THOR CORE2022 GUIDE

[Document subtitle]

ABSTRACT

Details for the Thor2022 processing core including programming model, memory management and instruction set architecture.

Robert Finch

[Course title]

Table of Contents

Overview

Thor is a powerful 64-bit superscalar processor that represents a generational refinement of processor architecture. The processor contains 64, 64 bit general purpose integer registers. Thor uses variable length instructions varying between two and eight bytes in length and handles 8, 16, 32, and 64 bit data within a 64 bit address space.

History

Thor2021 is a work in progress beginning in October 2021. Thor2021 originated from Thor which originated from RiSC-16 by Dr. Bruce Jacob. RiSC-16 evolved from the Little Computer (LC-896) developed by Peter Chen at the University of Michigan. See the comment in Thor2021.v. The author has tried to be innovative with this design borrowing ideas from many other processing cores.

Thor2021's graphics engine originate from the ORSoC GFX accelerator core posted at opencores.org by Per Lenander, and Anton Fosselius.

Design Objectives

This processor is somewhat pedantic in nature and targeted towards high performance operation as a general-purpose processor. Following are some of the criteria that were used on which to base the design.

- Designed for Superscalar operation - the ability to execute more than one instruction at a time. To achieve high performance it is generally accepted that a processor must be able to execute more than a single instruction in any given clock cycle.
- Support for vector operations.
- Simplicity - architectural simplicity leads to a design that is easy to implement resulting in reliability and assured correctness along with easy implementation of supporting tools such as compilers. Simplicity also makes it easier to obtain high performance and results in lower overall cost.
- Extensibility - the design must be extensible so that features not present in the first release can easily be added at a later date.
- Low Cost

This design meets the above objectives in the following ways. The instruction set has been designed to minimize the interactions between instructions, allowing instructions to be executed as independent units for superscalar operation. There are a sufficient number of registers to allow the compiler to schedule parallel processing of code. A reasonably large general purpose register set is available making the design reasonably compatible with many existing compilers and assemblers.

Where needed, additional specialized instructions have been added to the processor to support a sophisticated operating system and interrupt management.

Motivation

The author wanted an FPGA based processing core for experimental purposes.

Differences from the Original

The string instructions in the original Thor ISA are no longer present.

Stack operations have been removed.

Mnemonics changed for load and store instructions.

Byte, wyde and UTF21 search instructions have been added.

Support for decimal floating-point has been added.

Support for cryptographic accelerator functions has been added.

Case Comparison Hi-lites

Some of the more striking points of a handful of architectures are compared to what is available in Thor2021. The author has studied and used several of these processors. Thor2021 has many influences from prior designs.

Case Comparison 6502

6502 vs Thor2021

Overview

This is a bit of an apples to oranges comparison as the two designs are for different environments. The 6502 was designed for a much smaller operating environment and is extremely frugal with transistor usage. The Thor2021 was designed as 64-bit processor used for experimentation in a much larger environment.

Instruction Format

The 6502 as a byte-oriented design has a compact variable instruction length encoding. Many instructions are encoded using an average of about two bytes.

While variable sized instructions offer great advantage for code density, they add complexity to the processing core. Thor2021 also uses a variable size instruction encoding. As such for a given single instruction it requires roughly twice the memory of a 6502. However, the instructions in the Thor2021 operate on 64-bit values, to perform the same operations in the 6502 would require many more bytes. Several instructions in the Thor2021 are more powerful than what can be found in the 6502.

Registers

The Thor2021 has many more registers than the 6502. It is a general-purpose register-oriented design while the 6502 is accumulator oriented. A register file of about 32 registers has been found to be a good match to many computing environments. This is somewhat of a historical determination. The Thor2021 has available many more transistors than were available to the 6502 design. The Thor2021 has many special purpose registers. The 6502 does not have any.

Instructions

The 6502 uses relative branches to allow a code dense instruction encoding. Thor2021 also uses relative branches to help reduce the instruction size. It has a larger branch displacement than the 6502 as that is what could be encoded easily.

The 6502 offers only basic instructions (ADD, SUB, CMP, AND, ORA, EOR, LDA, STA) as examples. There are no complex instructions in the 6502 ISA. All instructions execute within a handful of clock cycles. the Thor2021 has a ton of instructions compared to a 6502. It supports floating point and posit arithmetic.

The 6502 is an accumulator-based architecture that allows one memory-based operand for most instructions. Thor2021 is register based and the only instructions accessing memory are load and store type instructions.

Case Comparison 6809

Overview

The 6809 microprocessor was designed to run more powerful software than earlier micro-controllers. As such it has a couple of more registers than the 6502 for instance, and separate user mode and system mode stack pointers.

Instruction Format

6809 instructions have one or two opcode bytes followed by operands. Most instructions use only a single byte opcode. The instruction encoding is variable length.

