stack pointer
62	HSP		Hypervisor Stack pointer
62	MSP		Machine Stack pointer

Pn - Predicate Registers

There are 8 128-bit registers selectable as predicate registers. Most instructions support a three-bit predicate register field to select one of these registers. Predicate registers are part of the general-purpose register file and may be manipulated using the same instructions as for other registers.

Predicate registers are used to mask off vector operations so that a vector instruction doesn't perform the operation on all elements of the vector. They are also used as Boolean predicate values for scalar operations.

Code Address Registers

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

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

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

LRn – Link Registers

There are two registers in the Thor2024 architecture reserved for subroutine linkage. These registers are used to store the address after the calling instruction. They may be used to implement fast returns for two levels of subroutines or to used to call milli-code routines. The jump to subroutine, JSR, and branch to subroutine, BSR, instructions update a link register. The return from subroutine,. RTS, instruction is used to return to the next instruction.

PC – Program Counter

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

LC - Loop Counter (reg 55)

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

SR - Status Register (CSR 0x?004)

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

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

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

T indicates that trace mode is active.

OM processor operating mode.

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

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

IPL is the interrupt mask level

RT specifies the return type for an RTI instruction.

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

Decimal Mode

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

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

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

Vector Programming Model

Register File

Vn – SIMD Registers

The SIMD register file contains 64 512-bit registers.

Regno	ABI	ABI Usage
0		
1	VA0	First argument / return value
2	VA1	Second argument / return value
3	VA2	Third argument
4 to 15	VT0 to VT11	
16 to 31	VS0 to VS15	
32 to 39	VA3 to VA10	

Vector Related CSRs

VGM		Global mask register
VRM		Restart mask register
VERR		Error mask register
VRGSZ		Vector register size
VED		Vector element descriptor

The number of elements is limited to 128 as that is the width of a predicate register.

Vector Global Mask Register (VGM)

The global mask register contains predicate bits indicating which vector elements are active. Vector elements of the target are updated only when the corresponding global mask bit is set. The global mask register takes the place of the vector length register in other architectures. Normally the global mask contains a right aligned bitmask of all ones up to the number of elements to be processed.

Vector Restart Mask Register (VRM)

The restart mask register contains a bitmask indicating the vectors elements to be processed after a restart. The restart mask register is set to all ones at the end of a vector operation.

Vector Error Mask Register (VERR)

The vector error mask register contains a bit for each vector element indicating if an error occurred.

Vector Register Size (VRGSZ)

The vector register size register contains the length of a vector register in bytes. Only the low order eight bits of the register are implemented, other bits read as zero, and ignore writes.

Vector Element Description Register (VED)

This register contains bits describing an element of a vector.

	127 6	5 3	2 0
	~	OT_3	$Size_3$

$Size_3$	Bits	Bytes
0	8	1
1	16	2
2	32	4
3	64	8
4	128	16
5	256	32
6	512	64
7		reserved

OT_3	Operand Type
0	Integer
1	Float
2	Decimal
3	Posit
4	Char
5 to 7	reserved

Register-Register Format

Fmt$_3$	Rb	Ra	Rt	Mask
000	scalar	scalar	scalar	No
001	scalar	scalar	scalar	Yes
010	scalar	vector	vector	No
011	scalar	vector	vector	Yes
100	vector	vector	vector	No
101	vector	vector	vector	Yes

Register-Immediate Vector Decode

Fmt$_2$	Ra	Rt	Mask
00	scalar	scalar	No
01	scalar	scalar	Yes
10	vector	vector	No
11	vector	vector	Yes

Special Purpose Registers

SC - Stack Canary (GPR 53)

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

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

See status register.

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

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 12 bits are implemented. The high order bits read as zero and are not updateable.

U_REPBUF - (CSR – 0x008)

This register contains information needed for the REP instruction that must be saved and restored during context switches and interrupts. Note that the loop counter should also be saved.

127 112	121 48	47 44	43	42 40	39 8	7	6 0
Resv	pc	Resv2	V	ICnt	Limit	resv	Ins[15:9]

Pc: (64 bits) the address of the instruction following the REP
V: REP valid bit, 1 only if a REP instruction is active
ICnt: the current instruction count, distance from REP instruction.
Limit: a 32-bit amount to compare the loop counter against.
Ins: bits 9 to 15 of the REP instruction which contains the instruction count of instruction included in the repeat and condition under which the repeat occurs.

[U/S/H/M]_SCRATCH – CSR 0x?041

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

S_PTBR (CSR 0x1003)

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

63 12	11 8	7 6	5 4	3	2 1	0
Page Table Address$_{67..16}$	Levels	AL_2	$\sim_2$	S	~	Type

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

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

Levels are ignored for the inverted page table. For a normal page table gives the top entry level.

AL_2: TLB entry replacement algorithm, 0=fixed,1=LRU,2=random,3=reserved

S_ASID (CSR 0x101F)

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

S_KEYS (CSR 0x1020 to 0x1027)

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

			23 0
1020			key0
1021			key1
…			…
1027			key7

M_CORENO (CSR 0x3001)

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

M_TICK (CSR 0x3002)

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

M_SEED (CSR 0x3003)

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

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

M_BAD_INSTR (CSR 0x300B)

This register contains a copy of the exceptioned instruction.

M_SEMA (CSR 0x300C)

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

M_TVEC – CSR 0x3030 to 0x3034

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

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

Reg #	
0x3030	TVEC[0] – user mode

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

M_SR_STACK (CSR 0x303C to CSR 0x303D)

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

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

M_IOS – IO Select Register (CSR 0x3100)

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

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

M_EPC (CSR 0x3108 to 0x310F)

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

Reg #	Name
0x3108	EIP0
…	
0x310F	EIP7

AV – Application Vector Table Address

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

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

VB – Vector Base Register

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

63 3 2 1 0

Vector Table $Address_{63..3}$	~	~

Operating Modes

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

A subset of instructions is limited to machine mode.

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

Caches

Overview

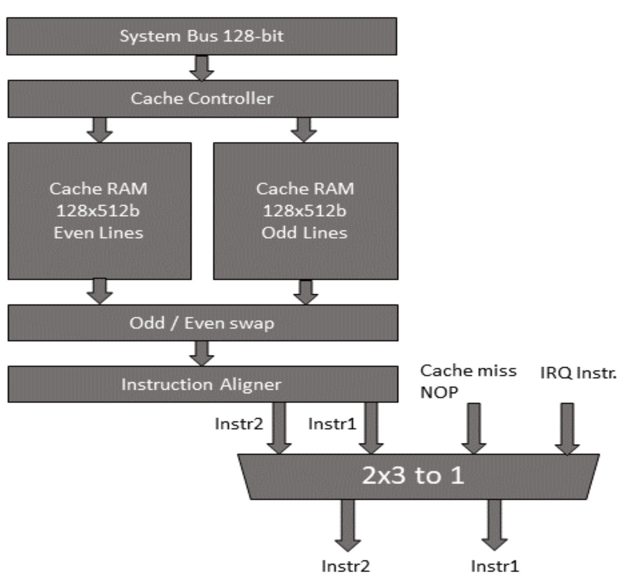
The core has both instruction and data caches to improve performance. Both caches are single level. The cache is four-way associative. The cache sizes of the instruction and data cache are available for reference from one of the info lines return by the CPUID instruction.

Instructions

Since the instruction format affects the cache design it is mentioned here. For this design instructions are of a fixed 40-bit format. Specific formats are listed under the instruction set description section of this book. Initially some thought was given to supporting a 16-bit instruction parcel as there would be a code density and therefore cache utilization benefit. A fixed 40-bit format was chosen because it's simpler for a hobbyist design and to limit the amount of multiplexing taking place for an instruction read. The author found that determining the length of an instruction and selecting the next instruction to fetch based on a varying length affected the timing of the core too much. A fixed size instruction allowed a faster timing rate. A simpler design makes it easier to achieve a higher clock rate. Having multiple instruction sizes would require additional multiplexing, logic, and possibly wide or more read ports on the cache.

L1 Instruction Cache

L1 is 16kB in size and made from block RAM with a single cycle of latency. L1 is organized as an odd, even pair of 128 lines of 64 bytes. Note that there are two copies of the L1 cache to support reading two instructions at one time. The following illustration shows the L1 cache organization for Thor2024.



The cache is organized into odd and even lines to allow instructions to span a cache line. Two cache lines are fetched for every access; the one the instruction is located on, and the next one in case the instruction spans a line.

A 128-line cache was chosen as that matches the inherent size of single distributed ram component in the FPGA. It is the author's opinion that it would be better if the L1 cache were larger because it often misses due to its small size. In short the current design is an attempt to make it easy for the tools to create a fast implementation.

Note that supporting interrupts and cache misses, a requirement for a realistic processor design, adds complexity to the instruction stream. Reading the cache ram, selecting the correct instruction word and accounting for interrupts and cache misses must all be done in a single clock cycle.

While the L1 cache has single cycle reads it requires two clock cycles to update (write) the cache. The cache line to update needs to be provided by the tag memory which is unknown until after the tag updates.

Data Cache

The data cache organization is somewhat simpler than that of the instruction cache. Data is cached with a single level cache because it's not critical that the data be available within a single clock cycle at least not for the hobby design. Some of the latency of the data cache can be hidden by the presence of non-memory operating instructions in the instruction queue.

The data cache is organized as 512 lines of 64 bytes (32kB) and implemented with block ram. Access to the data cache is multicycle. The data cache may be replicated to allow more memory instructions to be processed at the same time; however, just a single cache is in use for the demo system. The policy for stores is write-through. Stores always write through to memory. Since stores follow a write-through policy the latency of the store operation depends on the external memory system. It isn't critical that the cache be able to update in single cycle as external memory access is bound to take many more cycles than a cache update. There is only a single write port on the data cache.

Cache Enables

The instruction cache is always enabled to keep hardware simpler and faster. Otherwise, an additional multiplexor and control logic would be required in the instruction stream to read from external memory.

For some operations, it may desirable to disable the data cache so there is a data cache enable bit in control register #0. This bit may be set or cleared with one of the CSR instructions.

Cache Validation

A cache line is automatically marked as valid when loaded. The entire cache may be invalidated using the CACHE instruction. Invalidating a single line of the cache is not currently supported, but it is supported by the ISA. The cache may also be invalidated due to a write by another core via a snoop bus.

Un-cached Data Area

The address range $F…FDxxxxx is an un-cached 1MB data area. This area is reserved for I/O devices. The data cache may also be disabled in control register zero. There is also field in the load instructions that allows bypassing the data cache.

Fetch Buffers

There are two fetch buffers each of which holds a pair of instructions. When a fetch buffer becomes empty it is loaded with new instructions from the cache. While the processor is working with instructions from one fetch buffer, the other fetch buffer can be loading more instructions. In the case of a cache miss or interrupt a special instruction is loaded into the fetch buffer rather than the instruction output by the cache. For a cache miss this is the NOP instruction. For an interrupt this is the BRK instruction. The program counter increment is suppressed during a cache miss.

Fetch buffers may also be loaded from a micro-code store when a macro instruction is fetched.

The program counters are located in the fetch buffer component.

When SMT is enabled one half of the fetch buffers is used for each thread.

Fetch Rate

The fetch rate is two instructions per clock cycle. When SMT is on one instruction is fetched for each thread. This is fine-grained SMT.

Return Address Stack Predictor (RSB)

There is an address predictor for return addresses which can in some cases can eliminate the flushing of the instruction queue when a return instruction is executed. The RETD instruction is detected in the fetch stage of the core and a predicted return address used to fetch instructions following the return. The return address stack predictor has a stack depth of 64 entries. On stack overflow or underflow, the prediction will be wrong, however performance will be no worse than not having a predictor. The return address stack predictor checks the address of the instruction queued following the RET against the address fetched for the RET instruction to make sure that the address corresponds.

There is a separate RSB for each thread while operating with SMT turned on.

Branch Predictor

*Not implemented yet.

The branch predictor is a (2, 2) correlating predictor. The branch history is maintained in a 512-entry history table. It has four read ports for predicting branch outcomes, one port for each instruction in the fetch buffer. The branch predictor may be disabled by a bit in control register zero. When disabled all branches are predicted as not taken, unless specified otherwise in the branch instruction. A statically predicted branch does not use the branch predictor instead the prediction is based on the setting of the prediction bits in the branch instruction.

To conserve hardware the branch predictor uses a fifo that can queue up to two branch outcomes at the same time. Outcomes are removed from the fifo one at a time and used to update the branch history table which has only a single write port. In an earlier implementation of the branch predictor, two write ports were provided on the history table. This turned out to be relatively large compared to its usefulness.

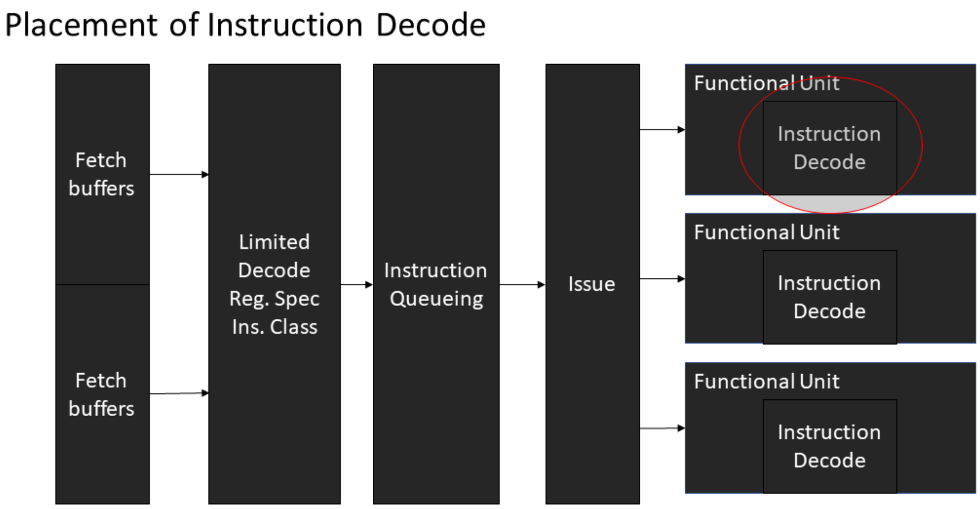
Correctly predicting a branch turns the branch into a single cycle operation. During execution of the branch instruction the address of the following instruction queued is checked against the address depending on the branch outcome. If the address does not match what is expected, then the queue will be flushed, and new instructions loaded from the correct program path.

Branch Target Buffer (BTB)

The core has a 1k entry branch target buffer for predicting the target address of flow control instructions where the address is calculated and potentially unknown at time of fetch. Instructions covered by the BTB include jump-and-link, interrupt return and breakpoint instructions and branches to targets contained in a register.

Decode Logic

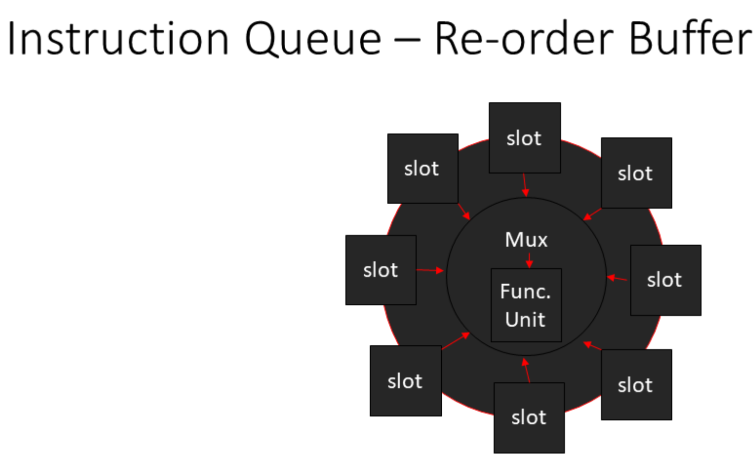
Instruction decode is distributed about the core. Although a number of decodes take place between fetch and instruction queue. Broad classes of instructions are decoded for the benefit of issue logic along with register specifications prior to instruction enqueue. Most of the decodes are done with functions defined early in Thor2024pkg.sv because decoding typically involves reducing a wide input into a smaller number of output signals. Other decodes are done at instruction execution time with case statements.



Limited decode takes place between fetch and queue. Between fetch and queue register specifications are decoded along with general instruction classes for the benefit of issue. A handful of additional signals (like sync) that control the overall operation of the core are also decoded. Much of the instruction decode is actually done in the functional unit. The instruction register is passed right through to the functional units in the core.

Instruction Queue (ROB)

The instruction queue is an eight-entry re-ordering buffer (ROB). The instruction queue tracks an instructions progress and provides a holding place for operands and results. Each instruction in queue may be in one of several different states. The instruction queue is a circular buffer with head and tail pointers. Instructions are queued onto the tail and committed to the machine state at the head.



The instruction queue is circular with eight slots. Each slot feeds a multiplexor which in turn feeds a functional unit. Providing arguments to the functional unit is done under the vise of issue logic. Output from the functional unit is fed back to the same queue slot that issued to the functional unit.
The queue slots are fed from the fetch buffers.

Queue Rate

Up to two instructions may queue during the same clock cycle depending on the availability of queue slots.

Sequence Numbers

The queue maintains a 6-bit instruction sequence number which gives other operations in the core a clue as to the order of instructions. The sequence number is assigned when an instruction queues. Branch instructions need to know when the next instruction has queued to detect branch misses. A separate sequence number is maintained for each hardware thread. The program counter cannot be used to determine the instruction sequence because there may be a software loop at work which causes the program counter to cycle backwards even though it’s really the next instruction executing.

Memory Management

Bank Swapping

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

The Page Map

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

Regions

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

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

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

PMA - Physical Memory Attributes Checker

Overview

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

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

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

Physical memory attributes checking is applied in all operating modes.

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

Region Table Description

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

PMT Address

The PMT address specifies the location of the associated PMT.

CTA – Card Table Address

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

Attributes

<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>~</td><td>reserved</td></tr>
<tr><td>4-7</td><td>C</td><td>Cache-ability bits</td></tr>
<tr><td>8-10</td><td>G</td><td>granularity
<table>
<tr><th>G</th><th></th></tr>
<tr><td>0</td><td>byte accessible</td></tr>
<tr><td>1</td><td>wyde accessible</td></tr>
<tr><td>2</td><td>tetra accessible</td></tr>
<tr><td>3</td><td>octa accessible</td></tr>
</table>
</td></tr>
</table>

		4 — hexi accessible 5 to 7 — reserved	
11	~	reserved	
12-14	S	number of times to shift address to right and store for telescopic STPTR stores.	
16-23	T	device type (rom, dram, eeprom, I/O, etc)	
24-31	~	reserved	

Page Management Table - PMT

Overview

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

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

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

Location

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

PMTE Description

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

Page Management Table Entry

<table>
<tr><td>V</td><td>N</td><td>M</td><td colspan="2">$\sim_{9}$</td><td>C</td><td>E</td><td>AL_{2}</td><td>$\sim_{16}$</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.