Case Comparison 68000

Overview

The 68000 is a 16/32-bit design.

Instruction Format

The 68000 uses 16-bit instruction parcels. All instructions are a multiple of two bytes in length. The instruction encoding is variable length.

Registers

The 68000 has 16 general purpose registers split into eight data and eight address registers. There is a separate stack pointer for user and system mode.

Case Comparison ARM

Overview

The ARM architecture has become extremely popular.

Instruction Format

The ARM machine was originally a 32-bit fixed instruction format machine. It has had added onto it 16-instruction formats.

Registers

The program counter is referenced using one the registers codes available for general purpose registers.

The author is not fond of architectures that use a general-purpose register as the program counter. He believes a separate register is a better approach. Having the pc as part of the general register file is archaic.

Case Comparison RISCV

RISCV vs Thor2021

Instruction Format

While variable sized instructions offer great advantage for code density, they add complexity to the processing core.

In RISCV support for 16-bit compressed instructions consumes two opcode bits, and opcode bits are valuable. The use of these two bits and the reduction of the opcode space for other instructions is an excellent trade-off. Compressed instructions can improve code density by about 25% or more and consequently make better use of the cache. There is only the occasional instruction that can not be encoded using two fewer encoding bits, so only a very small percentage would be gained back in code density by having two more bits available. Thor2021 uses a variable length instruction encoding which allow it to achieve code density similar to RISCV.

The JAL instruction in RISCV allows any register to be used to store the return address. In practice only one or two registers which are fixed by the ABI are used. This means that there are about four bits of opcode space wasted for unnecessary register specification. Making use of these extra four bits is extremely valuable. The Thor2021 design only requires two bits to specify the return address register. The presence of four extra bits to specify the target address makes absolute addressing appealing for this design.

To build constants the LUI instruction is used. In RISCV the LUI instruction allows any register to be used as the target and has a 20-bit constant field because of encoding constraints. In practice it is possible to get by using only one or two registers to build constants with. Thor2021 has more direct support for constants larger than 32 bits. It makes use of ADDIS (add immediate shifted), ORIS, and ANDIS instructions to build 64-bit constants. These instructions support building 64-bit constants directly. RISCV does not really provide much for building constants over 32 bits.

Instructions

RISCV does not include indexed addressing modes in the standard implementation. Indexed addressing is accomplished when required using additional instructions and registers to calculate the effective address. Thor2021 directly supports indexed addressing with an optionally scaled index register. When indexed addressing is required Thor2021 is more code dense than RISCV. However indexed addressing is not used that often.

RISCV accesses memory and I/O exclusively using load and store instructions. Thor2021 has several additional instructions which access memory and I/O.

Register File

RISCV does almost everything using general-purpose registers. This paradigm increases the pressure on the register file. In the Thor2021 design there are more register files involved. Effectively, there are a few more additional registers which reduce the pressure on the general-purpose register file. There is a trend to place some global variables in the register file for performance reasons. These variables include operating vars for garbage collection, pointers to global and thread data and pointers for exception handling.

One reason to use more register files is that in a superscalar design it may allow more instructions to be committed at the same time. There is usually a limit on the number of write ports to the general register file. This limit affects how many instructions can be committed at once. By providing separate register files for some operations it effectively increases the number of write ports available making it possible to commit more instructions per cycle.

Return Address Registers

There is not a requirement for more than a couple of return address registers. The instruction set may be refined to allow only a single bit to specify the return address register.

Compare Results Registers

RISCV stores comparison results if needed in general-purpose registers. It has just a single instruction (SLT) dedicated to generating compare results. RISCV makes use of branches that compare-and-branch encoded in a single instruction. This is effective at removing the need for most compare operations. The intermediate result of the compare is hidden in the architecture; there is no need for visible compare results registers. There is still a need for the computed result of a compare operation. Sometimes software records the comparison result for later usage. For example, there may be a line of code: x = y > 10. Which will set x true if y is greater than 10.

Compares are tightly coupled to branch operations. Some architectures like RISCV compare and branch in a single instruction. Other architectures use a flags register or several flags registers. Yet other architectures simply use the general-purpose registers.

One reason to use a separate group of compare results registers is that in a superscalar design it may allow more instructions to be committed at the same time. There is usually a limit on the number of write ports to the general register file. This limit affects how many instructions can be committed at once. By providing separate register files for some operations it effectively increases the number of write ports available making it possible to commit more instructions per cycle.

Operating modes.

This design uses four operating modes. It has the RISCV operating modes. The author has seen a comment to the effect that debug on a RISCV processor really acts like an additional mode.

Memory Management

RISCV offers several memory management options including several different paging arrangements and a couple of optional base and bound registers.

Case Comparison MMIX

Instruction Format

MMIX comes across as more of a pedantic processor design. MMIX instructions are structure simply for the most part using a 32-bit format divided into four-byte regions. The author assumes this is primarily to enhance the readability of instructions. The constant field is often limited to eight bits. Thor2021 has fewer registers and that allows more constant bits to be encoded in the same size instruction.

Register File

MMIX has a 256-entry register file. It is not clear that this number of registers has any benefit over a 32-register design, but it makes the instruction format clear and easy to understand which may be a goal for a processor used for academic purposes.

Instructions

There are a lot of conditional move instructions in the MMIX ISA. Thor2021 currently supports only a single conditional move instruction.

Case Comparison PowerPC

Instruction Format

The PowerPC uses a fixed 32-bit instruction format.

Instructions

The PowerPC supports indexed addressing like the Thor2021 although index scaling is not present. The author has found indexed addressing makes up about 3% of instructions and scaled indexes a much smaller percentage.

Registers

The PowerPC has a dedicated link register and eight condition code registers. Thor2021 has with a pair of link registers dedicated in the GPR file. The PowerPC also has a loop count register used for counted loops. Thor2021 also has a loop count register.

Case Comparison x86

Registers

The x86 series has a register file that is accessible in subparts. Parts of a single register may be referred to instructions. For example, EAX is a 32-bit register that is also accessible as AL for byte operations. This has no-doubt complicated the x86 design. This contrasts with Thor2021 and many RISC designs where the registers are always manipulated as whole units.

Case Comparison SPARC

Registers

The SPARC machine uses register windowing, where a subset of registers is available from a much larger set that is “windowed”. In the SPARC the subset register window scrolls up and down automatically during subroutine calls and returns. The idea was to improve performance by not having to stack and unstack registers to memory during subroutine operations. However, with a good modern optimizing compiler the performance level of the SPARC is not much different than that of other architectures.

Nomenclature

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

Number of Bits		Instructions
8	byte	LDB, STB
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 instruction pointer or PC register.

Endian

Thor2021 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 Thor2021 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 21	20 15	14 9	8	7 0
Constant$_{11}$	Ra$_6$	Rt$_6$	v	Opcode$_8$

Development Aspects

Device Target

The core has been developed with FPGA usage in mind. In particular it is expected that the register file is built out of block memories.

Implementation Language

The core is implemented in the System Verilog language primarily for its ability to process array objects. Much of the core is plain vanilla Verilog code.

Programming Model

General Registers

There are 64 general purpose registers. General purpose registers are 64 bits wide. The general register file is unified and may hold integer or floating-point values.

If register #0 is used as the Ra register for loads, stores, or the ADD immediate instruction then the value zero is used instead, otherwise r0 is a general-purpose register.

r0	
r1	return value / arg0
r2	return value / arg1
r3	temporary register caller save
r4	temporary register
r5	temporary register
r6	temporary register
r7	temporary register
r8	temporary register
r9	temporary register
r10	temporary register
r11	register var callee save
r12	register var
r13	register var
r14	register var
r15	register var
r16	register var
r17	register var
r18	register var
r19	function argument
r20	function argument
r21	function argument
r22	function argument
r24	function argument
r25	function argument
r26	Loop Counter (LC)
r27	Global Data Pointer (g2)
r28	Global Data Pointer (g1)
r29	Global Data Pointer (g0)
r30	Frame Pointer
r31	Stack Pointer

CA0	available
CA1	return address
CA2	milli-code return address
CA3	available
CA4	available
CA5	available
CA6	available
CA7	Instruction pointer, read only
CA8 to CA15	Exceptioned Instruction Pointer

PTA	Page table address
KYT	
TR	Task Register

DBAD0	address #0	
DBAD1	address #1	
DBAD2	address #2	
DBAD3	address #3	
DBCTRL	Debug Control	
DBSTAT	Debug Status	

Register Tags

For the bypassing network, commonly used registers have a register tag associated with them. The register tag for the general registers varies from 1 to 31 corresponding to registers 1 to 31. Vector registers use tags 32 to 63. Other registers use additional tags as noted in the text.

Stack and Frame Pointers

Although the stack and frame pointer registers may be used with any instruction the core has special hardware to detect stack bounds violations by either the stack pointer or frame pointer. The stack and frame pointer registers should be kept aligned on octa-byte boundaries. That is, they should be a multiple of eight, which has the least significant three bits as zero. There is currently no hardware in the core to enforce alignment.

The author considered having the stack pointer as an independent register but that would require replicating a number of instructions (add, sub, and, or, etc.) just for the stack pointer. The author feels it is better to keep the stack pointer general-purpose in nature so that it may leverage the usage of the existing instruction set. This design is primarily a load / store architecture.