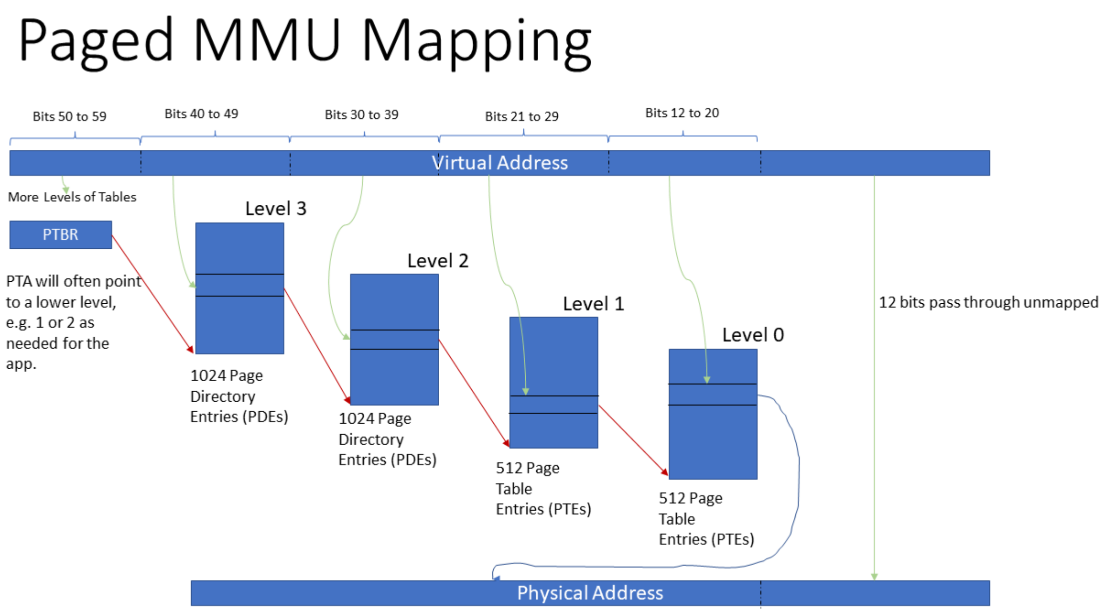
Page Tables

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

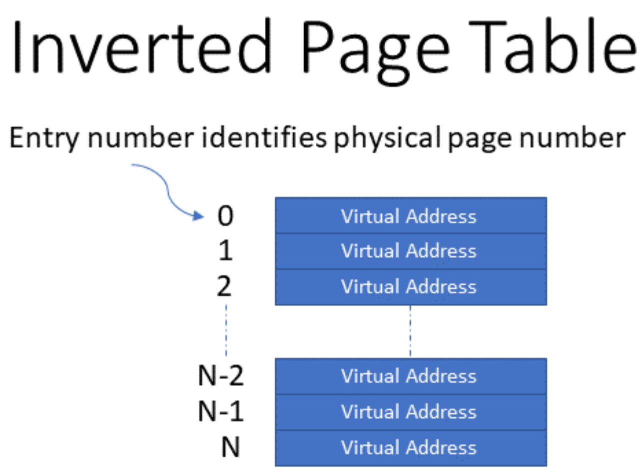


Inverted Page Tables

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

The Simple Inverted Page Table

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



Hashed Page Tables

Hashed Table Access

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

Clustered Hash Tables

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

Shared Memory

Another memory management issue to deal with is shared memory. Sometimes applications share memory with other apps for communication purposes, and to conserve memory space where there are common elements. The same shared library may be used by many apps running in the system. With a hierarchical paged memory management system, it is easy to share memory, just modify the page table entry to point to the same physical memory as is used by another process. With an inverted page table having only a single entry for each physical page is not sufficient to support shared memory. There needs to be multiple page table entries available for some physical pages but not others because multiple virtual addresses might map to the same physical address. One solution would be to have multiple buckets to store virtual addresses in for each physical address. However, this would waste a lot of memory because much of the time only a single mapped address is needed. There must be a better solution. Rather than reading off the table index as the physical page number, the association of the virtual and physical address can be stored. Since we now need to record the physical address multiple times the simple mechanism of using the table index as the physical page number cannot be used. Instead, the physical page number needs to be stored in the table in addition to the virtual page number.

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

Specifics: Thor2023 Page Tables

Thor2023 Hash Page Table Setup

Hash Page Table Entries - HPTE

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

<table>
<tr><td>V</td><td>LVL/BC$_5$</td><td>RGN$_3$</td><td>M</td><td>A</td><td>T</td><td>S</td><td>G</td><td>SW$_2$</td><td>CACHE$_4$</td><td>MRWX$_3$</td><td>HRWX$_3$</td><td>SRWX$_3$</td><td>URWX$_3$</td></tr>
<tr><td colspan="14">PPN$_{31..0}$</td></tr>
<tr><td colspan="14">PPN$_{63..32}$</td></tr>
<tr><td colspan="14">VPN$_{37..6}$</td></tr>
<tr><td colspan="14">VPN$_{69..38}$</td></tr>
<tr><td colspan="2">~$_4$</td><td colspan="6">ASID$_{11..0}$</td><td colspan="2">~$_2$</td><td colspan="4">VPN$_{83..70}$</td></tr>
</table>

Fields Description

V	1	translation Valid
G	1	global translation
RGN	3	region
PPN	64	Physical page number
VPN	84	Virtual page number
RWX	3	readable, writeable, executable
ASID	12	address space identifier
LVL/BC	5	bounce count
M	1	modified
A	1	accessed
T	1	PTE type (not used)
S	1	Shared page indicator
SW	3	OS usage

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

Small Hash Page Table Entries - SHPTE

The small HPTE is used for the test system which contains only 512MB of physical RAM to conserve hardware resources. The SHPTE is 72-bits in size. A 32-bit physical address is probably sufficient for this system. So, the physical page number could be 18-bits or less depending on the page size.

<table>
<tr><td>V</td><td>LVL/BC$_5$</td><td>RGN$_3$</td><td>M</td><td>A</td><td>T</td><td>S</td><td>G</td><td>SW</td><td>CACHE$_4$</td><td>ASID$_{3..0}$</td><td>HRWX$_3$</td><td>SRWX$_3$</td><td>URWX$_3$</td></tr>
<tr><td colspan="10">VPN$_{15..0}$</td><td colspan="4">PPN$_{15..0}$</td></tr>
<tr><td colspan="11"></td><td colspan="2">ASID$_{7..4}$</td><td>VPN$_{19..16}$</td></tr>
</table>

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 five HPTE entries. All entries in the group are searched in parallel for a match. Note that the entries are searched as the PTG is loaded, so that the PTG group load may be aborted early if a matching PTE is found before the load is finished.

191 0
PTE0
PTE1
PTE2
PTE3
PTE4

Small Page Table Group

For the small page table, a fetch size of 576 bits was chosen. This allows eight SHPTEs to fit into one group.

Size of Page Table

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

About 1/6 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 256k block RAM to store PTEs. 256k / 8 = 32768 entries. 32,768 / 4 = 8,192 physical pages. Since the RAM size is 512MB, each page would be 512MB/8,192 = 64kB. Since half the pages may be in secondary storage, 1GB of address range is available.

Since there are 32,768 entries in the table and they are grouped into groups of eight, there are 4,096 PTGs. To get to a page table group fast a hash function is needed then that returns a 12-bit number.

Reworking things with a 64kB page size and 32,768 PTEs. The maximum memory size that can be supported is: 2.0 GB. This is only 4x the amount of RAM in the system, but may be okay for demo purposes.

Hash Function

The hash function needs to reduce the size of a virtual address down to a 10-bit number. The asid should be considered part of the virtual address. Including the asid of 10-bits and a 32-bit address is 42 bits. The first thing to do is to throw away the lowest eighteen bits as they pass through the

MMU unaltered. We now have 24-bits to deal with. We can probably throw away some high order bits too, as a process is not likely to use the full 32-bit address range.

The hash function chosen uses the asid combined with virtual address bits 20 to 29. This should space out the PTEs according to the asid. Address bits 18 and 19 select one of four address ranges. the PTG supports seven PTEs. The translations where address bits 18 and 19 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 20 to 29.

Collision Handling

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

Finding a Match

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

Locality of Reference

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

Access Rights

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

Thor2023 Hierarchical Page Table Setup

Page Table Entries - PTE

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

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

Page Table Entry Format – PTE

V	LVL/BC_5	RGN_3	M	A	T	S	G	SW_2	$CACHE_4$	$MRWX_3$	$HRWX_3$	$SRWX_3$	$URWX_3$

$PPN_{31..0}$
$PPN_{63..32}$

Small Page Table Entry Format – SPTE

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

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

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

Super Pages

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

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

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

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

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

<table>
<tr><td>V</td><td>1_5</td><td>RGN_3</td><td>M</td><td>A</td><td>0</td><td>S</td><td>G</td><td>SW_2</td><td>$\sim_4$</td><td>$MRWX_3$</td><td>$HRWX_3$</td><td>$SRWX_3$</td><td>$URWX_3$</td></tr>
<tr><td colspan="11">PPN=$0x3FFFE0_{22}$</td><td colspan="3">NPG=$0x03F_{10}$</td></tr>
<tr><td colspan="14">PPN=$0xFFFFFFFF_{63..32}$</td></tr>
</table>

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

PTE Format for 16MB page

<table>
<tr><td>V</td><td>1_5</td><td>RGN_3</td><td>M</td><td>A</td><td>0</td><td>S</td><td>G</td><td>SW_2</td><td>$\sim_4$</td><td>$MRWX_3$</td><td>$HRWX_3$</td><td>$SRWX_3$</td><td>$URWX_3$</td></tr>
<tr><td colspan="11">$PPN_{31..10}$</td><td colspan="3">NPG_{10}</td></tr>
<tr><td colspan="14">$PPN_{63..32}$</td></tr>
</table>

PTE Format for 16GB page

<table>
<tr><td>V</td><td>2_5</td><td>RGN_3</td><td>M</td><td>A</td><td>0</td><td>S</td><td>G</td><td>SW_2</td><td>$\sim_4$</td><td>$MRWX_3$</td><td>$HRWX_3$</td><td>$SRWX_3$</td><td>$URWX_3$</td></tr>
<tr><td colspan="5">$PPN_{31..20}$</td><td colspan="9">NPG_{20}</td></tr>
<tr><td colspan="14">$PPN_{63..32}$</td></tr>
</table>

TLB – Translation Lookaside Buffer

Overview

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

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

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

Size / Organization

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

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

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

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

TLB Entries - TLBE

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

$Count_6$	LRU_3

<table>
<tr><td>V</td><td>LVL/BC_5</td><td>RGN_3</td><td>M</td><td>A</td><td>T</td><td>S</td><td>G</td><td>SW_2</td><td>$CACHE_4$</td><td>$MRWX_3$</td><td>$HRWX_3$</td><td>$SRWX_3$</td><td>$URWX_3$</td></tr>
<tr><td colspan="14">$PPN_{31..0}$</td></tr>
<tr><td colspan="14">$PPN_{63..32}$</td></tr>
</table>

<table>
<tr><td colspan="4">$VPN_{41..10}$</td></tr>
<tr><td colspan="4">$VPN_{73..42}$</td></tr>
<tr><td>$\sim_4$</td><td>$ASID_{11..0}$</td><td>$\sim_5$</td><td>$VPN_{83..73}$</td></tr>
</table>

Small TLB Entries - TLBE

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

$Count_6$	LRU_3

<table>
<tr><td>V</td><td>LVL/BC_5</td><td>RGN_3</td><td>M</td><td>A</td><td>T</td><td>S</td><td>G</td><td>SW_2</td><td>$CACHE_4$</td><td>$MRWX_3$</td><td>$HRWX_3$</td><td>$SRWX_3$</td><td>$URWX_3$</td></tr>
<tr><td colspan="2">$\sim_6$</td><td colspan="12">$PPN_{25..0}$</td></tr>
</table>

<table>
<tr><td colspan="5">$VPN_{41..10}$</td></tr>
<tr><td>$\sim_4$</td><td>$ASID_{11..0}$</td><td>PS</td><td>~</td><td>$VPN_{55..42}$</td></tr>
</table>

What is Translated?

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

Page Size

Because the TLB caches address translations it can get away with a much smaller page size than the page map can for a larger memory system. 4kB is a common size for many systems. There are some indications in contemporary documentation that a larger page size would be better. In this case the TLB uses 16kB. For a 512MB system (the size of the memory in the test system) there are 32768 16kB pages.

Ways

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

Management

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

Accessing the TLB

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

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

<table>
<tr><td>00</td><td colspan="11">TLBE ($PTE_{63..0}$)</td></tr>
<tr><td>08</td><td colspan="5"></td><td colspan="6">TLBE ($PTE_{95..64}$)</td></tr>
<tr><td>10</td><td colspan="11">TLBE ($VPN_{63..0}$)</td></tr>
<tr><td>18</td><td colspan="5"></td><td colspan="6">TLBE ($VPN_{95..64}$)</td></tr>
<tr><td>20</td><td colspan="11">TLB Miss $Address_{63..0}$</td></tr>
<tr><td>28</td><td>$\sim_4$</td><td colspan="2">Miss $ASID_{12}$</td><td colspan="2">$\sim_{16}$</td><td colspan="6">TLB Miss $Address_{95..64}$</td></tr>
<tr><td>30 to 68</td><td colspan="11"></td></tr>
<tr><td>70</td><td colspan="6"></td><td>AL_2</td><td>0</td><td>Entry Num_{10}</td><td>~</td><td>Way_4</td></tr>
<tr><td>78</td><td colspan="2">RWTRIG</td><td>WTRIG</td><td>RTRIG</td><td>$\sim_8$</td><td colspan="6">$\sim_{32}$</td></tr>
</table>

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

?RWX_3

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

$CACHE_4$

The $cache_4$ field is combined with the cache attributes specified in the region table. The region table takes precedence; however, if the $cache_4$ field indicates non-cache-ability then the data will not be cached.

Example TLB Update Routine

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

TLB Entry Replacement Policies

The TLB supports three algorithms for replacement of entries with new entries on a TLB miss. These are fixed replacement (0), least recently used replacement (1) and random replacement (2). The replacement method is stored in the AL_2 bits of the page table base register.

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

Flushing the TLB

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

Reset

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

Global Bit

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

Card Table

Overview

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

Organization

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

To improve the performance of scanning a hardware card table, HCT, is present which divides memory at an upper level into 8192-byte pages. The hardware card table indicates if a pointer store operation has taken place in one of the 8192-byte pages. It is then necessary to scan only cards representing the 8192-byte page rather than having to scan the entire 2MB card table. Note that this memory is organized as 2048 32-bit words. Allowing 32-bits at a time to be tested.

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

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

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

Location

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

Operation

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

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

Sample Write Barrier

```
        ; Milli-code routine for garbage collect write barrier.
        ; This sequence is short enough to be used in-line.
        ; Three level card memory.
        ; a2 is a register pointing to the card table.
        ; STPTR will cause an update of the master card table, and hardware card table.
        ;
GCWriteBarrier:
        STPTR       a0,[a1]             ; store the pointer value to memory at a1
        LSR         t0,a1,#8            ; compute card address
        STB         r0,[a2+t0]          ; clear byte in card memory
```

Instruction Set

Overview

Thor was a variable length instruction set with instructions varying in length from one to eight bytes. Thor2024 uses a fixed 40-bit instruction. However, instructions may be postfixed with immediates which are considered part of the instruction. Reducing the variety of instruction sizes makes implementation of decoders more economical. While instructions are 40-bits in length they are byte-aligned in memory. Program code may be relocated at any byte address.

Code Alignment

Program code may be relocated at any byte address.

Predicated Instruction Execution

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

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

Postfix Immediates

Immediate constants are supported via postfix immediates for most operands even if there is not an explicit immediate mode instruction. A postfix immediate is specified using the register-register form of the instruction. If all bits of the register specifier are set then the instruction should use a postfix immediate in place of that register value.

The following example shows an instruction using a 64-bit postfix immediates. Two postfix immediate buckets are required. These will be treated as NOPs when encountered in the instruction stream.

MULSU Rt, Ra, Imm

39 37	36 34	33 27	26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	21_7	0	0	18_6	63_6	Rt_6	2_7
$Immediate_{31..0}$							0	124_7
$Immediate_{63..32}$							0	124_7

A postfix immediate may also be used with an immediate mode instruction to use an extended immediate value.

ADDI Rt, Ra, Imm

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$\sim_{16}$	Ra_6	Rt_6	4_7
$Immediate_{31..0}$				0	124_7

Operation Size

Most instructions support multiple sizes for operands. The operation size is specified by the Sz_2 field of the instruction.

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	4_7	Sz_2	Rb_6	Ra_6	Rt_6	2_7

Sz_2	Size
0	Byte
1	Wyde-byte
2	Tetra-byte
3	Octa-byte

R2Q Group

Sz_2	Size
0	Hexi-Byte
1	reserved
2	reserved
3	reserved

Instruction Descriptions

Arithmetic Operations

Representations

long

127 0
128 bits

int

63 0
64 bits

short

31 0
32 bits

char

15 0
16 bits

byte

7 0
8 bits

decimal

127 120	119 0
	120 bits

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

Arithmetic Operations

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

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

Instruction Format: RI

<table>
<tr><td>39 38</td><td>37 35</td><td>34 19</td><td>18 13</td><td>12 7</td><td>6 0</td></tr>
<tr><td>Fmt$_2$</td><td>Pr$_3$</td><td>Immediate$_{15..0}$</td><td>Ra$_6$</td><td>Rt$_6$</td><td>4$_7$</td></tr>
</table>

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

MULSU Rt, Ra, Imm

<table>
<tr><td>39 37</td><td>36 34</td><td>33 27</td><td>26</td><td>25</td><td>24 19</td><td>18 13</td><td>12 7</td><td>6 0</td></tr>
<tr><td>Fmt$_3$</td><td>Pr$_3$</td><td>21$_7$</td><td>0</td><td>0</td><td>63$_6$</td><td>Ra$_6$</td><td>Rt$_6$</td><td>2$_7$</td></tr>
<tr><td colspan="7">Immediate$_{32}$</td><td>0</td><td>124$_7$</td></tr>
</table>

There are both signed and unsigned versions of the arithmetic operations. However, note there is no signed or unsigned compare operation as a single compare instruction produces results for both signed and unsigned comparisons.

ABS – Absolute Value

Description:

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

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	Sz_2	3_6	Ra_6	Rt_6	1_7

Operation:

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

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

ADD - Register-Register

Description:

Add two registers and place the sum in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	4_7	Sz_2	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra + Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ADDI - Add Immediate

Description:

Add a register and immediate value and place the sum in the target register. The immediate is sign extended to the machine width.

Instruction Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	4_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + immediate

Exceptions:

Notes:

AND – Bitwise And

Description:

Bitwise and two registers and place the result in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0_7	Sz_2	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra & Rb

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ANDC – Bitwise And Complement

Description:

Bitwise and a source register and the complement of a second source register and place the result in the target register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	11_7	Sz_2	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra & ~Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ANDI - Add Immediate

Description:

Add a register and immediate value and place the sum in the target register. The immediate is one extended to the machine width.

Instruction Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	8_7

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra + immediate

Exceptions:

Notes:

BMAP – Byte Map

Description:

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

Instruction Format: R2

BMAP Rt, Ra, Rb

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt$_3$	Pr$_3$	35$_7$	~$_2$	Rb$_6$	Ra$_6$	Rt$_6$	2$_7$

BMAP Rt, Ra, Imm64

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt$_3$	Pr$_3$	35$_7$	~$_2$	63$_6$	Ra$_6$	Rt$_6$	2$_7$
Immediate$_{31..0}$						0	124$_7$
Immediate$_{63..32}$						0	124$_7$

Operation:

Vector Operation

Execution Units: First Integer ALU

Exceptions: none

Notes:

BMM – Bit Matrix Multiply

BMM Rt, Ra, Rb

Description:

The BMM instruction treats the bits of register Ra and register Rb as an 8x8 matrix and performs a bit matrix multiply of the two registers and stores the result in the target register. An alternate mnemonic for this instruction is MOR.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	34_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Operation:

for I = 0 to 7
    for j = 0 to 7
        Rt.bit[i][j] = (Ra[i][0]&Rb[0][j]) | (Ra[i][1]&Rb[1][j]) | … | (Ra[i][7]&Rb[7][j])

Clock Cycles: 1

Execution Units: First Integer ALU

Exceptions: none

Notes:

The bits are numbered with bit 63 of a register representing I,j = 0,0 and bit 0 of the register representing I,j = 7,7.

CHARNDX – 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 character. The index result may vary from -1 to +15. The index of the first found byte is returned (closest to zero). The result is -1 if the character could not be found.

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

Instruction Format: R2 (byte)

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	37_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

Instruction Format: R2 (wyde)

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	38_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

Instruction Format: R2 (tetra)

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	39_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU’s

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 Ra and Rb then an exception will occur.

Instruction Format: R2

39 37	36 34	33 31	30 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	Op_3	$\sim_4$	$\sim_2$	Rb_6	Ra_6	Rt_6	12_7

Op_3	exception when not
0	Rs >= Ra and Rs < Rb
1	Rs >= Ra and Rs <= Rb
2	Rs > Ra and Rs < Rb
3	Rs > Ra and Rs <= Rb
4	Not (Rs >= Ra and Rs < Rb)
5	Not (Rs >= Ra and Rs <= Rb)
6	Not (Rs > Ra and Rs < Rb)
7	Not (Rs > Ra and Rs <= Rb)

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:

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	70_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

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

CMOVNZ – Conditional Move if Non-Zero

Description:

If Ra is non-zero then the target register is set to Rb, otherwise the target register is to Rc.

Instruction Format: R3

CMOVNZ Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	4_3	Rc_6	Rb_6	Ra_6	Rt_6	104_7

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
If Ra then
    Rt = Rb
else
    Rt = Rc
```

Exceptions: none

CMOVZ – Conditional Move if Zero

Description:

If Ra is zero then the target register is set to Rb, otherwise the target register is to Rc.

Instruction Format: BITFLD

CMOVZ Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	5_3	Rc_6	Rb_6	Ra_6	Rt_6	104_7

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
If Ra = 0 then
        Rt = Rb
else
        Rt = Rc
```

Exceptions: none

CMP - Comparison

Description:

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

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 Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	3_7	Sz_2	Rb_6	Ra_6	Rt_6	2_7

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

CMPI – Compare Immediate

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.