Loop Count

The loop count register, register r26 is used with the DBRA branch instruction to form counted loops. It may be automatically decremented and tested when the branch instruction is executing.

Code Address Registers

Thor2022 has eight code address registers C0 to C7. These are also referred to as branch registers in other architectures. C7 refers to the current instruction pointer.

Code Address Register Format

A code address register is composed of a 112-bit virtual address field, and a 12-bit micro-code IP field. Four bits are reserved for future use. The micro-code IP field is needed to record the micro-code IP for interrupts. The value of this field is normally zero.

127 116	115 112	111 0
MicroIP		Offset$_{112}$

Reg #		Usage
0	available for use	zero by convention
1	Subroutine return address	dedicated for subroutine linkage
2	Milli-code return address	dedicated for subroutine linkage
3	available for use	
4	available for use	
5	available for use	
6	available for use	
7	Instruction Pointer	dedicated to instruction addressing

8 to 15	Exception Instruction Pointer	stack of exception instruction pointers

The presence of multiple code address registers allows multi-level return addresses to be used for performance. Leaf routines may use C1 as the return address. Next to leaf routines may use C2, etc. So that memory operations are avoided when implementing subroutine call and return.

The instruction pointer register is read-only. The instruction pointer cannot be modified by moving a value to this register.

CA registers eight to fifteen are not accessible via jump or branch instructions. They are used as a stack of excepted instruction pointers and used by the RTI instruction.

Instruction Pointer

The instruction pointer, IP, points to the currently executing instruction. The lower 24-bits of the instruction pointer increment as instructions are processed. Branch instructions normally manipulate only the low order 24 bits of the instruction pointer. The entire pointer may be set using a branch-to-register instruction.

To conserve hardware and improve performance of the counter only the low order 24-bits increment. It is extremely rare to have 16MB or more of code without resets of the entire instruction pointer due to subroutine calls.

111 24	23 0
IP High	IP Low

Link Registers

Related to the instruction pointer are subroutine linkage registers. The architecture has two link registers for storing subroutine return addresses.

While many architectures have only a single link register, it is sometimes useful to have a second link register, for instance to implement milli-code routines. Some architectures allow any general- purpose register to be used for subroutine linkage. Often the ABI specifies that a specific register is used for this purpose. The author feels that supporting any GPR as a link register wastes instruction encoding bits that are better used for other purposes.

	127 0
Lk1	Return Address
Lk2	Return Address

Selector Registers

Selector registers are a piece of the segmented memory management. They are a short form for segment descriptors which they represent. There are 11 selector registers in the architecture. Several are dedicated to specific uses.

Reg	Tag	Usage
	…	

LDT	153	refers to address and size of local descriptor table
KYT	154	refers to address and size of memory key table
TCB	155	refers to address and size of current TCB

General Purpose Vector (v0 to v31) / Registers

v0 always has the value zero.

Register	Description / Suggested Usage	Saver
v0	always reads as zero (hardware)	
v1-v31		

Mask Registers (m0 to m7)

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

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

Vector Length (VL register)

The vector length register controls how many elements of a vector are processed. The vector length register may not be set to a value greater than the number of elements supported by hardware. After the vector length is set a SYNC instruction should be used to ensure that following instructions will see the updated version of the length register.

Vector length has register tag #87.

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

Extend Precision Carry Registers

The carry registers are used while performing extended precision arithmetic. They hold an intermediate carry value to be input to a following instruction. In normal circumstances, these registers need not be saved or restored as part of the context. The carry registers are 128-bits wide to allow extended precision on shifts and other operations.

Reg	
P0	always zero
P1	general usage
P2	"
P3	"

Summary of Special Purpose Registers

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

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

63 4	3	2	1	0
~	mie	hie	sie	uie

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

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

[U/S/H/M]_SCRATCH – CSR 0x?041

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

[U/S/H/M]_TIME (0x?FE0)

The TIME register corresponds to the wall clock real time. This register can be used to compute the current time based on a known reference point. The register value will typically be a fixed number of seconds offset from the real wall clock time. The lower 32 bits of the register are driven by the tm_clk_i clock time base input which is independent of the cpu clock. The tm_clk_i input is a fixed frequency used for timing that cannot be less than 10MHz. The low order 32 bits represent the fraction of one second. The upper 32 bits represent seconds passed. For example, if the tm_clk_i frequency is 100MHz the low order 32 bits should count from 0 to 99,999,999 then cycle back to 0 again. When the low order 32 bits cycle back to 0 again, the upper 32 bits of the register is incremented. The upper 32 bits of the register represent the number of seconds passed since an arbitrary point in the past.

Note that this register has a fixed time basis, unlike the TICK register whose frequency may vary with the cpu clock. The cpu clock input may vary in frequency to allow for performance and power adjustments.