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 Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	11_7

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

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.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	Sz_2	0_6	Ra_6	Rt_6	1_7

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTLO – Count Leading Ones

Description:

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

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	Sz_2	1_6	Ra_6	Rt_6	1_7

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.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	Sz_2	2_6	Ra_6	Rt_6	1_7

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CNTTZ – Count Trailing Zeros

Description:

This instruction counts the number of consecutive zero bits beginning at the least significant bit towards the most significant bit. This instruction can also be used to get the position of the first one bit from the right-hand side.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	Sz_2	6_6	Ra_6	Rt_6	1_7

Operation:

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

CPUID – Get CPU Info

Description:

This instruction returns general information about the core. The sum of Rb and register Ra is used as a table index to determine which row of information to return.

Supported Operand Sizes: N/A

Instruction Formats: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	7_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Info[Ra+Rb]

Exceptions: none

Index	bits	Information Returned
0	0 to 127	The processor core identification number. This field is determined from an external input. It would be hard wired to the number of the core in a multi-core system.
2	0 to 127	Manufacturer name first sixteen chars “Finitron”
3	0 to 127	Manufacturer name last sixteen characters
4	0 to 127	CPU class “64BitSS”
5	0 to 127	CPU class
6	0 to 127	CPU Name “Thor2024”
7	0 to 127	CPU Name
8	0 to 127	Model Number “M1”
9	0 to 127	Serial Number “1234”
10	0 to 127	Features bitmap
11	0 to 31	Instruction Cache Size (32kB)
11	32 to 63	Data cache size (64kB)
12	0 to 7	Maximum vector length

CSR – Control and Special Registers Operations

Description:

Perform an operation on a CSR. The previous value of the CSR is placed in the target register.

Operation	Op_3	Mnemonic
Read CSR	0	CSRRD
Write CSR	1	CSRRW
Or to CSR (set bits)	2	CSRRS
And complement to CSR (clear bits)	3	CSRRC

Supported Operand Sizes: N/A

Instruction Formats: CSR

39 38	37 35	3433	32 19	18 13	12 7	6 0
Fmt_2	Pr_3	Op_2	$Regno_{13..0}$	Ra_6	Rt_6	7_7

DIV – 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.

Operation:

Rt = Ra / Rb

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	17_7	Sz_2	Rb_6	Ra_6	Rt_6	2_7

Size	Clocks
Byte	6
Wyde-byte	10
Tetra-byte	18
Octa-byte	34
Hexi-byte	66

Execution Units: All Integer ALU’s

Exceptions: DBZ

Notes:

DIVI – Signed Immediate 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.

Operation:

Rt = Ra / Imm

Instruction Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	13_7

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

DIVU – Unsigned Division

Description:

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

Operation:

Rt = Ra / Rb

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	20_7	Sz_2	Rb_6	Ra_6	Rt_6	2_7

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

DIVUI – Unsigned Immediate Division

Description:

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

Operation:

Rt = Ra / Imm

Instruction Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	21_7

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ENOR – Bitwise Exclusive Nor

Description:

Bitwise exclusively nor two registers and place the result in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	10_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra ^ Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

EOR – Bitwise Exclusive Or

Description:

Bitwise exclusively or two registers and place the result in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	2_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra ^ Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

EORI – Exclusive Or Immediate

Description:

Exclusive Or a register and immediate value and place the sum in the target register. The immediate is zero extended to the machine width.

Instruction Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	10_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + immediate

Exceptions:

Notes:

MAX – Maximum Value

Description:

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

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	33_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Execution Units: ALU #0 only

Operation:

```
IF (Ra > Rb)
        Rt = Ra
else
        Rt = Rb
```

MAX3 – Maximum Value

Description:

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

Instruction Format: R3

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	2_3	Rc_6	Rb_6	Ra_6	Rt_6	104_7

Execution Units: ALU #0 only

Operation:

IF (Ra > Rb and Ra > Rc)
    Rt = Ra
Else if (Rb > Rc)
    Rt = Rb
Else
    Rt = Rc

MIN – Minimum Value

Description:

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

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	32_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Operation:

IF (Ra < Rb)
    Rt = Ra
else
    Rt = Rb

MIN3 – Minimum Value

Description:

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

Instruction Format: R3

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	3_3	Rc_6	Rb_6	Ra_6	Rt_6	104_7

Execution Units: ALU #0 only

Operation:

IF (Ra < Rb and Ra < Rc)
    Rt = Ra
Else if (Rb < Rc)
    Rt = Rb
Else
    Rt = Rc

MOV – Move Register to Register

Description:

Move register-to-register. This instruction may move between different types of registers. Raw binary data is moved. No data conversions are applied.

Instruction Format: R2

39 37	36 34	33 21	22 21	20 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_{13}$	Ra_2	Rt_2	Ra_5	Rt_6	15_7

Operation: R2

Rt = Ra

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Ra_8 / Rt_8	Register file
0 to 63	General purpose registers 0 to 63
64 to 127	Vector registers
128	Vector Length in elements
129	Vector Length in bytes
130	VRM – restart mask
131	VERR – error record
132	VED – element descriptor

MOVSXB – Move, Sign Extend Byte

Description:

A byte is extracted from the source operand, sign extended, and the result placed in the target register.

Operation:

Instruction Format: BITFLD

MOVSXB Rt, Ra

39 37	36 34	33	32 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	1	7_7	0_7	Ra_6	Rt_6	19_7

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

MOVSXT – Move, Sign Extend Tetra

Description:

A tetra is extracted from the source operand, sign extended, and the result placed in the target register.

Operation:

Instruction Format: BITFLD

MOVSXT Rt, Ra

39 37	36 34	33	32 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	1	31_7	0_7	Ra_6	Rt_6	19_7

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MOVSXW – Move, Sign Extend Wyde

Description:

A wyde is extracted from the source operand, sign extended, and the result placed in the target register.

Operation:

Instruction Format: BITFLD

MOVSXW Rt, Ra

39 37	36 34	33	32 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	1	15_7	0_7	Ra_6	Rt_6	19_7

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MUL – Multiply Register-Register

Description:

Multiply two registers and place the product in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers. Values are treated as signed integers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	16_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Size	Clocks
Byte	1
Wyde-byte	1
Tetra-byte	1
Octa-byte	4
Hexi-byte	4

Operation: R2

Rt = Ra * Rb

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

MULH – Multiply High

Description:

Compute the high order product of two values. Both operands must be in registers. Both the operands are treated as signed values, the result is a signed result.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	24_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Exceptions: none

Execution Units: ALU

Operation

Rt = Ra * Rb

Vector Operation

for x = 0 to VL - 1

if (Pr[x] & VGM[x]) Vt[x] = Va[x] * Vb[x]

else Vt[x] = Vt[x]

Exceptions: none

MULI - Multiply Immediate

Description:

Multiply a register and immediate value and place the product in the target register. The immediate is sign extended to the machine width. Values are treated as signed integers.

Instruction Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	6_7

Clock Cycles: 4

Execution Units: All ALU’s

Operation:

Rt = Ra * immediate

Exceptions:

Notes:

MULSU – Multiply Signed Unsigned

Description:

Multiply two registers and place the product in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers. The first operand is signed, the second unsigned.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	21_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra * Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

MULTSUH – Multiply Signed Unsigned High

Description:

Multiply two registers and place the high order product bits in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers. The first operand is signed, the second unsigned.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	29_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra * Rb

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

MULU – Unsigned Multiply Register-Register

Description:

Multiply two registers and place the product in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers. Values are treated as unsigned integers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	19_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra * Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

MULUH – Unsigned Multiply High

Description:

Multiply two registers and place the high order product bits in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers. Values are treated as unsigned integers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	27_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra * Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

MULUI - Multiply Unsigned Immediate

Description:

Multiply a register and immediate value and place the product in the target register. The immediate is sign extended to the machine width. Values are treated as unsigned integers.

Instruction Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	14_7

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra + immediate

Exceptions:

Notes:

MUX – Multiplex

Description:

If a bit in Ra is set then the bit of the target register is set to the corresponding bit in Rb, otherwise the bit in the target register is set to the corresponding bit in Rc.

Instruction Format: BITFLD

MUX Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0_3	Rc_6	Rb_6	Ra_6	Rt_6	104_7

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

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

Exceptions: none

NAND – Bitwise Nand

Description:

Bitwise nand two registers and place the result in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	8_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = ~(Ra & Rb)

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

NOR – Bitwise Or

Description:

Bitwise nor two registers and place the result in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	9_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = ~(Ra | Rb)

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

OR – Bitwise Or

Description:

Bitwise or two registers and place the result in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra | Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

ORC – Bitwise Or Complement

Description:

Bitwise 'or' a source register and the complement of a second source register and place the result in the target register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	12_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra | ~Rb

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

ORI - Or Immediate

Description:

Or a register and immediate value and place the sum in the target register. The immediate is zero extended to the machine width.

Instruction Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	9_7

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + immediate

Exceptions:

Notes:

PFX – Constant Postfix

Description:

The PFX instruction postfix is used to provide large constants for use in the preceding instruction as the immediate constant for the instruction. The constant postfix may override the second source operand of most instructions.

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

There may be multiple postfix buckets following an instruction to provide constants up to 128-bits.

Instruction Format: PFX

This format provides a thirty-two-bit constant bucket.

39 8	7	6 0
$Immediate_{32}$	0	124_7

PTRDIF – Difference Between Pointers

Description:

Subtract two values then shift the result right. Both operands must be in a register. The right shift is provided to accommodate common object sizes. It may still be necessary to perform a divide operation after the PTRDIF to obtain an index into odd sized or large objects. Sc may vary from zero to fifteen.

Instruction Format: BITFLD

PTRDIF Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_3	Rc_6	Rb_6	Ra_6	Rt_6	104_7

Operation:

Rt = Abs(Ra – Rb) >> $Rc_{[3:0]}$

Clock Cycles: 1

Execution Units: Integer

Exceptions:

None

REVBIT – Reverse Bit Order

Description:

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

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	5_6	Ra_6	Rt_6	1_7

Operation:

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

SEQ – Set if Equal

Description:

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

Operation:

Prt = Ra == Rb or Prt = Ra == Imm

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: R2

SEQ Rt, Ra, Rb

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	80_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

SLE – Set if Less Than or Equal

Description:

Compare two source operands for signed less than or equal and place the result in the target predicate register. The result is a Boolean true or false. This instruction may also test for greater than or equal by swapping operands.

Operation:

If (S)

Prt = Rb <= Ra or Prt = Imm <= Ra

Else

Prt = Ra <= Rb or Prt = Ra <= Imm

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: R2

SLE Rt, Ra, Rb

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	83_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

SLEU – Set if Unsigned Less Than or Equal

Description:

Compare two source operands for unsigned less than or equal and place the result in the target register. The result is a Boolean true or false. This instruction may also test for greater than or equal by swapping operands.

Operation:

Rt = Ra <= Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Format: R2

SLEU Rt, Ra, Rb

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	85_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

SLT – Set if Less Than

Description:

Compare two source operands for signed less than and place the result in the target predicate register. The result is a Boolean true or false. This instruction may also test for greater than by swapping operands.

Operation:

Prt = Ra < Rb or Prt = Ra < Imm

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: R2

SLT Rt, Ra, Rb

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	82_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

SLTI – Set if Less Than Immediate

Description:

Compare two source operands for signed less than and place the result in the target predicate register. The result is a Boolean true or false. This instruction may also test for greater than by swapping operands.

Operation:

Rt = Ra < Imm

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Format: RI

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

39 38	37 35	34 19	18 13	12 7	6 0
Fmt$_2$	Pr$_3$	Immediate$_{15..0}$	Ra$_6$	Rt$_6$	3$_7$

SLTU – Set if Unsigned Less Than

Description:

Compare two source operands for unsigned less than and place the result in the target register. The result is a Boolean true or false. This instruction may also test for greater than by swapping operands.

Operation:

Prt = Ra < Rb or Rt = Ra < Imm

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Format: R2

SLTU Rt, Ra, Rb

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	84_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

SNE – Set if Not Equal

Description:

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

Operation:

Prt = Ra != Rb or Prt = Ra != Imm

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: R2

SNE Rt, Ra, Rb

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	81_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

SQRT – Square Root

Description:

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

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	4_6	Ra_6	Rt_6	1_7

Operation:

Rt = SQRT(Ra)

Execution Units: Integer ALU #0

Clock Cycles: 1

Exceptions: none

Notes:

SUB – Subtract Register-Register

Description:

Subtract two registers and place the difference in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. All registers are integer registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	5_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

Operation: R2

Rt = Ra - Rb

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

SUBFI – Subtract from Immediate

Description:

Subtract a register from an immediate value and place the difference in the target register. The immediate is sign extended to the machine width.

Instruction Format: RI

39 38	37 35	34 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{15..0}$	Ra_6	Rt_6	5_7

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = immediate - Ra

Exceptions:

Notes:

WYDENDX – Wyde Index

Description:

This instruction searches Ra, which is treated as an array of wydes, for a wyde value specified by Rb and places the index of the wyde into the target register Rt. If the wyde is not found -1 is placed in the target register. A common use would be to search for a null. The index result may vary from -1 to +7. The index of the first found wyde is returned (closest to zero).

Supported Operand Sizes: .b, .t

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	38_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

Operation:

Rt = Index of (Rb in Ra)

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Shift and Rotate Operations

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

Thor2024 has a full complement of shift instructions including rotates.

ASL –Arithmetic Shift Left

Description:

Left shift an operand value by an operand value and place the result in the target register. The 'B' field of the instruction is shifted into the least significant bits. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rb field of the instruction, or an immediate value.

Instruction Format: R2

Instruction Format: R2

39 37	36 34	33	32	31 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	0	0_6	~	Rb_6	Ra_6	Rt_6	88_7

Operation:

Rt = Ra << Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ASLI –Arithmetic Shift Left

Description:

Left shift an operand value by an operand value and place the result in the target register. The 'B' field of the instruction is shifted into the least significant bits. The first operand must be in a register specified by the Ra. The second operand is an immediate value.

Instruction Format: RI7

39 37	36 34	33	32	31 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	1	0_6	Imm_7	Ra_6	Rt_6	88_7

Operation:

Rt = Ra << Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ASR –Arithmetic Shift Right

Description:

Right shift an operand value by an operand value and place the result in the target register. The sign bit is shifted into the most significant bits. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rb field of the instruction, or an immediate value.

Instruction Format: R2

39 37	36 34	33	32	31 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	0	2_6	~	Rb_6	Ra_6	Rt_6	88_7

Operation:

Rt = Ra << Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ASRI –Arithmetic Shift Right

Description:

Right shift an operand value by an operand value and place the result in the target register. The sign bit is shifted into the most significant bits. The first operand must be in a register specified by the Ra. The second operand is an immediate value.

Instruction Format: RI7

39 37	36 34	33	32	31 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	1	2_6	Imm_7	Ra_6	Rt_6	88_7

Operation:

Rt = Ra >> Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

LSR –Logic Shift Right

Description:

Right shift an operand value by an operand value and place the result in the target register. The 'B' field of the instruction is shifted into the most significant bits. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rb field of the instruction, or an immediate value.

Instruction Format: R2

39 37	36 34	33	32	31 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	0	1_6	~	Rb_6	Ra_6	Rt_6	88_7

Operation:

Rt = Ra >> Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

LSRI –Logical Shift Right

Description:

Right shift an operand value by an operand value and place the result in the target register. The ‘B’ field of the instruction is shifted into the most significant bits. The first operand must be in a register specified by the Ra. The second operand is an immediate value.

Instruction Format: RI7

39 37	36 34	33	32	31 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	1	1_6	Imm_7	Ra_6	Rt_6	88_7

Operation:

Rt = Ra >> Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ROL –Rotate Left

Description:

Rotate left an operand value by an operand value and place the result in the target register. The most significant bits are shifted into the least significant bits. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rb field of the instruction, or an immediate value.

Instruction Format: R2

39 37	36 34	33	32	31 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	0	3_6	~	Rb_6	Ra_6	Rt_6	88_7

Operation:

Rt = Ra << Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ROLI –Rotate Left by Immediate

Description:

Rotate left shift an operand value by an operand value and place the result in the target register. The most significant bits are shifted into the least significant bits. The first operand must be in a register specified by the Ra. The second operand is an immediate value.

Instruction Format: RI7

39 37	36 34	33	32	31 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	1	3_6	Imm_7	Ra_6	Rt_6	88_7

Operation:

Rt = Ra << Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

ROR –Rotate Right

Description:

Rotate right an operand value by an operand value and place the result in the target register. The least significant bits are shifted into the most significant bits. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rb field of the instruction, or an immediate value.

Instruction Format: R2

39 37	36 34	33	32	31 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	0	4_6	~	Rb_6	Ra_6	Rt_6	88_7

Operation:

Rt = Ra >> Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

RORI –Rotate Right by Immediate

Description:

Rotate right an operand value by an operand value and place the result in the target register. The least significant bits are shifted into the most significant bits. The first operand must be in a register specified by the Ra. The second operand is an immediate value.

Instruction Format: RI7

39 37	36 34	33	32	31 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	B	1	4_6	Imm_7	Ra_6	Rt_6	88_7

Operation:

Rt = Ra >> Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

Bit-field Manipulation Operations

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

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

General Format of Bitfield Instructions

Bitfield instructions are 48-bits in length to accommodate register and immediate constants.

CLR Rt, Ra, Rb, Rc

47 45	44 41	40 38	37	36	35 34	33 32	30 24	23 17	16 12	11 7	6 0
Fmt_3	Pr_4	Fn_3	Ci	Bi	Ot_2	$\sim_2$	Me_7	Mb_7	Ra_5	Rt_5	93_7

Mb_7 may be either a register spec or a seven-bit immediate constant specifying the start position of the bitfield.

Me_7 may be either a register spec or a seven-bit immediate constant specifying the end position of the bitfield.

The Ci field indicates (1) to use either an immediate constant, or (0) to use a register for the third source operand.

The Bi field indicates (1) to use either an immediate constant, or (0) to use a register for the second source operand.

The Ot_2 field determines the operand type for operands Ra and Rt.

Ot_2	Ra, Rt
0	Integer
1	Float
2	predicate
3	reserved

CLR – Clear Bit Field

Description:

A bit field in the source operand is cleared and the result placed in the target register.

Operation:

Instruction Format: BITFLD

CLR Rt, Ra, Rb, Rc

39 37	36 34	33	32 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0	$\sim_2$	Rc_6	Rb_6	Ra_6	Rt_6	16_7

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

CLRI – Clear Bit Field

Description:

A bit field in the source operand is cleared and the result placed in the target register.

Operation:

Instruction Format: BITFLD

CLR Rt, Ra, Imm, Imm

39 37	36 34	33	32 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	1	Me_7	Mb_7	Ra_6	Rt_6	16_7

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

COM – Complement Bit Field

Description:

A bit field in the source operand is one’s complemented and the result placed in the target register.

Operation:

Instruction Format: BITFLD

COM Rt, Ra, Rb, Rc

39 37	36 34	33	32 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0	$\sim_2$	Rc_6	Rb_6	Ra_6	Rt_6	20_7

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

COMI – Complement Bit Field

Description:

A bit field in the source operand is one’s complemented and the result placed in the target register.

Operation:

Instruction Format: BITFLD

COM Rt, Ra, Imm, Imm

39 37	36 34	33	32 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	1	Me_7	Mb_7	Ra_6	Rt_6	20_7

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

DEP – Deposit Bit Field

Description:

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

Supported Operand Sizes: N/A

Operation:

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

Instruction Formats:

DEP Rt, Ra, Rb, Rc

39 37	36 34	33	32 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0	$\sim_2$	Rc_6	Rb_6	Ra_6	Rt_6	23_7

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Notes:

DEPI – Deposit Bit Field

Description:

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

Supported Operand Sizes: N/A

Operation:

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

Instruction Formats:

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

39 37	36 34	33	32 26	25 19	18 13	12 7	6 0
Fmt$_3$	Pr$_3$	1	Me$_7$	Mb$_7$	Ra$_6$	Rt$_6$	23$_7$

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

EXT – Extract Bit Field

Description:

A bit field is extracted from the source operand, sign extended, and the result placed in the target register.