U_FSTAT - CSR 0x0014 Floating Point Status and Control Register

The floating-point status and control register may be read using the CSR instruction. Unlike other CSR's the control register has its own dedicated instructions for update. See the section on floating point instructions for more information.

Bit		Symbol	Description
63:53			reserved
52		inexact	inexact
51		dbz	divide by zero
50		under	underflow
49		over	overflow
48		invop	invalid operation
47		~	reserved
46:44	**RM**	rm	rounding mode
43	**E5**	inexe	- inexact exception enable
42	**E4**	dbzxe	- divide by zero exception enable
41	**E3**	underxe	- underflow exception enable
40	**E2**	overxe	- overflow exception enable
39	**E1**	invopxe	- invalid operation exception enable
38	**NS**	ns	- non standard floating point indicator
Result Status			
32		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
31	**RA**	rawayz	rounded away from zero (fraction incremented)
30	**SC**	C	denormalized, negative zero, or quiet NaN
29	**SL**	neg <	the result is negative (and not zero)
28	**SG**	pos >	the result is positive (and not zero)
27	**SE**	zero =	the result is zero (negative or positive)
26	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
21 to 25			reserved
20	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
19	**X5**	inerx	- inexact result exception occurred (sticky)
18	**X4**	dbzx	- divide by zero exception occurred
17	**X3**	underx	- underflow exception occurred
16	**X2**	overx	- overflow exception occurred
15	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
14	**GX**	gx	- global exception indicator – set if any enabled exception has happened
13	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
8 to 12			reserved
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions

4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

S_PTBR – CSR 0x1003

This register contains the base address of the page table, which must be a multiple of 4096. Also included in this register is table parameters depth and type. Register tag #152.

127 64	63 16	15 11	10 8	7 6	5	4	3 1	0
~	Page Table Address$_{63..16}$	~$_5$	Levels	AL$_2$	~	S	~	Type

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

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

Levels are ignored for the inverted page table.

AL$_2$: TLB entry replacement algorithm, 0=fixed,1=LRU,2=random,3=reserved

S_ASID – CSR 0x101F

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

S_KEYS – CSR 0x1020 to 0x1027

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

			31 0
1020			key0
1021			key1
…			…
1027			key7

CA0,CA1,…CA15 (CAREGS) – M_CSR 0x3100 to 0x310F

This set of registers allows access to the code address register array. Since code address registers are larger than 64-bits they are split across two register locations for each code address register.

Reg #	Name	Alternate Name	Reg Tag
0x3100	CA0		64
…			…
0x3106	CA6		70
0x3107	CA7	IP	71
0x3108	CA8	EIP	72
…			
0x310F	CA15		79

M_CR0 (CSR 0x3000) Control Register Zero

This register contains miscellaneous control bits including a bit to enable protected mode.

Bit		Description
0	Pe	Protected Mode Enable: 1 = enabled, 0 = disabled
8 to 13		
16		
30	DCE	data cache enable: 1=enabled, 0 = disabled
32	BPE	branch predictor enable: 1=enabled, 0=disabled
33	BTBE	branch target buffer enable: 1=enabled, 0=disabled, default=1
34	WBM	write buffer merging enable: 1 = enabled, 0 = disabled
35	SPLE	speculative load enable (1 = enable, 0 = disable) (0 default)
36		
63	D	debug mode status. this bit is set during an interrupt routine if the processor was in debug mode when the interrupt occurred.

This register supports bit set / clear CSR instructions.

DCE

Disabling the data cache is useful for some codes with large data sets to prevent cache loading of values that are used infrequently. Disabling the data cache may reduce security risks for some kinds of attacks. The instruction cache may not be disabled. Enabling / disabling the data cache is also available via the CACHE instruction.

BPE

Disabling branch prediction will significantly affect the cores performance but may be useful for debugging. Disabling branch prediction causes all branches to be predicted as not-taken. No entries will be updated in the branch history table if the branch predictor is disabled.

WBM bit

Merging of values stored to memory may be disabled by setting this bit. On reset write buffer merging is disabled because it is likely desirable to setup I/O devices. Many I/O devices require updates to individual bytes by separate store instructions. (Write buffer merging is not currently implemented).

SPLE

Enabling speculative loads give the processor better performance at an increased security risk to meltdown attacks.

M_HARTID (CSR 0x3001)

This register contains a number that is externally supplied on the hartid_i input bus to represent the hardware thread id or the core number.

M_TICK (CSR 0x3002)

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

M_SEED (CSR 0x3003)

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

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

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

M_BADADDR (CSR 0x3007)

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

M_BAD_INSTR (CSR 0x300B)

This register contains a copy of the exceptioned instruction.

M_DBADx (CSR 0x3018 to 0x301B) Debug Address Register

These registers contain selectors for addresses of instruction or data breakpoints. The registers may also be used as trace triggering address registers.

31 24	23	22 0
PL_8	T	$Index_{23}$

M_DBCR (CSR 0x301C) Debug Control Register

This register contains bits controlling the circumstances under which a debug interrupt will occur.

bits			
3 to 0	Enables a specific debug address register to do address matching. If the corresponding bit in this register is set and the address (instruction or data) matches the address in the debug address register then a debug		

	interrupt will be taken.		
17, 16	This pair of bits determine what should match the debug address register zero in order for a debug interrupt to occur. 17:16 – 00 – match the instruction address 01 – match a data store address 10 – reserved 11 – match a data load or store address		
19, 18	reserved		
23 to 20	Same as 16 to 19 except for debug address register one.		
27 to 24	Same as 16 to 19 except for debug address register two.		
31 to 28	Same as 16 to 19 except for debug address register three.		
32 to 35	Trace enable on address register		
36	Enable branch compression for trace.		
55 to 62	These bits are a history stack for single stepping mode. An exception will automatically disable single stepping mode and record the single step mode state on stack. Returning from an exception pops the single step mode state from the stack.		
63	This bit enables SSM (single stepping mode)		

M_DBSR (CSR 0x301D) - Debug Status Register

This register contains bits indicating which addresses matched. These bits are set when an address match occurs and must be reset by software.

bit	
0	matched address register zero
1	matched address register one
2	matched address register two
3	matched address register three
63 to 4	not used, reserved

M_TVEC – CSR 0x3030 to 0x3037

These registers contain the address of the exception handling routine for a given operating level. TVEC[3] (0x3036) 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. The low half of the register contains the offset of the exception processing routine. The high half of the register contains the selector value of the exception processing routine.

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] low
0x3031	TVEC[0] high
…	
0x3036	TVEC[3] low
0x3037	TVEC[3] high

M_PLSTACK – CSR 0x303F

This register contains an eight-entry stack of privilege levels. Each stack entry occupies eight bits. When an exception occurs, this register is shifted to the left by eight bits and the low order bits are set to the highest privilege level, 0xFF. When an RTI instruction is executed this register is shifted to the right by eight bits restoring the previous privilege level.

M_PMSTACK – CSR 0x3040

This register contains an eight-entry operating mode and interrupt mask stack. When an exception or interrupt occurs, this register is shifted to the left by eight bits and the low order bits are set according to the exception mode, when an RTI instruction is executed this register is shifted to the right by eight bits. On RTI the last stack entry is set to $3F masking all interrupts on stack underflow. The low order eight bits represent the current operating mode and interrupt mask.

63 8	7 6	5 4	3 1	0
<seven more groups>	$\sim_2$	OM	IPL	IM

OM = operating mode, 0 to 3
IPL = interrupt priority level
IM = interrupt mask

M_SCRATCH – CSR 0x3041

This is a scratchpad register. Useful when processing exceptions.

D_VSTEP – CSR 0x4046

This register holds the current vector step number. It may need to be saved and restored during exception processing to ensure vector operations work as expected.

D_VTMP – CSR 0x4047

This register holds state for an internal temporary register used during vector processing. This register may need to be saved and restored during exception processing.

M_KYT – CSR 0x3053

The KYT register holds the selector for the memory key table. Register tag #154.

31 24	23	22 0
PL_8	T	$Index_{23}$

M_TCB - CSR 0x3054

This register holds the selector for the currently active task control block. Register tag #155.

31 24	23	22 0
PL_8	T	$Index_{23}$

M_TIME – CSR 0x?FE0

The TIME register corresponds to the wall clock real time. This register can be used to compute the current time based on a known reference point. The register value will typically be a fixed number of seconds offset from the real wall clock time. The lower 32 bits of the register are driven by the tm_clk_i clock time base input which is independent of the cpu clock. The tm_clk_i input is a fixed frequency used for timing that cannot be less than 10MHz. The low order 32 bits represent the fraction of one second. The upper 32 bits represent seconds passed. For example, if the tm_clk_i frequency is 100MHz the low order 32 bits should count from 0 to 99,999,999 then cycle back to 0 again. When the low order 32 bits cycle back to 0 again, the upper 32 bits of the register is incremented. The upper 32 bits of the register represent the number of seconds passed since an arbitrary point in the past.

This register is available to all operating modes, however it may be written only from machine mode.

Note that this register has a fixed time basis, unlike the TICK register whose frequency may vary with the cpu clock. The cpu clock input may vary in frequency to allow for performance and power adjustments.

M_TIMECMP – CSR 0x3FE1

If the wall-clock time TIME register is equal to the value in this register, then an interrupt will be generated.