Operation:

Instruction Format: BITFLD

EXT Rt, Ra, Rb, Rc

Instruction Format: R2

39 37	36 34	33	32 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0	$\sim_2$	Rc_6	Rb_6	Ra_6	Rt_6	19_7

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

EXTI – Extract Bit Field

Description:

A bit field is extracted from the source operand, sign extended, and the result placed in the target register.

Instruction Format: BITFLD

EXT Rt, Ra, Imm$_7$, Imm$_7$

39 37	36 34	33	32 26	25 19	18 13	12 7	6 0
Fmt$_3$	Pr$_3$	1	Me$_7$	Mb$_7$	Ra$_6$	Rt$_6$	19$_7$

Clock Cycles: 1

Operation:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

EXTU – Extract Unsigned Bit Field

Description:

A bit field is extracted from the source operand, zero extended, and the result placed in the target register.

Operation:

Instruction Format: BITFLD

EXTU Rt, Ra, Rb, Rc

39 37	36 34	33	32 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0	$\sim_2$	Rc_6	Rb_6	Ra_6	Rt_6	18_7

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

EXTUI – Extract Unsigned Bit Field

Description:

A bit field is extracted from the source operand, zero extended, and the result placed in the target register.

Operation:

Instruction Format: BITFLD

EXTU Rt, Ra, Imm_7, Imm_7

39 37	36 34	33	32 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	1	Me_7	Mb_7	Ra_6	Rt_6	18_7

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

SET – Set Bit Field

Description:

A bit field in the source operand is set to all ones and the result placed in the target register.

Operation:

Instruction Format: BITFLD

SET Rt, Ra, Rb, Rc

39 37	36 34	33	32 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0	$\sim_2$	Rc_6	Rb_6	Ra_6	Rt_6	17_7

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

SETI – Set Bit Field

Description:

A bit field in the source operand is set to all ones and the result placed in the target register.

Operation:

Instruction Format: BITFLD

SET Rt, Ra, Imm, Imm

39 37	36 34	33	32 26	25 19	18 13	12 7	6 0
Fmt_3	Pr_3	1	Me_7	Mb_7	Ra_6	Rt_6	17_7

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Cryptographic Accelerator Instructions

AES64DS – Final Round Decryption

Description:

Perform the final round of decryption for the AES standard. Register Ra represents the entire AES state.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	18_6	Ra_6	Rt_6	1_7

Operation:

Exceptions: none

AES64DSM – Middle Round Decryption

Description:

Perform a middle round of decryption for the AES standard. Register Ra represents the entire AES state.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	19_6	Ra_6	Rt_6	1_7

Operation:

Exceptions: none

AES64ES – Final Round Encryption

Description:

Perform the final round of encryption for the AES standard. Register Ra represents the entire AES state.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	20_6	Ra_6	Rt_6	1_7

Operation:

Exceptions: none

AES64ESM – Middle Round Encryption

Description:

Perform a middle round of encryption for the AES standard. Register Ra represents the entire AES state.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	21_6	Ra_6	Rt_6	1_7

Operation:

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: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	24_6	Ra_6	Rt_6	1_7

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: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	25_6	Ra_6	Rt_6	1_7

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: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	26_6	Ra_6	Rt_6	1_7

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: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	27_6	Ra_6	Rt_6	1_7

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

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	28_6	Ra_6	Rt_6	1_7

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

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	29_6	Ra_6	Rt_6	1_7

Operation:

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

Execution Units: ALU #0

Exceptions: none

SHA512SUM0

Description:

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	30_6	Ra_6	Rt_6	1_7

SHA512SUM1

Description:

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	31_6	Ra_6	Rt_6	1_7

SM3P0

Description:

P0 transform of SM3 hash function.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	14_6	Ra_6	Rt_6	1_7

Operation

Rt = Ra ^ rol(Ra,9) ^ rol(Ra,17)

SM3P1

Description:

P1 transform of SM3 hash function.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	15_6	Ra_6	Rt_6	1_7

Operation

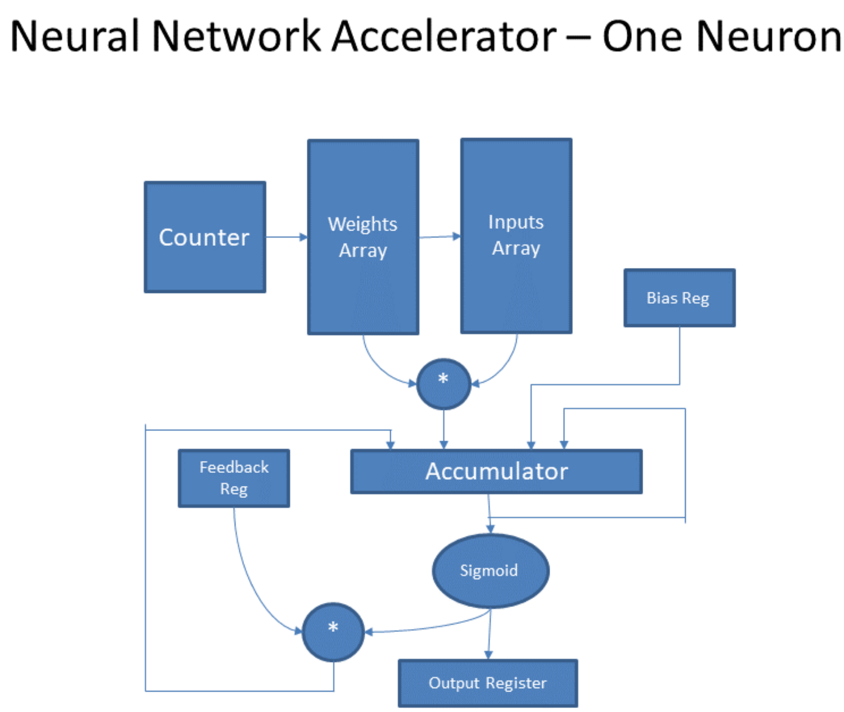
Rt = Ra ^ rol(Ra,15) ^ rol(Ra,23)

Neural Network Accelerator Instructions

Overview

Included in the ISA are instructions for neural network acceleration. Each neuron is composed of an accumulator that sums the product of weights and inputs and an output activation function. Neurons may be biased with a bias value and may also have feedback from output to input via a feedback constant. The neurons are implemented using 16.16 fixed-point arithmetic. There are 8 neurons in a single layer which may calculate simultaneously. Following is a sketch of the NNA organization. Note that multi-layer networks and additional neurons may be implemented by appropriate software modification of the NNA. The weights and input arrays have a depth of 1024 entries. Not all entries need be used. The number of entries in use is configurable programmatically with the base count and maximum count register using the NNA_MTBC and NNA_MTMC instruction.

Several of the NNA instructions allow multiple neurons to be updated at the same time by representing the neuron update list as a bitmask.



NNA_MFACT – Move from Output Activation

Description:

Move from activation output register. Move a value from the neuron's activation register output to the target register Rt. Bits 0 to 3 of Ra specify the neuron.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	10_6	Ra_6	Rt_6	1_7

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTBC – Move to Base Count

Description:

Move to base count register. Move the value in Ra to the base count register for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the base count value.

The neuron calculates the activation output using weight and input array entries between the base count and maximum count inclusive.

Manipulating the base count and maximum count registers ease the implementation of multi-layer networks that do not require the use of all array entries.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt$_3$	Pr$_3$	45$_7$	Im$_2$	Rb$_6$	Ra$_6$	Rt$_6$	2$_7$

Im$_2$	Meaning
00	Both Ra and Rb are register values
01	Ra specifies an immediate, Rb is a register value
10	Ra is a register value, Rb specifies an immediate
11	reserved

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTBIAS – Move to Bias

Description:

Move to bias value. Move the value in Ra to the bias register for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the bias value.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	42_7	Im_2	Rb_6	Ra_6	Rt_6	2_7

Im_2	Meaning
00	Both Ra and Rb are register values
01	Ra specifies an immediate, Rb is a register value
10	Ra is a register value, Rb specifies an immediate
11	reserved

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTFB – Move to Feedback

Description:

Move to feedback constant. Move the value in Ra to the feedback constant for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the feedback constant.

The feedback constant acts to create feedback in the neuron by multiplying the output activation level by the feedback constant and using the result as an input. If no feedback is desired then this constant should be set to zero.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	43_7	Im_2	Rb_6	Ra_6	$\sim_6$	2_7

Im_2	Meaning
00	Both Ra and Rb are register values
01	Ra specifies an immediate, Rb is a register value
10	Ra is a register value, Rb specifies an immediate
11	reserved

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTIN – Move to Input

Description:

Move to input array. Move the value in Ra to the input memory cell identified with Rb. Bits 0 to 15 of Rb specify the memory cell address, bits 32 to 63 of Rb are a bit mask specifying the neurons to update. Bits 0 to 15 of Rb are incremented and stored in Rt.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	41_7	Im_2	Rb_6	Ra_6	$\sim_6$	2_7

Im_2	Meaning
00	Both Ra and Rb are register values
01	Ra specifies an immediate, Rb is a register value
10	Ra is a register value, Rb specifies an immediate
11	reserved

Clock Cycles: 1

Execution Units: NNA

Notes:

Multiple neurons may have their inputs updated at the same time with the same value. All the neurons may have the same inputs but the weights for the individual neurons would be different so that a pattern may be recognized.

NNA_MTMC – Move to Max Count

Description:

Move to maximum count register. Move the value in Ra to the maximum count register for the neurons identified with a bitmask in Rb. Each bit of Rb represents a neuron. Multiple neurons may be initialized at the same time. Ra contains the maximum count value.

The maximum count is the upper limit of inputs and weights to use in the calculation of the activation function. The maximum count should not exceed the hardware table size. The table size is 1024 entries.

The neuron calculates the activation output using weight and input array entries between the base count and maximum count inclusive.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	44_7	Im_2	Rb_6	Ra_6	$\sim_6$	2_7

Im_2	Meaning
00	Both Ra and Rb are register values
01	Ra specifies an immediate, Rb is a register value
10	Ra is a register value, Rb specifies an immediate
11	reserved

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_MTWT – Move to Weights

Description:

Move to weights array. Move the value in Ra to the weight memory cell identified with Rb. Bits 0 to 15 or Rb specify the memory cell address, bits 32 to 63 of Rb are a bit mask specifying the neurons to update. Bits 0 to 15 of Rb are incremented and stored in Rt.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt$_3$	Pr$_3$	40$_7$	Im$_2$	Rb$_6$	Ra$_6$	Rt$_6$	2$_7$

Im$_2$	Meaning
00	Both Ra and Rb are register values
01	Ra specifies an immediate, Rb is a register value
10	Ra is a register value, Rb specifies an immediate
11	reserved

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_STAT – Get Status

Description:

This instruction gets the status of the neurons. There is a bit in Rt for each neuron. A bit will be set if the neuron is finished performing the calculation of the activation function, otherwise the bit will be clear.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	9_6	$\sim_6$	Rt_6	1_7

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_TRIG – Trigger Calc

Description:

This instruction triggers an NNA cycle for the neurons identified in the bit mask. The bit mask is contained in register Ra.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_7$	0_2	8_6	Ra_6	$\sim_6$	1_7

Clock Cycles: 1

Execution Units: NNA

Notes:

Floating-Point Operations

Precision

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

Four storage formats are supported for binary floats: 128-bit quad precision, 64-bit double precision, 32-bit single precision and 16-bit half precision values.

Representations

Binary Floats
Quad Precision, Float:128

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

Quad Precision, long double

127	126 112	111 0
S	Exponent$_{15}$	Significand$_{112}$

Double Precision, double

63	62 52	51 0
S	Exponent$_{11}$	Significand$_{52}$

Single Precision, float

31	30 23	22 0
S	Exponent$_{8}$	Significand$_{23}$

Half Precision, short float

15	14 10	9 0
S	Exponent$_{5}$	Significand$_{10}$

Decimal Floats

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

127	126 122	121 110	109 0
S	$Combo_5$	$Exponent_{12}$	$Significand_{110}$

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

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

64-bit double precision decimal floating point:

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.

NaN Boxing

The core performs all floating-point operations at the highest precision; however, values of a lower precision may be held in the floating-point registers. Lower precision values are ‘NaN boxed’ meaning all the bits needed to extend the value to the width of the register are filled with ones. The sign bit of the number is preserved. Thus, lower precision values encountered in calculations are treated as NaNs.

Example: NaN boxed double precision value.

127	126 112	111	0
S	$Exponent_{15}$	$Significand_{112}$	
S	7FFFh	FFFFFFFFFFFFh	Double Precision $Float_{64}$

Note that the floating-point load instructions automatically convert values to quad precision.

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

General Instruction Format

Precision:

A two-bit field, P, in the instruction determines the precision of the calculations.

39 37	36 34	33 29	28 26	25	24 19	18 13	12 7	6 3	2 1	0
Fmt_3	Pr_3	4_5	Rm_3	~	Rb_6	Ra_6	Rt_6	12_4	P_2	T

P_2	Precision
0	Quad
1	Double
2	Single
3	Half

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

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_5	$\sim_4$	0_6	Ra_6	Rt_6	$Opcode_7$

Opcode7:

96	Q	Quad precision
98	D	Double precision
100	S	Single precision
102	H	Half precision

Operation:

Ft = Abs(Fa)

Execution Units: All FPUs

Clock Cycles: 1

Exceptions: none

Notes:

FADD –Float Addition

Description:

Add two source operands and place the sum in the target register. Immediate values are converted from lower precision to operational precision.

Supported Operand Sizes:

Operation:

Rt = Ra + Rb or Rt = Ra + Imm

Clock Cycles:

Half	8
Single	8
Double	8
Quad	8

Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Format: FLT2

FADD Ft, Fa, Fb

39 37	36 34	33 29	28 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	4_5	Rm_3	~	Rb_6	Ra_6	Rt_6	$Opcode_7$

Opcode7:

96	Q	Quad precision
98	D	Double precision
100	S	Single precision
102	H	Half precision

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. The source operands are floating-point values, the target operand is an integer. No rounding occurs.

Operation:

Rt = Fa ? Fb or Rt = Fa ? Imm

Clock Cycles: 1

Execution Units: All Integer ALU's

Exceptions: none

Instruction Format: FLT2

FCMP Rt, Ra, Rb

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	13_5	$\sim_4$	Rb_6	Ra_6	Rt_6	$Opcode_7$

Opcode7:

96	Q	Quad precision
98	D	Double precision
100	S	Single precision
102	H	Half precision

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

FCONST – Load Float Constant

Description:

This instruction loads a constant from the constant ROM and places the value in Rt.

Integer Instruction Format: R1

FCONST Rt, N

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	$\sim_2$	4_6	N_6	Rt_6	98_7

Clock Cycles: 1

Operation:

Ft = FConst[N]

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

N_6	Binary64	Decimal	
0	3fe0000000000000	0.5	
1	3ff0000000000000	1.0	
2	4000000000000000	2.0	
3	3ff8000000000000	1.5	
4	0x5FE6EB50C7B537A9		Lomont reciprocal square root magic
21			
22			
23			
57	7FF0000000000000		infinity
58	7FF0000000000001		Nan – infinity - infinity
59	7FF0000000000002		Nan – infinity / infinity
60	7FF0000000000003		Nan – zero / zero
61	7FF0000000000004		Nan – infinity * zero
62	7FF0000000000005		Nan – square root of infinity
63	7FF0000000000006		Nan – square root of negative

FCOS – Float Cosine

Description:

This instruction computes an approximation of the co-sine value of the contents of the source operand and places the result in Rt.

Integer Instruction Format: R1

FCOS Rt, Ra

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	$\sim_2$	33_6	Ra_6	Rt_6	$Opcode_7$

Opcode7:

96	Q	Quad precision
98	D	Double precision
100	S	Single precision
102	H	Half precision

Clock Cycles:

Half	24
Single	
Double	42
Quad	

Operation:

Ft = cos(Fa)

Execution Units: FPU #0

Clock Cycles: 125

Exceptions: none

Notes:

FCVTD2Q – Convert Double to Quad Precision

Description:

This instruction converts the contents of the source operand to the equivalent of a quad precision value and places the result in Rt.

Integer Instruction Format: R1

FCVTD2Q Rt, Ra

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_5	$\sim_4$	11_6	Ra_6	Rt_6	98_7

Clock Cycles: 1

Operation:

Rt = Cvt(Ra, double)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTQ2D – Round Quad to Double Precision

Description:

This instruction rounds the contents of the source operand to the equivalent of a double precision value and places the result in Rt. Note the register continues to contain a quad precision value. This instruction may be used in preparation for a store.

Integer Instruction Format: R1

FCVTQ2D Rt, Ra

39 37	36 34	33 29	28 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_5	Rm_3	~	18_6	Ra_6	Rt_6	98_7

Clock Cycles: 1

Operation:

Rt = Round(Ra, double)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTQ2H – Round Quad to Half Precision

Description:

This instruction rounds the contents of the source operand to the equivalent of a half precision value and places the result in Rt. Note the register continues to contain a quad precision value. This instruction may be used in preparation for a store.

Integer Instruction Format: R1

FCVTQ2H Rt, Ra

39 37	36 34	33 29	28 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_5	Rm_3	~	16_6	Ra_6	Rt_6	98_7

Clock Cycles: 1

Operation:

Rt = Round(Ra, half)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTQ2S – Round Quad to Single Precision

Description:

This instruction rounds the contents of the source operand to the equivalent of a single precision value and places the result in Rt. Note the register continues to contain a quad precision value. This instruction may be used in preparation for a store.

Integer Instruction Format: R1

FCVTQ2S Rt, Ra

39 37	36 34	33 29	28 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_5	Rm_3	~	17_6	Ra_6	Rt_6	98_7

Clock Cycles: 1

Operation:

Rt = Round(Ra, single)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCVTS2Q – Convert Single to Quad Precision

Description:

This instruction converts the contents of the source operand to the equivalent of a quad precision value and places the result in Rt.

Integer Instruction Format: R1

FCVTD2Q Rt, Ra

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_5	$\sim_4$	10_6	Ra_6	Rt_6	98_7

Clock Cycles: 1

Operation:

Rt = Cvt(Ra, single)

Execution Units: FPU #0

Clock Cycles: 1

Exceptions: none

Notes:

FCX – Clear Floating-Point Exceptions

Description:

This instruction clears floating point exceptions. The Exceptions to clear are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: EX

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
$\sim_3$	Pr_3	3_7	0_2	$\sim_6$	Ra_6	$Uimm_6$	112_7

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	global invalid operation clears the following: - division of infinities - zero divided by zero - subtraction of infinities - infinity times zero - NaN comparison - division by zero
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	summary exception

FDIV –Float Division

Description:

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

Supported Operand Sizes:

Operation:

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

Clock Cycles:

Execution Units: All Integer ALU's

Exceptions: none

Notes:

This instruction is currently implemented as a macro instruction with interruptible micro-code. Interrupt latency is eight clock cycles.

Instruction Format: FLT2

FDIV Rt, Ra, Rb

39 37	36 34	33 29	28 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	7_5	Rm_3	~	Rb_6	Ra_6	Rt_6	98_7

Opcode7:

$Opcode_7$	Ext	Precision	Clocks
96	Q	Quad precision	
98	D	Double precision	46
100	S	Single precision	30
102	H	Half precision	10

FDP –Float Dot Product

Description:

Multiply two pairs of source operands, add the products and place the result in the target register. All register values are treated as quad precision floating-point values.

Note this instruction uses the target register as a source operand and will overwrite the value in that register.

Instruction Format: FLT3

FDP Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	7_3	Rc_6	Rb_6	Ra_6	Rt_6	99_7

Operation:

Rt = (Rt * Ra) + (Rb * Rc)

Clock Cycles: 8

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

FDX – Disable Floating Point Exceptions

Description:

This instruction disables floating point exceptions. The Exceptions disabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: EX

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
$\sim_3$	Pr_3	4_7	0_2	$\sim_6$	Ra_6	$Uimm_6$	112_7

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Disabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FEX – Enable Floating Point Exceptions

Description:

This instruction enables floating point exceptions. The Exceptions enabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: EX

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
$\sim_3$	Pr_3	5_7	0_2	$\sim_6$	Ra_6	$Uimm_6$	112_7

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

FMA –Float Multiply and Add

Description:

Multiply two source operands, add a third operand and place the result in the target register. All register values are treated as quad precision floating-point values.

Instruction Format: FLT3

FMA Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0_3	Rc_6	Rb_6	Ra_6	Rt_6	99_7

Operation:

Rt = Ra * Rb + Rc

Clock Cycles: 8

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

FMS –Float Multiply and Subtract

Description:

Multiply two source operands, subtract a third operand and place the result in the target register. All register values are treated as quad precision floating-point values.