Hardware Queues

There are sixteen hardware FIFO queues. Queue #15 is used to implement instruction tracing. The queues are accessible with the PUSHQ, POPQ, PEEKQ, and STATQ system instructions.

Queue #	Usage
0	Graphics
…	
15	Instruction Trace

Operating Modes

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

A subset of instructions is limited to machine mode.

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

Exceptions

External Interrupts

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

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

Polling for Interrupts

To support code that needs to run with interrupts disabled an interrupt polling instruction (PFI) is provided in the instruction set. For instance, the system could be running a high priority task with interrupts disabled. There may be sections of code where it is possible to process an interrupt however. In some code environments, it is not enough to disable and enable interrupts around critical code. The code must be effectively run with interrupt disabled all the time. This makes it necessary to poll for interrupts in software. For instance, stack prologue code may cause false pointer matches for the garbage collector because stack space is allocated before the contents are defined. If the GC scan occurs on this allocated but undefined area of memory, there could be false matches.

Effect on Machine Status

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

Exception Stack

The current register set, operating mode and interrupt enable bits are pushed onto an internal stack when an exception occurs. This stack is only eight entries deep as that is the maximum amount of nesting that can occur. Further nesting of exceptions can be achieved by saving the state contained in the exception registers.

Exception Vectoring

Exceptions are handled through a vector table. The vector table has four entries, one for each operating level the core may be running at. The location of the vector table is determined by TVEC[3]. If the core is operating at mode three for instance and an interrupt occurs vector table address number three is used for the interrupt handler. Note that the interrupt automatically switches the core to operating mode three. An exception handler at the machine level may redirect exceptions to a lower-level handler identified in one of the vector registers. More specific exception information is supplied in the cause register.

Operating	Address (If TVEC[3]	

Level	contains $F…FC0000)	
0	$F…FC0000	Handler for operating level zero
1	$F…FC0020	
2	$F…FC0040	
3	$F…FC0060	

Reset

Reset is treated as an exception. The reset routine should exit using an RTI instruction. The PL ad PM stack registers must be set appropriately for the return.

The core begins executing instructions at address $F…FD0000. All registers are in an undefined state.

Precision

Exceptions in Thor2021 are precise. They are processed according to program order of the instructions. If an exception occurs during the execution of an instruction, then an exception field is set in the reorder buffer. The exception is processed when the instruction commits which happens in program order. If the instruction was executed in a speculative fashion, then no exception processing will be invoked unless the instruction makes it to the commit stage.

Exceptions During Vector Operations

If an exception occurs during a vector operation, the vector state is recorded, and the operation aborted. The vector operation may be continued after the exception is processed. The vector state consists of a step number and a temporary register.

Exception Cause Codes

The following table outlines the cause code for a given purpose. These codes are specific to Thor2021. Under the HW column an ‘x’ indicates that the exception is internally generated by the processor; the cause code is hard-wired to that use. An ‘e’ indicates an externally generated interrupt, the usage may vary depending on the system.

Cause Code		HW	Description	
0			no exception	
1	IBE	x	instruction bus error	
2	EXF	x	Executable fault	
4	TLB	x	tlb miss	
			FMTK Scheduler	
128		e		
129	KRST	e	Keyboard reset interrupt	
130	MSI	e	Millisecond Interrupt	

131	TICK	e		
156	KBD	e	Keyboard interrupt	
157	GCS	e	Garbage collect stop	
158	GC	e	Garbage collect	
159	TSI	e	FMTK Time Slice Interrupt	
3			Control-C pressed	
20			Control-T pressed	
26			Control-Z pressed	
32	SSM	x	single step	
33	DBG	x	debug exception	
34	TGT	x	call target exception	
35	MEM	x	memory fault	
36	IADR	x	bad instruction address	
37	UNIMP	x	unimplemented instruction	
38	FLT	x	floating point exception	
39	CHK	x	bounds check exception	
40	DBZ	x	divide by zero	
41	OFL	x	overflow	
47				
48	ALN	x	data alignment	
49	KEY	x	memory key fault	
50	DWF	x	Data write fault	
51	DRF	x	data read fault	
52	SGB	x	segment bounds violation	
53	PRIV	x	privilege level violation	
54	CMT	x	commit timeout	
55	BT	x	branch target	
56	STK	x	stack fault	
57	CPF	x	code page fault	
58	DPF	x	data page fault	
59	LVL	x	level error	
60	DBE	x	data bus error	
61	PMA	x	physical memory attributes check fail	
62	NMI	x	Non-maskable interrupt	
63	BRK		BRK instruction encountered	
200	PFX	x	Too many instruction prefixes	
225	FPX_IOP	x	Floating point invalid operation	
226	FPX_DBZ	x	Floating point divide by zero	
227	FPX_OVER	x	floating point overflow	
228	FPX_UNDER	x	floating point underflow	
229	FPX_INEXACT	x	floating point inexact	
231	FPX_SWT	x	floating point software triggered	

239			Software exception handling	
240	SYS		Call operating system (FMTK)	
241			FMTK Schedule interrupt	
242	TMR	x	system timer interrupt	
243	GCI	x	garbage collect interrupt	
253	RST	x	reset	
254	NMI	x	non-maskable interrupt	
255	PFI		reserved for poll-for-interrupt instruction	

DBG

A debug exception occurs if there is a match between a data or instruction address and an address in one of the debug address registers.