Instruction Format: FLT3

FMS Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_3	Rc_6	Rb_6	Ra_6	Rt_6	99_7

Operation:

Rt = Ra * Rb - Rc

Clock Cycles: 8

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

FMUL –Float Multiplication

Description:

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

Operation:

Rt = Ra * Rb or Rt = Ra * Imm

Clock Cycles: 8

Execution Units: All Integer ALU's

Exceptions: none

Notes:

Instruction Format: FLT2

FMUL Rt, Ra, Rb

39 37	36 34	33 29	28 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	6_5	Rm_3	~	Rb_6	Ra_6	Rt_6	98_7

FNMA –Float Negate Multiply and Add

Description:

Multiply two source operands, add a third operand and place the negative of the result in the target register. All register values are treated as quad precision floating-point values.

Instruction Format: FLT3

FNMA Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	2_3	Rc_6	Rb_6	Ra_6	Rt_6	99_7

Operation:

Rt = -(Ra * Rb + Rc)

Clock Cycles: 8

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

FNMS –Float Negate Multiply and Subtract

Description:

Multiply two source operands, subtract a third operand and place the negative of the result in the target register. All register values are treated as quad precision floating-point values.

Instruction Format: FLT3

FNMS Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	3_3	Rc_6	Rb_6	Ra_6	Rt_6	99_7

Operation:

Rt = -(Ra * Rb – Rc)

Clock Cycles: 8

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

FRES – Floating point Reciprocal Estimate

Description:

Estimates the reciprocal of the floating-point number in register Ra and place the result into target register Rt.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	Est_2	23_6	Ra_6	Rt_6	98_7

Est_2	Bits	Clocks
0	8	2
1	16	22
2	32	38
3	53	54

Operation:

Rt = fres (Ra)

Execution Units: Floating Point

Notes:

This function is currently micro-coded and interruptible.

FRSQRTE – Float Reciprocal Square Root Estimate

Description:

Estimate the reciprocal of the square root of the number in register Ra and place the result into target register Rt.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	Est_2	22_6	Ra_6	Rt_6	98_7

Est_2	Bits	Clocks
0	9	46
1	17	70
2	34	94
3	68	119

Execution Units: Floating Point

Notes:

Taking the reciprocal square root of a negative number or of infinity results in a Nan output.

This function is currently micro-coded and interruptible.

FSCALEB –Scale Exponent

Description:

Add the source operand to the exponent. The second source operand is an integer value. No rounding occurs.

Instruction Formats:

FSCALEB Rt, Ra, Rb

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0_5	$\sim_4$	Rb_6	Ra_6	Rt_6	98_7

Operation:

Clock Cycles:

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

FSEQ – Float Set if Equal

Description:

Compares two source operands for equality and places the result in the target register. The result is a Boolean true or false. Positive and negative zero are considered equal. 32, 64, and 128-bit immediates are supported. For FSEQ if either operand is a NaN zero the result is false. No rounding occurs.

Operation:

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

Clock Cycles: 1

Execution Units: All FPU’s

Exceptions: none

Notes:

Instruction Formats:

FSEQ Rt, Ra, Rb

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	8_5	$\sim_4$	Rb_6	Ra_6	Rt_6	$Opcode_7$

Opcode7:

96	Q	Quad precision
98	D	Double precision
100	S	Single precision
102	H	Half precision

FSGNJ – Float Sign Inject

Description:

Copy the sign of Ra and the exponent and significand of Rb into the target register Rt. No rounding occurs.

Operation:

Rt = {Ra.sign, Rb.exp, Rb.sig}

Clock Cycles: 1

Execution Units: All FPU’s

Exceptions: none

Notes:

Instruction Formats:

FSGNJ Rt, Ra, Rb

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	16_5	$\sim_4$	Rb_6	Ra_6	Rt_6	98_7

FSGNJN – Float Negative Sign Inject

Description:

Copy the negative of the sign of Ra and the exponent and significand of Rb into the target register Rt. No rounding occurs.

Operation:

Rt = {~Ra.sign, Rb.exp, Rb.sig}

Clock Cycles: 1

Execution Units: All FPU’s

Exceptions: none

Notes:

Instruction Formats:

FSGNJN Rt, Ra, Rb

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	17_5	$\sim_4$	Rb_6	Ra_6	Rt_6	98_7

FSGNJX – Float Sign Inject Xor

Description:

Copy the xor of the sign of Ra and Rb and the exponent and significand of Rb into the target register Rt. No rounding occurs.

Operation:

Rt = {Ra.sign ^ Rb.sign, Rb.exp, Rb.sig}

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSGNJX Rt, Ra, Rb

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	18_5	$\sim_4$	Rb_6	Ra_6	Rt_6	98_7

FSIGMOID – Sigmoid Approximate

Description:

This function uses a 1024 entry 32-bit precision lookup table with linear interpolation to approximate the logistic sigmoid function in the range -8.0 to +8.0. Outside of this range 0.0 or +1.0 is returned. The sigmoid output is between 0.0 and +1.0. The value of the sigmoid for register Ra is returned in register Rt as a 64-bit double precision floating-point value.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	$\sim_2$	48_6	Ra_6	Rt_6	98_7

Clock Cycles: 5

Execution Units: Floating Point

FSIGN – Sign of Number

Description:

This instruction provides the sign of a double precision floating point number contained in a general-purpose register as a floating-point double result. The result is +1.0 if the number is positive, 0.0 if the number is zero, and -1.0 if the number is negative.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	$\sim_2$	6_6	Ra_6	Rt_6	98_7

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = sign of (Ra)

FSIN – Float Sine

Description:

This instruction computes an approximation of the sine value of the contents of the source operand and places the result in Rt.

Integer Instruction Format: R1

FSIN Rt, Ra

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	$\sim_2$	32_6	Ra_6	Rt_6	98_7

Clock Cycles: 1

Operation:

Ft = sin(Fa)

Execution Units: FPU #0

Clock Cycles: 125

Exceptions: none

Notes:

FSLE – Float Set if Less Than or Equal

Description:

Compares two source operands for less than or equal and places the result in the target register. The target register is a predicate register. The result is a Boolean true or false. Positive and negative zero are considered equal. 16, 32, 64, and 128-bit immediates are supported. For FSLE if either operand is a NaN zero the result is false. No rounding occurs. This instruction may also test for greater than or equal by swapping operands.

Operation:

Prt = Fa < Fb or Prt = Fa < Imm

Clock Cycles: 1

Execution Units: All FPU’s

Exceptions: none

Notes:

Instruction Formats:

FSLE Prt, Ra, Rb

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	11_5	$\sim_4$	Rb_6	Ra_6	Rt_6	98_7

FSLT – Float Set if Less Than

Description:

Compares two source operands for less than and places the result in the target register. The target register is a predicate register. The result is a Boolean true or false. Positive and negative zero are considered equal. 16, 32, 64, and 128-bit immediates are supported. For FSLT if either operand is a NaN zero the result is false. No rounding occurs. This instruction may also test for greater than by swapping operands.

Operation:

Prt = Fa < Fb or Prt = Fa < Imm

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSLT Prt, Ra, Rb

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	10_5	$\sim_4$	Rb_6	Ra_6	Rt_6	98_7

FSNE – Float Set if Not Equal

Description:

Compares two source operands for equality and places the result in the target predicate register. The result is a Boolean true or false. Positive and negative zero are considered equal. 16, 32, 64, and 128-bit immediates are supported. No rounding occurs.

Operation:

Prt = Fa != Fb or Prt = Fa != Imm

Clock Cycles: 1

Execution Units: All FPU's

Exceptions: none

Notes:

Instruction Formats:

FSNE Ft, Fa, Fb

39 37	36 34	33 29	28 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	9_5	$\sim_4$	Rb_6	Ra_6	Rt_6	98_7

FSQRT – Floating point square root

Description:

Take the square root of the floating-point number in register Ra and place the result into target register Rt. The sign bit (bit 63) of the register is set to zero. This instruction can generate NaNs.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	$\sim_2$	8_6	Ra_6	Rt_6	98_7

Operation:

Rt = fsqrt (Ra)

Clock Cycles: 72

Execution Units: Floating Point

FSUB –Float Subtraction

Description:

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

Supported Operand Sizes:

Operation:

Ft = Fa - Fb or Ft = Fa - Imm

Clock Cycles: 8

Execution Units: All Integer ALU’s

Exceptions: none

Notes:

Instruction Format: FLT2

FSUB Ft, Fa, Fb

39 37	36 34	33 29	28 26	25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	5_5	Rm_3	~	Rb_6	Ra_6	Rt_6	98_7

FTRUNC – Truncate Value

Description:

The FTRUNC instruction truncates off the fractional portion of the number leaving only a whole value. For instance, ftrunc(1.5) equals 1.0. Ftrunc does not change the representation of the number. To convert a value to an integer in a fixed-point representation see the FTOI instruction.

Instruction Format: R1

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	1_7	$\sim_2$	21_6	Ra_6	Rt_6	$Opcode_7$

Opcode7:

96	Q	Quad precision
98	D	Double precision
100	S	Single precision
102	H	Half precision

Clock Cycles: 1

Execution Units: Floating Point

FTX – Trigger Floating Point Exceptions

Description:

This instruction triggers floating point exceptions. The Exceptions to trigger are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

Instruction Format: EX

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
$\sim_3$	Pr_3	2_7	0_2	$\sim_6$	Ra_6	$Uimm_6$	112_7

Execution Units: All Floating Point

Operation:

Exceptions:

Bit	Exception Enabled
0	global invalid operation
1	overflow
2	underflow
3	divide by zero
4	inexact operation
5	reserved

Decimal Floating-Point Instructions

DFADD – Add Register-Register

Description:

Add two registers and place the sum in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The values are treated as quad precision decimal floating-point values.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	4_7	$\sim_2$	Rb_6	Ra_6	Rt_6	102_7

Execution Units: All ALU's

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

DFMUL – Multiply Register-Register

Description:

Multiply two registers and place the product in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The values are treated as quad precision decimal floating-point values.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	6_7	$\sim_2$	Rb_6	Ra_6	Rt_6	102_7

Execution Units: All ALU’s

Operation:

Rt = Ra * Rb

Exceptions:

Notes:

DFSUB – Add Register-Register

Description:

Subtract two registers and place the difference in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The values are treated as quad precision decimal floating-point values.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	5_7	$\sim_2$	Rb_6	Ra_6	Rt_6	102_7

Execution Units: All ALU’s

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

Load / Store Instructions

Overview

Addressing Modes

Load and store instructions have two addressing modes, register indirect with displacement and indexed addressing.

For vector indexed addressing Ra acts as a base address register. If Rb is a scalar value then it is used to increment the load / store address according to the vector element. Otherwise, if Rb is a vector value it is used directly as an index.

The ‘C’ bit of the instruction indicates the vector is compressed in memory. When compressed, for stores if a mask bit is clear then no value is stored to memory and the memory address does not increment. Loads are similar.

Load Formats

Register Indirect with Displacement Format

For register indirect with displacement addressing the load or store address is the sum of a register Ra and a displacement constant found in the instruction. A postfix immediate may be used for a larger displacement.

Instruction Format: d[Ra]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	64_7

Instruction Format: d[Ra+Rb*]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	Fn_6	Ca_2	~	Sc	Rb_6	Ra_6	Rt_6	79_7

Store Formats

Register Indirect with Displacement Format

For register indirect with displacement addressing the load or store address is the sum of a register Ra and a displacement constant found in the instruction.

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rs_6	$Opcode_7$

Indexed Format

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	Fn_6	Ca_2	~	Sc	Rb_6	Ra_6	Rs_6	$Opcode_7$

AMOADD – AMO Addition

Description:

Atomically add source operand register Ra 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 the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOADD Rt, Ra, d[Rc+Rb*]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	0_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

Clock Cycles:

AMOAND – AMO Bitwise ‘And’

Description:

Atomically bitwise ‘and’ source operand register Ra 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 the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOAND Rt, Ra, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	1_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

Clock Cycles:

AMOASL – AMO Arithmetic Shift Left

Description:

Atomically shift the contents of memory to the left by one bit and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOASL Rt, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	8_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

Clock Cycles:

AMOEOR – AMO Bitwise Exclusively ‘Or’

Description:

Atomically bitwise exclusively ‘or’ source operand register Ra 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 the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOEOR Rt, Ra, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	3_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

Clock Cycles:

AMOLSR – AMO Logical Shift Right

Description:

Atomically shift the contents of memory to the right by one bit and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOASL Rt, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	9_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

Clock Cycles:

AMOMAX – AMO Maximum

Description:

Atomically determine the maximum of source operand register Ra and a 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 the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOMAX Rt, Ra, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	5_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

AMOMAXU – AMO Unsigned Maximum

Description:

Atomically determine the maximum of source operand register Ra and a value from memory and store the result back to memory. Values are treated as unsigned integers. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOMAX Rt, Ra, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	13_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

AMOMIN – AMO Minimum

Description:

Atomically determine the minimum of source operand register Ra and a 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 the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOAND Rt, Ra, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	4_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

AMOMINU – AMO Unsigned Minimum

Description:

Atomically determine the minimum of source operand register Ra and a value from memory and store the result back to memory. Values are treated as unsigned integers. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOAND Rt, Ra, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt$_2$	Pr$_3$	12$_4$	Ar	Ca$_2$	D	Sc	Rb$_6$	Ra$_6$	Rs$_6$	92$_7$

AMOOR – AMO Bitwise ‘Or’

Description:

Atomically bitwise ‘and’ source operand register Ra 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 the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOOR Rt, Ra, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	2_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

Clock Cycles:

AMOROL – AMO Rotate Left

Description:

Atomically rotate the contents of memory to the left by one bit and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOASL Rt, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	10_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

Clock Cycles:

AMOROR – AMO Rotate Right

Description:

Atomically rotate the contents of memory to the right by one bit and store the result back to memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOASL Rt, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	11_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

Clock Cycles:

AMOSWAP – AMO Swap

Description:

Atomically swap source operand register Ra with value from memory. The original value of the memory cell is stored in register Rt. The memory address is the sum of Rc and scaled index Rb.

Supported Operand Sizes: .h

Instruction Formats: AMO

AMOSWAP Rt, Ra, d[Rc+Rb*Sc]

39 38	37 35	34 31	30 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	6_4	Ar	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	92_7

Clock Cycles:

CACHE <cmd>,<ea>

Description:

Issue command to cache controller.

Instruction Format: d[Rn]

39 38	37 35	34 23	22 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{11..0}$	Pi_2	Ca_2	Ra_6	Cmd_6	75_7

Instruction Format: d[Ra+Rb*]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	0_6	Ca_2	D	Sc	Rb_6	Ra_6	Cmd_6	78_7

Notes:

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

CAS – Compare and Swap

Description:

If the contents of the addressed memory cell equals the contents of Rb then a 128-bit value is stored to memory from the source register Rc. The original contents of the memory cell are loaded into register Rt. The memory address is contained in register Ra. If the operation was successful then Rt and Rb will be the same value and predicate register Prt will be set to true, otherwise Prt is set to false. The compare and swap operation is an atomic operation performed by the memory controller.

Instruction Format:

6362	61 59	58 53	5251	5049	4845	44 29	28 24	2322	22 17	16 12	11 7	6 0
Fmt_2	Pr_3	31_6	Ot_2	Ca_2	Prt_4	$Disp_{15..0}$	Rc_5	$\sim_2$	Rb_5	Ra_5	Rt_5	79_7

Operation:

```
Prt = false
Rt = memory[d[Ra]]
if memory[d[Ra]] = Rb
        memory[d[Ra]] = Rc
        Prt = true
```

Assembler:

CAS Rt, Prt, Rb, Rc, [Ra]

Note:

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

FLDD Rn,<ea> - Load Float Double Precision

Description:

Load register Ft with a double precision float from memory. The source value is converted to a quad precision value. No rounding occurs.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	58_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	22_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

FLDH Rn,<ea> - Load Float Half Precision

Description:

Load register Ft with a half precision float from memory. The source value is converted to a quad precision value. No rounding occurs.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	56_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	18_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

FLDQ Rn,<ea> - Load Float Quad Precision

Description:

Load register Ft with a quad precision float from memory. No rounding occurs.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	58_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	24_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

FLDS Rn,<ea> - Load Float Single Precision

Description:

Load register Ft with a single precision float from memory. The source value is converted to a quad precision value. No rounding occurs.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	57_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	20_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

FSTD Rn,<ea> - Store Float Double Precision

Description:

Store register Ft as a double precision float to memory. The value will be rounded according to the current rounding mode.

Instruction Format: d[Rn]

39 38	37 35	34 21	2019	18 13	12 7	6 0
Fmt$_2$	Pr$_3$	Disp$_{13..0}$	Ca$_2$	Ra$_6$	Rs$_6$	62$_7$

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt$_2$	Pr$_3$	11$_6$	Ca$_2$	D	Sc	Rb$_6$	Ra$_6$	Rs$_6$	87$_7$

Ca$_2$	Policy	Qualifier	Comment
0	Write through	.wt	Always write through to main memory
1	Writeback	.wb	Store to main memory only when data not in cache
2	Write through, write allocate	.wta	Write to main memory, and allocate in cache
3	Write back, write allocate	.wba	Allocate in cache, write to cache

FSTH Rn,<ea> - Store Float Half Precision

Description:

Store register Ft as a half precision float to memory. The value will be rounded according to the current rounding mode.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rs_6	60_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	9_6	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	87_7

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

FSTQ Rn,<ea> - Store Float Quad Precision

Description:

Store register Ft as a quad precision float to memory.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rs_6	63_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	12_6	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	87_7

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

FSTS Rn,<ea> - Store Float Single Precision

Description:

Store register Ft as a single precision float to memory. The value will be rounded according to the current rounding mode.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	61_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	10_6	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	87_7

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

LDA Rn,<ea> - Load Address

Description:

Load register Rt with the computed memory address.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	$\sim_2$	Ra_6	Rt_6	74_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	10_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

LDB Rn,<ea> - Load Byte

Description:

Load register Rt with a byte from source. The source value is sign extended to the machine width.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	64_7

Instruction Format: d[Ra+Rb*]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	0_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

LDBU Rn,<ea> - Load Unsigned Byte

Description:

Load register Rt with a byte from source. The source value is zero extended to the machine width.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	65_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	1_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

LDM Pn,<ea>

Description:

Load multiple registers from source. Pr3 contains a bitmask indicating which registers to load.

Instruction Format: d[Rn]

31 30	29 27	26 19	1817	16 12	11	10 7	6 0
Fmt_2	Pr_3	$Disp_{7..0}$	Ca_2	Ra_5	F	$\sim_4$	73_7

Instruction Format: d[Ra+Rb*Sc]

63 62	61 59	58 53	5251	5049	48 25	24 22	22 17	16 12	11 7	6 0
Fmt_2	Pr_3	9_6	Ot_2	Ca_2	$Disp_{23..0}$	Sc_3	Rb_5	Ra_5	$\sim_5$	79_7

F	Register File
0	Integer
1	Float

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

LDO Rn,<ea> - Load Octa

Description:

Load register Rt with an octa from memory. The memory value is sign extended to the machine width.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	70_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	6_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

LDOU Rn,<ea> - Load Unsigned Octa

Description:

Load register Rt with an octa from memory. The memory value is zero extended to the machine width.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	71_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	7_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

LDT Rn,<ea> - Load Tetra

Description:

Load register Rt with a tetra from memory. The memory value is sign extended to the machine width.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	68_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	4_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

LDTU Rn,<ea> - Load Unsigned Tetra

Description:

Load register Rt with a tetra from memory. The memory value is zero extended to the machine width.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	69_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	5_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

LDW Rn,<ea> - Load Wyde

Description:

Load register Rt with a wyde from source. The source value is sign extended to the machine width.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	66_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	2_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

LDWU Rn,<ea> - Load Unsigned Wyde

Description:

Load register Rt with a wyde from source. The source value is zero extended to the machine width.

Instruction Format: d[Rn]

39 38	37 35	34 21	2019	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rt_6	67_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	3_6	Ca_2	D	Sc	Rb_6	Ra_6	Rt_6	79_7

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

STB Rs,<ea> - Store Byte

Description:

Store a byte from register Rs to memory.

Instruction Format: d[Rn]

39 38	37 35	34 21	2019	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rs_6	80_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	0_6	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	87_7

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

STM Pn,<ea>

Description:

Store multiple registers to memory. Pr3 contains a bitmask indicating which registers to store.

Instruction Format: d[Rn]

31 30	29 27	26 19	1817	16 12	11	10 7	6 0
Fmt$_2$	Pr$_3$	Disp$_{7..0}$	Ca$_2$	Ra$_5$	F	~$_4$	85$_7$

Instruction Format: d[Ra+Rb*Sc]

63 62	61 59	58 53	5251	5049	48 25	24 22	22 17	16 12	11 7	6 0
Fmt$_2$	Pr$_3$	5$_6$	Ot$_2$	Ca$_2$	Disp$_{23..0}$	Sc$_3$	Rb$_5$	Ra$_5$	~$_5$	87$_7$

F	Register File
0	Integer
1	Float

Ca$_2$	Policy	Qualifier	Comment
0	Write through	.wt	Always write through to main memory
1	Writeback	.wb	Store to main memory only when data not in cache
2	Write through, write allocate	.wta	Write to main memory, and allocate in cache
3	Write back, write allocate	.wba	Allocate in cache, write to cache

STO Rs,<ea> - Store Octa

Description:

Store an octa from register Rs to memory.

Instruction Format: d[Rn]

39 38	37 35	34 21	2019	18 13	12 7	6 0
Fmt$_2$	Pr$_3$	Disp$_{13..0}$	Ca$_2$	Ra$_6$	Rs$_6$	83$_7$

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt$_2$	Pr$_3$	3$_6$	Ca$_2$	D	Sc	Rb$_6$	Ra$_6$	Rs$_6$	87$_7$

Ca$_2$	Policy	Qualifier	Comment
0	Write through	.wt	Always write through to main memory
1	Writeback	.wb	Store to main memory only when data not in cache
2	Write through, write allocate	.wta	Write to main memory, and allocate in cache
3	Write back, write allocate	.wba	Allocate in cache, write to cache

STT Rs,<ea> - Store Tetra

Description:

Store a tetra from register Rs to memory.

Instruction Format: d[Rn]

39 38	37 35	34 21	2019	18 13	12 7	6 0
Fmt$_2$	Pr$_3$	Disp$_{13..0}$	Ca$_2$	Ra$_6$	Rs$_6$	82$_7$

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	2827	26	25	24 19	18 13	12 7	6 0
Fmt$_2$	Pr$_3$	2$_6$	Ca$_2$	D	Sc	Rb$_6$	Ra$_6$	Rs$_6$	87$_7$

Ca$_2$	Policy	Qualifier	Comment
0	Write through	.wt	Always write through to main memory
1	Writeback	.wb	Store to main memory only when data not in cache
2	Write through, write allocate	.wta	Write to main memory, and allocate in cache
3	Write back, write allocate	.wba	Allocate in cache, write to cache

STW Rs,<ea> - Store Wyde

Description:

Store a wyde from register Rs to memory.

Instruction Format: d[Rn]

39 38	37 35	34 21	20 19	18 13	12 7	6 0
Fmt_2	Pr_3	$Disp_{13..0}$	Ca_2	Ra_6	Rs_6	81_7

Instruction Format: d[Ra+Rb*Sc]

39 38	37 35	34 29	28 27	26	25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	1_6	Ca_2	D	Sc	Rb_6	Ra_6	Rs_6	87_7

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

Block Instructions

BCMP – Block Compare

Description:

This instruction compares data from the memory location addressed by Ra to the memory location addressed by Rb until the loop counter LC reaches zero or until a mismatch occurs. Ra and Rb increment by the specified amount. This instruction is interruptible. A predicate register is set to true if the entire block is equal, otherwise it is set to false.

Instruction Format:

39 37	36 34	33 31	30 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_3$	Sz_4	Im_2	Rb_6	Ra_6	Rt_6	109_7

Sz_4	Adjustment Amount
1	1
2	2
3	4
4	8
5	16
15	-1
14	-2
13	-4
12	-8
11	-16
others	reserved

Assembler Example

```
LDI LC,200
BCMP.O Pr1,[Ra]+,[Rb]+
SUBF LC,LC,200          ; get index of difference
```

Execution Units: Memory

Operation:

```
temp = 0
Prt = true
while LC <> 0 and mem[Rb] = mem[Ra]
        Ra = Ra + amt
        Rb = Rb + amt
        LC = LC – 1
        If mem[Rb] != mem[Ra]
                Prt = false
```

BFND – Block Find

Description:

This instruction compares data from the memory location in Rb to the data in register Ra. A target predicate register is set if the data is found.

Instruction Format:

39 37	36 34	33 31	30 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_3$	Sz_4	Im_2	Rb_6	Ra_6	Rt_6	108_7

Sz_4	Adjustment Amount
1	1
2	2
3	4
4	8
5	16
15	-1
14	-2
13	-4
12	-8
11	-16
others	reserved

Execution Units: Memory

Operation:

BMOV –Block Move

Description:

This instruction moves a data from the memory location addressed by Ra to the memory location addressed by Rb until the loop counter LC reaches zero. Ra and Rb are adjusted by a specified amount after the move. This instruction is interruptible.

Instruction Format:

39 37	36 34	33 31	30 27	26 25	24 19	18 13	12 11	10 9	8 7	6 0
Fmt_3	Pr_3	$\sim_3$	Sz_4	Im_2	Rb_6	Ra_6	$\sim_2$	Bi_2	Ai_2	111_7

Sz_4	Adjustment Amount
1	1
2	2
3	4
4	8
5	16
15	-1
14	-2
13	-4
12	-8
11	-16
others	reserved

Ai_2 / Bi_2	
0	No change
1	Increment
2	Decrement
3	reserved

Assembler Example

```
LDI LC,200
BMOV.B [Ra]+,[Rb]+
```

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        t0 = mem[Ra]
        mem[Rb] = t0
        Ra = Ra + amt
```

Rb = Rb + amt
LC = LC – 1

BSET – Block Set

Description:

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

Instruction Format:

39 37	36 34	33 31	30 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	$\sim_3$	Sz_4	Im_2	Rb_6	Ra_6	$\sim_6$	110_7

Sz_4	Adjustment Amount
1	1
2	2
3	4
4	8
5	16
15	-1
14	-2
13	-4
12	-8
11	-16
others	reserved

Execution Units: Memory

Operation:

```
if LC <> 0
        mem[Ra] = Rb
        Ra = Ra + amt
        LC = LC – 1
```

Assembler Example

```
LDI LC,200
BSETB Rb,[Ra]+
```

Vector Specific Instructions

MFVL – Move from Vector Length

Description:

This instruction moves the vector length register to a general-purpose register. This is an alternate mnemonic for the MOV instruction.

Instruction Format: MOV

31 29	28 26	25 21	20 19	18 17	16 12	11 7	6 0
Fmt_3	Pr_3	$\sim_5$	1_2	Rt_2	20_5	Rt_5	15_7

Operation:

Rt = VL

MTVL – Move to Vector Length

Description:

This instruction moves a general-purpose register to the vector length register. This is an alternate mnemonic for the MOV instruction. Moving a value larger than the maximum vector length of the machine will result in setting the vector length to the maximum vector length.

Instruction Format: MOV

31 29	28 26	25 21	20 19	18 17	16 12	11 7	6 0
Fmt_3	Pr_3	$\sim_5$	Ra_2	1_2	Ra_5	20_5	15_7

Operation:

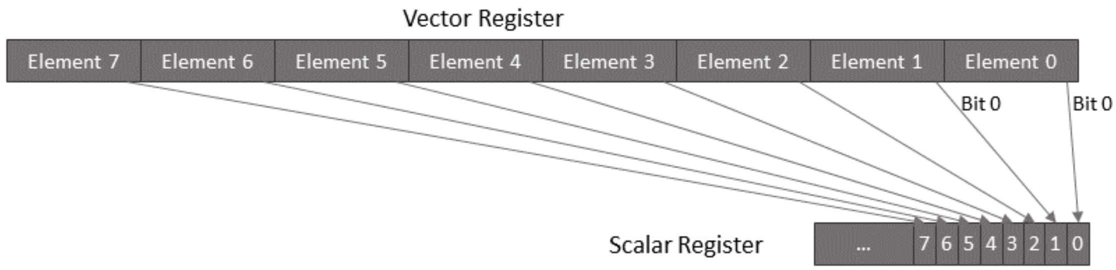
VL = min(Ra, maximum vector length)

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 or a predicate register. Usually, Rb would be zero so that the least significant bit of the vector is copied.

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



V2BITS Rt, Ra, Rb

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	48_7	$\sim_2$	Rb_6	Ra_6	Rt_6	2_7

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 pr1,v1,#8    ; move NE status to bits in m1
vadd v4,v5,v6,pr1         ; perform some masked vector operations
vmuls v7,v8,v9,pr1
vadd v7,v7,v4,pr1
```

VEINS / VMOVSV – Vector Element Insert

Synopsis

Vector element insert.

Description

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

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	51_7	$\sim_2$	Rb_6	Ra_6	Vt_6	2_7

Operation

Vt[Rb] = Ra

Exceptions: none

VEX / VMOVS – Vector Element Extract

Synopsis

Vector element extract.

Description

A vector register element from Va is transferred into a general-purpose register Rt. The element to extract is identified by Rb. Rb and Rt are scalar registers.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	50_7	$\sim_2$	Rb_6	Va_6	Rt_6	2_7

Operation

Rt = Va[Rb]

Exceptions: none

VGNDX – Generate Index

Description

A value in a register Ra is multiplied by the element number and added to a value in Rb and copied to elements of vector register Vt guided by a vector mask register. Ra is a scalar register. This operation may be used to compute memory addresses for a subsequent vector load or store operation. Only the low order 24-bits of Ra are involved in the multiply. The result of the multiply is a product less than 41 bits in size. The multiply is a fast 24x16 bit multiply.

Instruction Format: R2

39 37	36 34	33 27	26 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	52_7	$\sim_2$	Rb_6	Ra_6	Vt_6	2_7

Operation

y = 0

for x = 0 to VL - 1

if (Pr[x])

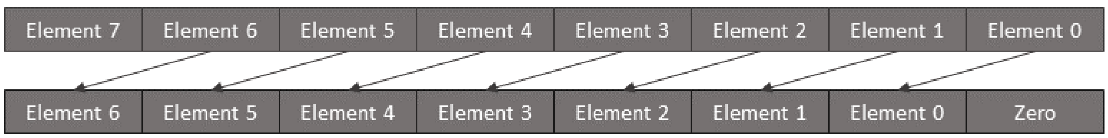
Vt[y] = Ra * y + Rb

y = y + 1

VSHLV – Shift Vector Left

Description

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



Instruction Formats:

VSHLV Vt, Va, Rb

31 29	28 26	25 22	21 17	16 12	11 7	6 0
Fmt_3	Pr_3	6_4	Rb_5	Va_5	Vt_5	18_7

VSHLV Rt, Ra, Imm$_5$

31 29	28 26	25 22	21 17	16 12	11 7	6 0
Fmt_3	Pr_3	14_4	Imm_5	Va_5	Vt_5	18_7

Operation

Amt = Rb

For x = VL-1 to Amt

Vt[x] = Va[x-amt]

For x = Amt-1 to 0

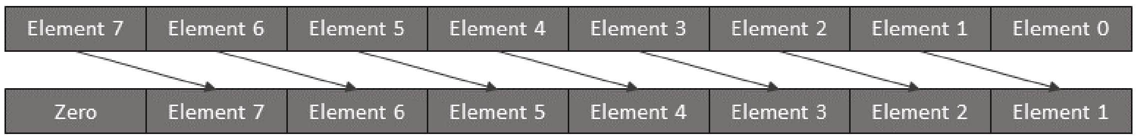
Vt[x] = 0

Exceptions: none

VSHRV – Shift Vector Right

Description

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



VSHRV Rt, Ra, Rb

31 29	28 26	25 22	21 17	16 12	11 7	6 0
Fmt_3	Pr_3	7_4	Rb_5	Va_5	Vt_5	18_7

VSHRV Rt, Ra, Imm$_5$

31 29	28 26	25 22	21 17	16 12	11 7	6 0
Fmt_3	Pr_3	15_4	Imm_5	Va_5	Vt_5	18_7

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

Predicate Operations

PRLAST – Find Last Set Bit

Description

The position of the last bit set in the predicate register is copied to the target register. If no bits are set the value is -1. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

Instruction Format:

31 29	28 26	25 21	20 19	18 15	14 12	11 7	6 0
$\sim_3$	Pr_3	15_5	$\sim_2$	Prb_4	$\sim_3$	Rt_5	$48h_8$

Operation

Rt = last set bit number of (Prb)

Exceptions: none

Execution Units: ALUs

Branch / Flow Control Instructions

Overview

Mnemonics

There are mnemonics for specifying the comparison method. Floating-point comparisons prefix the branch mnemonic with ‘F’ as in FBEQ. Decimal-floating point comparisons prefix the branch mnemonic with ‘DF’ as in DFBEQ. And finally posit comparisons prefix the branch mnemonic with a ‘P’ as in ‘PBEQ’.

Predicated Execution

Flow control instructions do not support predicated instruction execution. Instead, a branch instruction must be used to conditionally branch around the instruction.

Conditions

Conditional branches branch to the target address only if the condition is true. The condition is determined by the comparison of two general-purpose registers.

The original Thor machine used instruction predicates to implement conditional branching. Another instruction was required to set the predicate before branching. Combining compare and branch in a single instruction may reduce the dynamic instruction count. An issue with comparing and branching in a single instruction is that it may lead to a wider instruction format.

The comparison used is determined by a two-bit field in the instruction. There are three comparison types that may be performed as outlined in the table below.

Cm_2	Comparison Type
0	signed integer comparisons
1	Unsigned integer comparisons
2	float comparison
3	reserved

Conditional Branch Format

Branches are 40-bit opcodes.

A 32-bit opcode does not leave a large enough target field for all cases and would end up using two or more instructions to implement most branches. With the prospect of using two instructions to perform compare then branches as many architectures do, it is more space efficient to simply use a wider instruction format.

39 … 25	24 … 19	18 … 13	12 11	10	9 8	7	6 … 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	Cm_2	Lk	$2xh_7$

Branch Conditions

The branch opcode determines the condition under which the branch will execute.

39 25	24 19	18 13	12 11	10	9 8	7	6 0
Target$_{16..2}$	Rb$_6$	Ra$_6$	T$_{1..0}$	Op	$\boxed{\text{Cm}_2}$	Lk	$\boxed{\text{2xh}_7}$

2x	Integer Comparison Test	Float / Decimal Float	Posit
28h	signed less than	less than	less than
29h	signed greater or equal	greater than or equal	greater than or equal
2Ah	signed less than or equal	less than or equal	less than or equal
2Bh	signed greater than	greater than	greater than
2Ch	Bit clear		
2Dh	Bit set		
2Eh	Bit clear imm	Ordered	
2Fh	Bit set imm	Unordered	
26h	equal	Equal	equal
27h	not equal	not equal	not equal

Linkage

Branches may specify a linkage register which is updated with the address of the next instruction. This allows subroutines to be called. Only the primary link register may be updated.

39 25	24 19	18 13	12 11	10	9 8	7	6 0
Target$_{16..2}$	Rb$_6$	Ra$_6$	T$_{1..0}$	Op	Cm$_2$	$\boxed{\text{Lk}}$	2xh$_7$

Lk$_2$	Meaning
0	do not store return address
1	update Lr1

Branch Target

For conditional branches, the target address is formed as the sum of the instruction pointer and a constant specified in the instruction. Branches are IP relative with a range of ±64kB.

The target displacement field is recommended to be at least 16-bits. It is possible to get by with a displacement as small as 12-bits before a significant percentage of branches must be implemented as two or more instructions.

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	Cm_2	Lk	$2xh_7$

Branch to Register

The branch to register instruction allows a conditional return from subroutine to be used or a branch to a value in a register. This is useful when the branch target is more than 16 displacement bits away. Branching to a value in a register allows all bits of the instruction pointer to be set. The target constant may be set to zero.

39 31	30 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{10..2}$	Rc_6	Rb_6	Ra_6	$T_{1..0}$	1	Cm_2	Lk	$2xh_7$

Op	Target
0	17-bit PC relative displacement
1	Sum of Rc and 11-bit constant

Micro-Code Branches

Instruction Format

39 37	36 25	24 19	18 13	12 10	9 8	7	6 0
~$_3$	Target$_{11..0}$	Rb$_6$	Ra$_6$	Cnd$_{2..0}$	Cm$_2$	Lk	34$_7$

Branch Conditions

The Cnd$_3$ field determines the condition under which the branch will execute.

39 37	36 25	24 19	18 13	12 10	9 8	7	6 0
~$_3$	Target$_{11..0}$	Rb$_6$	Ra$_6$	$\boxed{\text{Cnd}_{2..0}}$	Cm$_2$	Lk	34$_7$

Cnd$_3$	Signed / Unsigned	Float	Posit
0	Equal	Equal	
1	Not equal	Not equal	
2	Less than	Less than	
3	Greater than or equal	Greater than or equal	
4	Less than or equal	Less than or equal	
5	Greater than	Greater than	
6	Bit clear / Bit clear imm	unordered	
7	Bit Set / Bit set imm		

Linkage

Branches may specify a linkage register which is updated with the address of the next instruction. This allows subroutines to be called. Only the *micro-code* link register may be updated.

39 37	36 25	24 19	18 13	12 10	9 8	7	6 0
~$_3$	Target$_{11..0}$	Rb$_6$	Ra$_6$	Cnd$_{2..0}$	Cm$_2$	$\boxed{\text{Lk}}$	34$_7$

Branch Target

For micro-code conditional branches, the target address is an absolute micro-code address. Only the micro-code address portion of the PC is updated.

39 37	36 25	24 19	18 13	12 10	9 8	7	6 0
~$_3$	$\boxed{\text{Target}_{11..0}}$	Rb$_6$	Ra$_6$	Cnd$_{2..0}$	Cm$_2$	Lk	34$_7$

BBC – Branch if Bit Clear

Description:

This instruction branches to the target address if bit Rb of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	44_7

Operation:

Lk = next IP
If (Ra.bit[Rb] == 0)
    IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

BBCI – Branch if Bit Clear Immediate

Description:

This instruction branches to the target address if a bit specified in an immediate field of the instruction of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Imm_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	46_7

Operation:

Lk = next IP
If ($Ra.bit[Imm_6]$ == 1)
IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

BBS – Branch if Bit Set

Description:

This instruction branches to the target address if bit Rb of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	45_7

Operation:

Lk = next IP
If (Ra.bit[Rb] == 0)
IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

BBSI – Branch if Bit Set Immediate

Description:

This instruction branches to the target address if a bit specified in an immediate field of the instruction of Ra is set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Imm_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	47_7

Operation:

Lk = next IP
If (Ra.bit[Imm_5] == 1)
    IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

BCC –Branch if Carry Clear

BCC Ra, Rb, label

Description:

Branch if the carry would be set when comparing the first source operand to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

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

Instruction Format: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	1_2	Lk	41_7

Clock Cycles: 4

BCS –Branch if Carry Set

BCS Rm, Rn, label

Description:

This is an alternate mnemonic for the BLO instruction. Branch if the carry would be set because of the comparison of the first operand to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

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

Instruction Format: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	1_2	Lk	40_7

Clock Cycles: 4

BGE –Branch if Greater Than or Equal

BGE Rm, Rn, label

Description:

Branch if the first source operand is greater than or equal to the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	41_7

Clock Cycles: 4

BGEU –Branch if Unsigned Greater Than or Equal

BGEU Rm, Rn, label

Description:

Branch if the first source operand is greater than or equal to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

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

Instruction Format: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	1_2	Lk	41_7

Clock Cycles: 4

BGT –Branch if Greater Than

BGE Rm, Rn, label

Description:

Branch if the first source operand is greater than the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	43_7

Clock Cycles: 4

BGTU –Branch if Unsigned Greater Than

BGE Rm, Rn, label

Description:

Branch if the first source operand is greater than the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	1_2	Lk	43_7

Clock Cycles: 4

BHI –Branch if Higher

BHI Rm, Rn, label

Description:

This is an alternate mnemonic for BGTU. Branch if the first source operand is greater than the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	1_2	Lk	43_7

Clock Cycles: 4

BEQ –Branch if Equal

BEQ Ra, Rb, label

Description:

Branch if two source operands are equal. Both operands are treated as integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	38_7

Clock Cycles: 4

BLE –Branch if Less Than or Equal

BLE Ra, Rb, label

Description:

Branch if the first source operand is less than or equal to the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	42_7

Clock Cycles: 4

BLEU –Branch if Unsigned Less Than or Equal

BLEU Ra, Rb, label

Description:

Branch if the first source operand is less than or equal to the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	1_2	Lk	42_7

Clock Cycles: 4

BLT –Branch if Less Than

BLT Ra, Rb, label

Description:

Branch if the first source operand is less than the second. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	40_7

Clock Cycles: 4

BLTU –Branch if Unsigned Less Than

BLT Ra, Rb, label

Description:

Branch if the first source operand is less than the second. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	1_2	Lk	40_7

Clock Cycles: 4

BNE –Branch if Not Equal

BNE Rm, Rn, label

Description:

Branch if two source operands are not equal. Both operands are treated as integer values. The displacement is relative to the address of the branch instruction.

Instruction Format: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	0_2	Lk	39_7

Clock Cycles: 4

BRA – Branch Always

Description:

This instruction always branches to the target address. The target address range is ±1GB.

Formats Supported: BSR

39 … 9	8 7	6 … 0
$Target_{30..0}$	0_2	$20h_7$

Operation:

IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

BSR – Branch to Subroutine

Description:

This instruction always jumps to the target address. The address of the next instruction is stored in a link register. The target address range is ±1GB.

Formats Supported: BSR

39 9	8 7	6 0
$Target_{30..0}$	Lk_2	$20h_7$

Operation:

Lk = next IP
IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

FBEQ –Branch if Equal

FBEQ Fa, Fb, label

Description:

Branch if two source operands are equal. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as floating-point values. Positive and negative zero are considered equal. If either operand is a NaN the branch will not be taken. The displacement is relative to the address of the branch instruction.

A postfix instruction containing an immediate value may follow the branch instruction, in which case the immediate is used instead of Rb. Rb should be set to 63.

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	2_2	Lk	38_7

Clock Cycles: 4

FBNE –Branch if Not Equal

FBNE Fa, Fb, label

Description:

Branch if two source operands are not equal. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as floating-point values. Positive and negative zero are considered equal. The displacement is relative to the address of the branch instruction.

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

Formats Supported: B

39 25	24 19	18 13	12 11	10	9 8	7	6 0
$Target_{16..2}$	Rb_6	Ra_6	$T_{1..0}$	Op	2_2	Lk	39_7

Clock Cycles: 4

JMP – Jump to Target

Description:

This instruction always jumps to the target address. The target address is the sum of a register and an immediate constant.

Instruction Format: JSR

39 19	18 13	12 9	8 7	6 0
$Immediate_{24..4}$	Ra_6	$Im_{3..0}$	0_2	36_7

Operation:

PC = Ra + sign extend (Constant)

Execution Units: Branch

Exceptions: none

Notes:

JSR – Jump to Subroutine

Description:

This instruction always jumps to the target address. The target address is the sum of a register and an immediate constant. The address of the next instruction is stored in a link register.

Instruction Format: JSR

39 19	18 13	12 9	8 7	6 0
$\text{Immediate}_{24..4}$	Ra_6	$\text{Im}_{3..0}$	Lk_2	36_7

Operation:

Lk = next IP
PC = Ra + sign extend (Constant)

Execution Units: Branch

Exceptions: none

Notes:

NOP – No Operation

NOP

Description:

This instruction does not perform any operation.

Instruction Format:

39 8	7	6 0
$0xFFFFFFFF_{32}$	1	127_7

RTD – Return from Subroutine and Deallocate

Description:

This instruction returns from a subroutine by transferring program execution to the address stored in a link register plus an offset amount. Additionally, the stack pointer is incremented by the amount specified. The const field is shifted left three times before use.

Formats Supported: RTD

39 19	18 11	10 9	8 7	6 0
$Const_{21}$	$Offs_8$	2_2	Lk_2	35_7

Operation:

Execution Units: Branch

Exceptions: none

Notes:

Return address prediction hardware may make use of the RTS instruction.

RTE – Return from Exception

Description:

This instruction returns from an exception routine by transferring program execution to the address stored in an internal stack. The const field is shifted left once before use. This instruction may perform a two-up level return.

Formats Supported: RTS

39 … 11	10 9	8 7	6 … 0
$Const_{29}$	1_2	0_2	35_7

Formats Supported: RTS – Two up level return.

39 … 11	10 9	8 7	6 … 0
$Const_{29}$	1_2	1_2	35_7

Operation:

Optionally pop the status register and program counter from the internal stack. Add Const wydes to the program counter. If returning from an application trap the status register is not popped from the stack.

Execution Units: Branch

Exceptions: none

Notes:

Graphics Instructions

BLEND – Blend Colors

Description:

This instruction blends two colors whose values are in Ra and Rb according to an alpha value in Rc. The resulting color is placed in register Rt. The alpha value is a ten-bit value assumed to be a fixed-point number with one whole digit and nine fraction digits. The same alpha value should be placed in each RGB component location of Rc. The color values in Ra and Rb are assumed to be RGB10.10.10 format colors. The result is a RGB10.10.10 format color. Note that a close approximation to 1.0 – alpha is used. Each component of the color is blended independently. Component overflow saturates towards white.

Instruction Format: R3

BLEND Rt, Ra, Rb, Rc

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	0_3	Rc_6	Rb_6	Ra_6	Rt_6	89_7

Operation:

Rt.R = (Ra.R * alpha) + (Rb.R * ~alpha)

Rt.G = (Ra.G * alpha) + (Rb.G * ~alpha)

Rt.B = (Ra.B * alpha) + (Rb.B * ~alpha)

Clock Cycles: 2

TRANSFORM – Transform Point

Description:

The point transform instruction transforms a point from one location to another using a transform function. The transform function has 12 co-efficients in the form of a matrix used in the calculation.

Points are represented in 16.16 fixed-point format.

Instruction Format: R1

39 37	36 34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_3	Pr_3	Op_3	Rz_6	Ry_6	Rx_6	Rt_6	89_7

Op_3	Operation
4	Return new X
5	Return new Y
6	Return new Z
7	Set coefficient

To set a coefficient Rx specifies which coefficient to set, Ry specifies the value.

Rx	Co-efficient
0	aa
1	ab
2	ac
3	tx
4	ba
5	bb
6	bc
8	ty
9	ca
10	cb
11	cc
12	tz

Operation:

Input matrix M:

```
        | aa ab ac tx |
M =     | ba bb bc ty |
        | ca cb cc tz |
```

Input point X:

```
        | x |
X =     | y |
        | z |
        | 1 |
```

Output point X':

```
        | x' |             | aa*x + ab*y + ac*z + tx |
X' =    | y' | = MX =      | ba*x + bb*y + bc*z + ty |
        | z' |             | ca*x + cb*y + cc*z + tz |
```

Clock Cycles: 3

System Instructions

BRK – Break

Description:

This instruction initiates the processor debug routine. The processor enters debug mode. The cause code register is set to indicate execution of a BRK instruction. Interrupts are disabled. The instruction pointer is reset to the contents of tvec[3] and instructions begin executing. There should be a jump instruction placed at the break vector location. The address of the BRK instruction is stored in the EIP.

Instruction Format: BRK

39 7	6 0
00_{33}	$00h_7$

Operation:

PUSH SR
PUSH IP
IP = tvec[3]

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

IRQ – Generate Interrupt

Description:

Generate interrupt. This instruction invokes the system exception handler. The return address is stored in the EIP register (code address register #8 to 15).

The return address stored is the address of the interrupt instruction, not the address of the next instruction. To call system routines use the SYS instruction.

The level of the interrupt is checked and if the interrupt level in the instruction is less than or equal to the current interrupt level then the instruction will be ignored.

Instruction Format: EX

39 37	36 34	33 26	25 22	21 19	18 7	6 0
Lvl_3	Pr_3	$\sim_8$	0_4	$\sim_3$	$Cause_{12}$	112_7

Operation:

PUSH SR
PUSH IP
CAUSE = $Cause_{12}$
IP = tvec[3]

Execution Units: Branch

MEMDB – Memory Data Barrier

Description:

All memory accesses before the MEMDB command are completed before any memory accesses after the data barrier are started. This is an alternate mnemonic for the FENCE instruction.

Instruction Format:

31 30	29 27	26 24	23 16	15 12	11 8	7	6 0
Fmt_2	Pr_3	0_3	$255_{7..0}$	0_4	0_4	~	114_7

Clock Cycles: 1

Execution Units: Memory

MEMSB – Memory Synchronization Barrier

Description:

All instructions before the MEMSB command are completed before any memory access is started. This is an alternate mnemonic for the FENCE instruction.

Instruction Format:

31 30	29 27	26 24	23 16	15 12	11 8	7	6 0
Fmt_2	Pr_3	0_3	$192_{7..0}$	0_4	15_4	~	114_7

Clock Cycles: 1

Execution Units: Memory

PFI – Poll for Interrupt

Description:

The poll for interrupt instruction polls the interrupt status lines and performs an interrupt service if an interrupt is present. Otherwise, the PFI instruction is treated as a NOP operation. Polling for interrupts is performed by managed code. PFI provides a means to process interrupts at specific points in running software. Rt is loaded with the cause code in the low order twelve bits, and the interrupt level in bits twelve to fourteen of the register.

Instruction Format: OSR2

39 38	37 35	34 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{21..0}$	Rt_6	115_7

Clock Cycles: 1 (if no exception present)

Operation:

if (irq <> 0)
- Rt[11:0] = cause code
- Rt[14:12] = irq level
- PMSTACK = (PMSTACK << 4) | 6
- CAUSE = $Const_{12}$
- EIP = IP
- IP = tvec[3]

Execution Units: Branch

FENCE – Synchronization Fence

Description:

All instructions for a particular unit before the FENCE are completed and committed to the architectural state before instructions of the unit type after the FENCE are issued. This instruction is used to ensure that the machine state is valid before subsequent instructions are executed.

Instruction Format:

31 30	29 27	26 24	23 16	15 12	11 8	7	6 0
Fmt_2	Pr_3	Op_3	$Mask_{7..0}$	Aft_4	Bef_4	~	114_7

Mask Bit	Access		
0	Wr	Before	
1	Rd		
2	Out		
3	In		
4	Wr	After	
5	Rd		
6	Out		
7	In		

Aft_4 / Bef_4	Unit
0	MEM
1	ALU
2	FPU
3	Branch
4 to 14	Reserved
15	All units

Op3	Fence Type
0	Normal
1 to 6	reserved
7	TSO fence

REX – Redirect Exception

Description:

This instruction redirects an exception from an operating mode to a lower operating mode. This instruction if successful jumps to the target exception handler and does not return. If this instruction fails execution will continue with the next instruction.

This instruction may fail if exceptions are not enabled at the target level.

The location of the target exception handler is found in the trap vector register for that operating mode (tvec[xx]).

The cause (cause) and bad address (badaddr) registers of the originating mode are copied to the corresponding registers in the target mode.

Instruction Format: EX

31 29	28 26	25 22	21 18	17	16 12	11 9	8 7	6 0
$\sim_3$	Pr_3	7_4	$\sim_4$	~	Ra_5	$\sim_3$	Tm_2	$112h_7$

Tm_2	
0	redirect to user mode
1	redirect to supervisor mode
2	redirect to hypervisor mode
3	reserved

Clock Cycles: 4

Execution Units: Branch

Example:

```
REX 1           ; redirect to supervisor handler
; If the redirection failed, exceptions were likely disabled at the target level.
; Continue processing so the target level may complete its operation.
RTE             ; redirection failed (exceptions disabled ?)
```

Notes:

Since all exceptions are initially handled in machine mode the machine handler must check for disabled lower mode exceptions.

SYS – System Call

Description:

Perform a system call. Interrupts are disabled. The instruction pointer is reset to the contents of tvec[3] and instructions begin executing. There should be a jump instruction placed at the break vector location. The address of the SYS instruction is stored in the EIP register.

Instruction Format: BRK

39 16	15 7	6 0
$\sim_{24}$	Callno_9	0_7

Operation:

PUSH SR onto internal stack
PUSH PC onto internal stack
PC = tvec[3]

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

STOP – STOP Processor

Description:

The STOP instruction waits for an external interrupt to occur before proceeding. While waiting for the interrupt, the processor clock is slowed down or stopped placing the processor in a lower power mode. A sixteen-bit constant is provided for the CPU's stop instruction.

Instruction Format: STOP

39 38	37 35	34 13	12 7	6 0
Fmt_2	Pr_3	$Immediate_{21..0}$	Rt_6	113_7

Clock Cycles: 1 (if no exception present)

Execution Units: Branch

Macro Instructions

ENTER – Enter Routine

Description:

This instruction is used for subroutine linkage at entrance into a subroutine. First it pushes the frame pointer and return address onto the stack, next the stack pointer is loaded into the frame pointer, and finally the stack space is allocated. This instruction is code dense, replacing eight other instructions with a single instruction.

A maximum of 2GB may be allocated on the stack. An immediate postfix may not be used with this instruction. The stack and frame pointers are assumed to be r63 and r62 respectively.

Note that the constant must be a negative number and a multiple of sixteen.

Note that the instruction reserves room for two words in addition to the return address and frame pointer. One use for the extra words may to store exception handling information.

Note this instruction uses T0 as a temporary register.

Integer Instruction Format: RI

39 8	7	6 0
$\text{Constant}_{31..0}$	0	52_7

Operation:

SP = SP - 64
Memory[SP] = FP
Memory16[SP] = LR0
Memory32[SP] = 0 ; zero out catch handler address
Memory48[SP] = 0
FP = SP
SP = SP + constant

LEAVE – Leave Routine

Description:

This instruction is used for subroutine linkage at exit from a subroutine. First it moves the frame pointer to the stack pointer deallocating any stack memory allocations. Next the frame pointer and return address are popped off the stack. The stack pointer is adjusted by the amount specified in the instruction. Then a jump is made to the return address. This instruction is code dense, replacing six other instructions with a single instruction. The stack pointer adjustment is multiplied by sixteen keeping the stack pointer word aligned. A six-bit constant is added to the link register to form the return address. This allows returning up to 64 bytes past the normal return address.

Instruction Format: LEAVE

39 13	12 7	6 0
Constant$_{27}$	Cnst$_6$	53$_7$

Operation:

SP = FP
FP = Memory[SP]
T0 = Memory16[SP]
LR0 = T0
SP = SP + 64 + Constant$_{20}$ * 16
IP = LR0 + Cnst$_5$ * 2

POP – Pop Registers from Stack

Description:

This instruction pops up to four registers from the stack.

Instruction Format: POP

39 38	37 35	34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	~	N_3	Rc_6	Rb_6	Ra_6	Rt_6	55_7

Operation:

Rt = Mem[SP]
Ra = Mem[SP+16]
Rb = Mem[SP+32]
Rc = Mem[SP+48]
SP = SP + N * 16

PUSH – Push Registers on Stack

Description:

This instruction pushes up to four registers onto the stack.

Instruction Format: PUSH

39 38	37 35	34	33 31	30 25	24 19	18 13	12 7	6 0
Fmt_2	Pr_3	~	N_3	Rc_6	Rb_6	Ra_6	Rs_6	54_7

Operation:

SP = SP – N * 16
if (N > 3) $Memory_{16}$[SP+(N-3)*16] = Rc
if (N > 2) $Memory_{16}$[SP+(N-2)*16] = Rb
if (N > 1) $Memory_{16}$[SP+(N-1)*16] = Ra
if (N > 0) $Memory_{16}$[SP+N*16] = Rs

Modifiers

ATOM

Description:

Treat the following sequence of instructions as an "atom". The instruction sequence is executed with interrupts set to the specified mask level. Interrupts may be disabled for up to eight instructions. The non-maskable interrupt may not be masked.

The 24-bit mask is broken into eight three-bit interrupt level numbers. Bit 7 to 9 represent the interrupt level for the first instruction, bits 10 to 12 for the second and so on.

Instruction Format: ATOM

39 38	37 35	34 31	30 7	6 0
Fmt_2	Pr_3	$\sim_4$	$Mask_{24}$	122_7

	Mask Bit	
Modifier Scope	0 to 2	Instruction zero (always 7)
	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

Assembler Syntax:

Example:

```
ATOM "777777"
LOAD a0,[a3]
CMP t0,a0,a1
PEQ t0,"TTF"
STORE a2,[a3]
LDI a0,1
LDI a0,0
```

```
ATOM "6666"
LOAD a1,[a3]
ADD t0,a0,a1
MOV a0,a1
STORE t0,[a3]
```

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

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: REP

39 38	37 35	34 29	28 13	12 7	6 0
Fmt$_2$	Pr$_3$	~$_6$	Immediate$_{15..0}$	Rp$_6$	121$_7$

Assembler Syntax:

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 seven instructions is present.

Example:

```
PRED r2,”TTTFFFI”   ; next three execute if true, three after execute if false, one 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
```

REGS – Registers List

Description:

This instruction modifier specifies additional operands for the next instruction. When applied to a load or store operation it causes a multiple register load or store to be performed. The ‘P’ field of the instruction indicates whether to pack (0) or skip (1) over data addresses when performing the load or store operation.

Instruction Format:

63 – 9	8	7	6 – 0
$Reglist_{55}$	F	P	117_7

Bit	Reg
0 to 30	GPR r1 to r31
31 to 34	LR0 to LR3
35 to 50	Predicate Pr0 to Pr15
51	REP state
52	SSP
53	HSP
54	MSP

Assembler Syntax:

Example:

~~REP scrapped~~

~~**Description:**~~

~~This modifier indicates a short series of instructions to repeat while the loop counter condition is met. The repeat modifier includes instructions according to a count specified in the $Icnt_3$ field. The number of included instructions is one greater than $Icnt_3$. Up to eight instructions may be part of the repeat operation. The loop counter may be incremented or decremented for each repeat. Loop counter tests perform signed comparisons. The 19-bit immediate may be overridden with a constant postfix instruction. The constant postfix does not count as an instruction in the loop.~~

~~REP is limited to a 32-bit immediate value.~~

~~Context for the REP instruction is stored in a context buffer which must be saved and restored when the context changes or during interrupt processing.~~

Instruction Format: REP

39 38	37 35	34 16	15 14	13	12 10	9 7	6 0
Fmt_2	Pr_3	$Immediate_{18..0}$	Pi_2	D	$Icnt_3$	Cnd_3	120_7

Pi_2	
0	Use 19 bit constant
1	Use 32-bit postfix
2,3	reserved

D	Meaning
0	Decrement loop counter
1	Increment loop counter

Cnd_3	Loop Counter Test	
0h	Equal	LC == Imm
1h	Not equal	LC != Imm
2h	Signed less than	LC < Imm
3h	Signed less than or equal	LC <= Imm
4h	Signed greater than or equal	LC >= Imm
5h	Signed greater than	LC > Imm
6h	Bit clear	LC[imm] = 0
7h	Bit set	LC[imm] == 1

Assembler Syntax:

ROUND

Description:

Set the rounding mode for following eight instructions. Note that postfixes do not count as instructions.

Instruction Format: ATOM

39 38	37 35	34 31	30 7	6 0
Fmt_2	Pr_3	$\sim_4$	$Mask_{24}$	116_7

	Mask Bit	
Modifier Scope	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

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

Assembler Syntax:

Example:

Thor2024 Root Opcode

	0	1	2	3	4	5	6	7
0x	0 BRK SYS	1 {R1}	2 {R2}	3 SLTI	4 ADDI	5 SUBFI	6 MULI	7 CSR
	8 ANDI	9 ORI	10 EORI	11 CMPI	12 CHK	13 DIVI	14 MULUI	15 MOV
1x	16 CLR	17 SET	18 EXTU	19 EXT	20 COM	21 DIVUI	22 {R2Q}	23 DEP
	24	25	26	27	28	29	30	31
2x	32 BSR / BRA	33 DBRA	34 MCB	35 RTx	36 JSR	37 MCB	38 BEQ	39 BNE
	40 BLT BLTU	41 BGE BGEU	42 BLE BLEU	43 BGT BGTU	44 BBC	45 BBS	46 BBCI	47 BBSI
3x	48	49	50	51	52 ENTER	53 LEAVE	54 PUSH	55 POP
	56 FLDH	57 FLDS	58 FLDD	59 FLDQ	60 FSTH	61 FSTS	62 FSTD	63 FSTQ
4x	64 LDB	65 LDBU	66 LDW	67 LDWU	68 LDT	69 LDTU	70 LDO	71 LDOU
	72 LDH	73 LDM	74 LDA	75 CACHE	76 PLDS	77 PLDD	78 DFLD	79 {LDX}
5x	80 STB	81 STW	82 STT	83 STO	84 STH	85 STM	86 STPTR	87 {STX}
	88 {SHIFT}	89 BLEND	90 {DFLT2}	91 {DFLT3}	92 AMO	93 DFST	94 PSTS	95 PSTD
6x	96 {FLTQ2}	97 {FLTQ3}	98 {FLTD2}	99 {FLTD3}	100 {FLTS2}	101 {FLTS3}	102 {FLTH2}	103 {FLTH3}
	104 64 {R3}	105 {PST2I}	106 {PST2}	107 {PST3}	108 BFND	109 BCMP	110 BSET	111 BMOV
7x	112 IRQ	113 STOP	114 FENCE	115 PFI	116 ROUND	117 REGS	118	119
	120 REP	121 PRED	122 ATOM	123	124 PFX	125	126	127 NOP

{R1} Operations

	0	1	2	3	4	5	6	7
0x	0 CNTLZ	1 CNTLO	2 CNTPOP	3 ABS	4 SQRT	5 REVBIT	6 CNTTZ	7
	8 NNA_TRIG	9 NNA_STAT	10 NNA_MFACT	11	12	13	14 SM3P0	15 SM3P1
1x	16	17	18 AES64DS	19 AES64DSM	20 AES64ES	21 AES64ESM	22 AES64IM	23
	24 SHA256 SIG0	25 SHA256 SIG1	26 SHA256 SUM0	27 SHA256 SUM1	28 SHA512 SIG0	29 SHA512 SIG1	30 SHA512 SUM0	31 SHA512 SUM1

{R2} Operations

	0	1	2	3	4	5	6	7
2	0 AND	1 OR	2 EOR	3 CMP	4 ADD	5 SUB	6	7 CPUID
	8 NAND	9 NOR	10 ENOR	11 ANDC	12 ORC	13	14	15
2	16 MUL	17 DIV	18	19 MULU	20 DIVU	21 MULSU	22 DIVSU	23
	24 MULH	25 MOD	26	27 MULUH	28 MODU	29 MULSUH	30 MODSU	31
2	32 MIN	33 MAX	34 BMM	35 BMAP	36 DIF	37 CHARNDX	38 CHARNDX	39 CHARNDX
	40 NNA_MTWT	41 NNA_MTIN	42 NNA_MTBIAS	43 NNA_MTFB	44 NNA_MTMC	45 NNA_MTBC	46	47
2	48 V2BITS	49 BITS2V	50 VEX	51 VEINS	52 VGNDX	53	54 VSHLV	55 VSHRV
	56 V2BITSP	57 PBITS2V	58	59	60	61	62 VSHLVI	63 VSHRVI
2	64 AES64K1 I	65 AES64KS2	66 SM4ED	67 SM4KS	68	69	70 CLMUL	72
	72	73	74	75	76	77	78	79
2	80 SEQ	81 SNE	82 SLT	83 SLE	84 SLTU	85 SLEU	86	87
	88	89	90	91	92	93	94	95

{FLT2} Operations

	0	1	2	3	4	5	6	7
98	0 FSCALEB	1 **{FLT1}**	2 FMIN	3 FMAX	4 FADD	5 FSUB	6 FMUL	7 FDIV
	8 FSEQ	9 FSNE	10 FSLT	11 FSLE	12	13 FCMP	14 FNXT	15 FREM
	16 FSGNJ	17 FSGNJN	18 FSGNJX	19	20	21	22	23
	24	25	26	27	28	29	30	31

{FLT1} Operations

	0	1	2	3	4	5	6	7
0x	0 FABS	1 FNEG	2 FTOI	3 ITOF	4 FCONST	5	6 FSIGN	7 FSIG
	8 FSQRT	9 FCVTS2D	10 FCVTS2Q	11 FCVTD2Q	12 FCVTH2S	13 FCVTH2D	14 ISNAN	15 FINITE
1x	16 FCVTQ2H	17 FCVTQ2S	18 FCVTQ2D	19	20 FCVTH2Q	21 FTRUNC	22 FRSQRTE	23 FRES
	24	25 FCVTD2S	26	27	28	29	30 FCLASS	31
2x	32 FSIN	33 FCOS	34 FTAN					
	40	41	42 FATAN					
3x	48 FSIGMOID							
	56							

{FLT3} Operations

	0	1	2	3	4	5	6	7
99	0 FMA	1 FMS	2 FNMA	3 FNMS	4 FDP	5 FDP	6 FDP	7 FDP
	8 FAND	9 ORF	10 FEOR	11	12	13	14	15
	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31

{LDX} – Indexed Loads

	0	1	2	3	4	5	6	7
0x	0 LDBX	1 LDBUX	2 LDWX	3 LDWUX	4 LDTX	5 LDTUX	6 LDOX	7 LDOUX
	8 LDHX	9 LDMX	10 LDAX	11 CACHEX	12 PLDSX	13 PLDDX	14	15
1x	16	17	18 FLDHX	19	20 FLDSX	21	22 FLDDX	23
	24 FLDQX	25 DFLDSX	26 DFLDDX	27 DFLDQX	28 BFNDX	29	30	31 CAS
2x	32	33	34	35	36	37	38	39
	40	41	42	43	44	45	46	47
3x	48	49	50	51	52	53	54	55
	56	57	58	59	60	61	62	63

{STX} – Indexed Stores

	0	1	2	3	4	5	6	7
0x	0 STBX	1 STWX	2 STTX	3 STOX	4 STHX	5 STMX	6 STPTRX	7 PUSH
	8	9 FSTHX	10 FSTSX	11 FSTDX	12 FSTQX	13	14 PSTSX	15 PSTDX
1x	16	17	18 DFSTSX	19 DFSTDX	20 DFSTQX	21	22	23
	24	25	26	27	28	29	30	31

{AMO} – Atomic Memory Ops

	0	1	2	3	4	5	6	7
92	0 AMOADD	1 AMOAND	2 AMOOR	3 AMOEOR	4 AMOMIN	5 AMOMAX	6 AMOSWAP	7
	8 AMOASL	9 AMOLSR	10 AMOROL	11 AMOROR	12 AMOMINU	13 AMOMAXU	14	15

{PR} Thor2024 Predicate Operations

	0	1	2	3	4	5	6	7
0x	0 PRASL	1 PRROL	2 PRLSR	3 PRROR	4 ADD	5 SUB	6	7
	8 PRAND	9 PROR	10 PREOR	11 MFPR	12 MTPR	13 PRCNTPOP	14 PRFIRST	15 PRLAST
1x	16 PRANDN	17	18	19	20 PRLDI	21	22	23
	24	25	26	27	28	29	30	31

{R3} Thor2024 R3 Operations

	0	1	2	3	4	5	6	7
0x	0 MUX	1 PTRDIF	2 MIN3	3 MAX3	4 CMOVNZ	5 CMOVZ	6	7

{EX} Exception Instructions

	0	1	2	3	4	5	6	7
2	0 IRQ	1	2 FTX	3 FCX	4 FDX	5 FEX	6	7 REX
	8	9	10	11	12	13	14	15

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	32	R/W		Capabilities area		

0FF						

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 0FF	32	R/W		Capabilities area		

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.

Glossary

AMO

AMO stands for atomic memory operation. An atomic memory operation typically reads then writes to memory in a fashion that may not be interrupted by another processor. Some examples of AMO operations are swap, add, and, and or. AMO operations are typically passed from the CPU to the memory controller and the memory controller performs the operation.

Assembler

A program that translates mnemonics and operands into machine code OR a low-level language used by programmers to conveniently translate programs into machine code. Compilers are often capable of generating assembler code as an output.

ATC

ATC stands for address translation cache. This buffer is used to cache address translations for fast memory access in a system with an mmu capable of performing address translations. The address translation cache is more commonly known as the TLB.

Base Pointer

An alternate term for frame pointer. The frame or base pointer is used by high-level languages to access variables on the stack.

Burst Access

A burst access is several bus accesses that occur rapidly in a row in a known sequence. If hardware supports burst access the cycle time for access to the device is drastically reduced. For instance, dynamic RAM memory access is fast for sequential burst access, and somewhat slower for random access.

BTB

An acronym for Branch Target Buffer. The branch target buffer is used to improve the performance of a processing core. The BTB is a table that stores the branch target from previously executed branch instructions. A typical table may contain 1024 entries. The table is typically indexed by part of the branch address. Since the target address of a branch type instruction may not be known at fetch time, the address is speculated to be the address in the branch target buffer. This allows the machine to fetch instructions in a continuous fashion without pipeline bubbles. In many cases the calculated branch address from a previously executed instruction remains the same the next time the same instruction is executed. If the address from the BTB turns out to be incorrect, then the machine will have to flush the instruction queue or pipeline and begin fetching instructions from the correct address.

Card Memory

A card memory is a memory reserved to record the location of pointer stores in a garbage collection system. The card memory is much smaller than main memory; there may be card memory entry for a block of main memory addresses. Card memory covers memory in 128 to 512-byte sized blocks. Usually, a byte is dedicated to record the pointer store status even though a bit would be adequate, for performance reasons. The location of card memory to update is found by shifting the pointer value to the right some number of bits (7 to 9 bits) and then adding the base address of the table. The update to the card memory needs to be done with interrupts disabled.

Commit

As in commit stage of processor. This is the stage where the processor is dedicated or committed to performing the operation. There are no prior outstanding exceptions or flow control changes to prevent the instruction from executing. The instruction may execute in the commit stage, but registers and memory are not updated until the retire stage of the processor.

Decimal Floating Point

Floating point numbers encoded specially to allow processing as decimal numbers. Decimal floating point allows processing every-day decimal numbers rounding in the same manner as would be done by hand.

Decode

The stage in a processor where instructions are decoded or broken up into simpler control signals. For instance, there is often a register file write signal that must be decoded from instructions that update the register file.

Diadic

As in diadic instruction. An instruction with two operands.

Endian

Computing machines are often referred to as big endian or little endian. The endian of the machine has to do with the order bits and bytes are labeled. Little endian machines label bits from right to left with the lowest bit at the right. Big endian machines label bits from left to right with the lowest numbered bit at the left.

FIFO

An acronym standing for ‘first-in first-out’. Fifo memories are used to aid data transfer when the rate of data exchange may have momentary differences. Usually when fifos transfer data the average data rate for input and output is the same. Data is stored in a buffer in order then retrieved from the buffer in order. Uarts often contain fifos.

FPGA

An acronym for Field Programmable Gate Array. FPGA's consist of a large number of small RAM tables, flip-flops, and other logic. These are all connected with a programmable connection network. FPGA's are 'in the field' programmable, and usually re-programmable. An FPGA's re-programmability is typically RAM based. They are often used with configuration PROM's so they may be loaded to perform specific functions.

Floating Point

A means of encoding numbers into binary code to allow processing. Floating point numbers have a range within which numbers may be processed, outside of this range the number will be marked as infinity or zero. The range is usually large enough that it is not a concern for most programs.

Frame Pointer

A pointer to the current working area on the stack for a function. Local variables and parameters may be accessed relative to the frame pointer. As a program progresses a series of "frames" may build up on the stack. In many cases the frame pointer may be omitted, and the stack pointer used for references instead. Often a register from the general register file is used as a frame pointer.

HDL

An acronym that stands for 'Hardware Description Language'. A hardware description language is used to describe hardware constructs at a high level.

HLL

An acronym that stands for "High Level Language"

Instruction Bundle

A group of instructions. It is sometimes required to group instructions together into bundle. For instance, all instructions in a bundle may be executed simultaneously on a processor as a unit. Instructions may also need to be grouped if they are oddball in size for example 41 bits, so that they can be fit evenly into memory. Typically, a bundle has some bits that are global to the bundle, such as template bits, in addition to the encoded instructions.

Instruction Pointers

A processor register dedicated to addressing instructions in memory. It is also often called a program counter. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

Instruction Prefix

An instruction prefix applies to the following instruction to modify its operation. An instruction prefix may be used to add more bits to a following immediate constant, or to add additional register fields for the instruction. The prefix essentially extends the number of bits available to encode instructions. An instruction prefix usually locks out interrupts between the prefix and following instruction.

Instruction Modifier

An instruction modifier is similar to an instruction prefix except that the modifier may apply to multiple following instructions.

ISA

An acronym for Instruction Set Architecture. The group of instructions that an architecture supports. ISA's are sometimes categorized at extreme edges as RISC or CISC. RTF64 falls somewhere in between with features of both RISC and CISC architectures.

Keyed Memory

A memory system that has a key associated with each page to protect access to the page. A process must have a matching key in its key list in order to access the memory page. The key is often 20 bits or larger. Keys for pages are usually cached in the processor for performance reasons. The key may be part of the paging tables.

Linear Address

A linear address is the resulting address from a virtual address after segmentation has been applied.

Machine Code

A code that the processing machine is able execute. Machine code is lowest form of code used for processing and is not usually delt with by programmers except in debugging cases. While it is possible to assemble machine code by hand usually a tool called an assembler is used for this purpose.

Milli-code

A short sequence of code that may be used to emulate a higher-level instruction. For instance, a garbage collection write barrier might be written as milli-code. Milli-code may use an alternate link register to return to obtain better performance.

Monadic

An instruction with just a single operand.

Opcode

A short form for operation code, a code that determines what operation the processor is going to perform. Instructions are typically made up of opcodes and operands.

Operand

The data that an opcode operates on, or the result produced by the operation. Operands are often located in registers. Inputs to an operation are referred to as source operands, the result of an operation is a destination operand.

Physical Address

A physical address is the final address seen by the memory system after both segmentation and paging have been applied to a virtual address. One can think of a physical address as one that is “physically” wired to the memory.

Physical Memory Attributes (PMA)

Memory usually has several characteristics associated with it. In the memory system there may be several different types of memory, rom, static ram, dynamic ram, eeprom, memory mapped I/O devices, and others. Each type of memory device is likely to have different characteristics. These characteristics are called the physical memory attributes. Physical memory attributes are associated with address ranges that the memory is located in. There may be a hardware unit dedicated to verifying software is adhering to the attributes associated with the memory range. The hardware unit is called a physical memory attributes checker (PMA checker).

Posits

An alternate representation of numbers.

Program Counter

A processor register dedicated to addressing instructions in memory. It is also often and perhaps more aptly called an instruction pointer. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

Retire

As in retire an instruction. This is the stage in processor in which the machine state is updated. Updates include the register file and memory. Buffers used for instruction storage are freed.

ROB

An acronym for ReOrder Buffer. The re-order buffer allows instructions to execute out of order yet update the machine's state in order by tracking instruction state and variables. In FT64 the re-order buffer is a circular queue with a head and tail pointers. Instructions at the head are committed if done to the machine's state then the head advanced. New instructions are queued at the buffer's tail as long as there is room in the queue. Instructions in the queue may be processed out of the order that they entered the queue in depending on the availability of resources (register values and functional units).

RSB

An acronym that stands for return stack buffer. A buffer of addresses used to predict the return address which increases processor performance. The RSB is usually small, typically 16 entries. When a return instruction is detected at time of fetch the RSB is accessed to determine the address of the next instruction to fetch. Predicting the return address allows the processing core to continuously fetch instructions in a speculative fashion without bubbles in the pipeline. The return address in the RSB may turn out to be detected as incorrect during execution of the return instruction, in which case the pipeline or instruction queue will need to be flushed and instructions fetched from the proper address.

SIMD

An acronym that stands for 'Single Instruction Multiple Data'. SIMD instructions are usually implemented with extra wide registers. The registers contain multiple data items, such as a 128-bit register containing four 32-bit numbers. The same instruction is applied to all the data items in the register at the same time. For some applications SIMD instructions can enhance performance considerably.

Stack Pointer

A processor register dedicated to addressing stack memory. Sometimes this register is assigned by convention from the general register pool. This register may also sometimes index into a small dedicated stack memory that is not part of the main memory system. Sometimes machines have multiple stack pointers for different purposes, but they all work on the idea of a stack. For instance, in Forth machines there are typically two stacks, one for data and one for return addresses.

Telescopic Memory

A memory system composed of layers where each layer contains simplified data from the topmost layer downwards. At the topmost layer data is represented verbatim. At the bottom layer there may be only a single bit to represent the presence of data. Each layer of the telescopic memory uses far less memory than the layer above. A telescopic memory could be used in garbage collection systems. Normally however the extra overhead of updating multiple layers of memory is not warranted.

TLB

TLB stands for translation look-aside buffer. This buffer is used to store address translations for fast memory access in a system with an mmu capable of performing address translations.

Trace Memory

A memory that traces instructions or data. As instructions are executed the address of the executing instruction is stored in a trace memory. The trace memory may then be dumped to allow debugging of software. The trace memory may compress the storage of addresses by storing branch status (taken or not taken) for consecutive branches rather than storing all addresses. It typically requires only a single bit to store the branch status. However, even when branches are traced, periodically the entire address of the program executing is stored. Often trace buffers support tracing thousands of instructions.

Triadic

An instruction with three operands.

Vector Length (VL register)

The vector length register controls the maximum number of elements of a vector that are processed. The vector length register may not be set to a value greater than the number of elements supported by hardware. Vector registers often contain more elements than are required by program code. It would be wasteful to process all elements when only a few are needed. To improve the processing performance only the elements up to the vector length are examined.

Vector Mask (VM)

A vector mask is used to restrict which elements of a vector are processed during a vector operation. A one bit in a mask register enables the processing for that element, a zero bit disables it. The mask register is commonly set using a vector set operation.

Virtual Address

The address before segmentation and paging has been applied. This is the primary type of address a program will work with. Different programs may use the same virtual address range without being concerned about data being overwritten by another program. Although the virtual address may be the same the final physical addresses used will be different.

Writeback

A stage in a pipelined processing core where the machine state is updated. Values are 'written back' to the register file.

Miscellaneous

Reference Material

Below is a short list of some of the reading material the author has studied. The author has downloaded a fair number of documents on computer architecture from the web. Too many to list.

Modern Processor Design Fundamentals of Superscalar Processors by John Paul Shen, Mikko H. Lipasti. Waveland Press, Inc.

Computer Architecture A Quantitative Approach, Second Edition, by John L Hennessy & David Patterson, published by Morgan Kaufman Publishers, Inc. San Francisco, California is a good book on computer architecture. There is a newer edition of the book available.

Memory Systems Cache, DRAM, Disk by Bruce Jacob, Spencer W. Ng., David T. Wang, Samuel Rodriguez, Morgan Kaufman Publishers

PowerPC Microprocessor Developer's Guide, SAMS publishing. 201 West 103rd Street, Indianapolis, Indiana, 46290

80386/80486 Programming Guide by Ross P. Nelson, Microsoft Press

Programming the 286, C. Vieillefond, SYBEX, 2021 Challenger Drive #100, Alameda, CA 94501

Tech. Report UMD-SCA-2000-02 ENEE 446: Digital Computer Design — An Out-of-Order RiSC-16

Programming the 65C816, David Eyes and Ron Lichty, Western Design Centre Inc.

Microprocessor Manuals from Motorola, and Intel,

The SPARC Architecture Manual Version 8, SPARC International Inc, 535 Middlefield Road. Suite210 Menlo Park California, CA 94025

The SPARC Architecture Manual Version 9, SPARC International Inc, Sab Jose California, PTR Prentice Hall, Englewood Cliffs, New Jersey, 07632

The MMIX processor: http://mmix.cs.hm.edu/doc/instructions-en.html

RISCV 2.0 Spec, Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovi´c CS Division, EECS Department, University of California, Berkeley {waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu

The Garbage Collection Handbook, Richard Jones, Antony Hosking, Eliot Moss published by CRC Press 2012

RISC-V Cryptography Extensions Volume I Scalar & Entropy Source Instructions See github.com/riscv/riscv-crypto for more information.

Trademarks

WISHBONE Compatibility Datasheet

The Thor2021 core may be directly interfaced to a WISHBONE compatible bus.

WISHBONE Datasheet WISHBONE SoC Architecture Specification, Revision B.3		
Description:	Specifications:	
General Description:	Central processing unit (CPU core)	
Supported Cycles:	MASTER, READ / WRITE MASTER, READ-MODIFY-WRITE MASTER, BLOCK READ / WRITE, BURST READ (FIXED ADDRESS)	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing	128 bit 8 bit 128 bit Little Endian any (undefined)	
Clock frequency constraints:	tm_clk_i must be >= 10MHz	
Supported signal list and cross reference to equivalent WISHBONE signals	Signal Name: ack_i adr_o(31:0) clk_i dat_i(127:0) dat_o(127:0) cyc_o stb_o wr_o sel_o(7:0) cti_o(2:0) bte_o(1:0)	WISHBONE Equiv. ACK_I ADR_O() CLK_I DAT_I() DAT_O() CYC_O STB_O WE_O SEL_O CTI_O BTE_O
Special Requirements:		