IADR

This exception is currently not implemented but reserved for the purpose of identifying bad instruction addresses. If the two least significant bits of the instruction address are non-zero then this exception will occur.

UNIMP

This exception occurs if an instruction is encountered that is not supported by the processor. It may also occur if there is an attempt to use an instruction in a mode that does not support it.

OFL

If an arithmetic operation overflows (multiply, add, or shift) and the overflow exception is enabled in the arithmetic exception enable register then an OFL exception will be triggered.

KEY

This fault will occur if an attempt is made to access memory for which the app does not have the key.

FLT

A floating-point exception is triggered if an exceptional condition occurs in the floating-point unit and the exception is enabled. Please see the section on floating-point for more details.

DRF, DWF, EXF

Data read fault, data write fault, and execute fault are exceptions that are returned by the memory management unit when an attempt is made to access memory for which the corresponding access type is not allowed. For instance, if the memory page is marked as non-executable an attempt is made to load the instruction cache from the page then an execute fault EXF exception will occur.

CPF, DPF

The code page fault and data page fault exceptions are activated by the mmu if the page is not present in memory. Access may be allowed but simply unavailable. These faults are not currently implemented.

PRIV

Some instructions and CSR registers are legal to use only at a higher operating level. If an attempt is made to use the privileged instruction by a lower operating level, then a privilege violation exception may occur. For instance, attempting to use RTI instruction from user operating level.

STK

If the value loaded into one of the stack pointer registers (the stack pointer sp or frame pointer fp) is outside of the bounds defined by the stack bounds registers, then a stack fault exception will be triggered.

DBE

A timeout signal is typically wired to the err_i input of the core and if the data memory does not respond with an ack_i signal fast enough an error will be triggered. This will happen most often when the core is attempting to access an unimplemented memory area for which no ack signal is generated. When the err_i input is activated during a data fetch, an exception is flagged in a result register for the instruction. The core will process the exception when the instruction commits. If the instruction does not commit (it could be a speculated load instruction) then the exception will not be processed.

PMA

The addressed memory did not pass the physical memory attributes testing. For example a write operation attempted to a ROM address space.

IBE

A timeout signal is typically wired to the err_i input of the core and if the instruction memory does not respond with an ack_i signal fast enough and error will be triggered. This will happen most often when the core is attempting to access an unimplemented memory area for which no ack signal is generated. When the err_i input is activated during an instruction fetch, a breakpoint instruction is loaded into the cache at the address of the error.

NMI

Non-maskable interrupt.

BT

The core will generate the BT (branch target) exception if a branch instruction points back to itself. Branch instructions in this sense include jump (JMP) and call (CALL) instructions.

Memory Management

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. Thor2022 has a region table that supports up to eight separate regions.

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

Once the region is identified the access rights for a particular page within the region can be found from the PMT corresponding to the region.

PMA - Physical Memory Attributes Checker

Overview

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

Physical memory attributes are stored in an eight-entry region table. This table includes the address range the attributes apply to and the attributes themselves. Address ranges are resolved only to bit four of the address. Meaning the granularity of the check is 16 bytes.

Most of the entries in the table are hard-coded and configured when the system is built. However, they may be modified at the address range $FF9F0xxx.

Physical memory attributes checking is applied in all operating modes.

Region Table Description

Address	Bits		
00	64	start	start address bits 4 to 67 of the physical address range
10	64	nd	end address bits 4 to 67 of the physical address range
20	18	pmt	associated PMT address
30	64	cta	card table address
40	19	at	memory attributes
50 to 70	…	…	not used
…	…	…	7 more register sets

PMT Address

The PMT address specifies the location of the associated PMT. Only the low order 18 bits of this value are significant. The high order bits of the PMT table address are fixed at $F..FD.

CTA – Card Table Address

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

Attributes

Bitno		
0	X	may contain executable code
1	W	may be written to
2	R	may be read
3	C	may be cached
4-6	G	granularity

G	
0	byte accessible
1	wyde accessible
2	tetra accessible
3	octa accessible
4	hexi accessible
5 to 7	reserved

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

Page Management Table - PMT

Overview

After the region is determined and page number calculated the page management table is referenced to obtain the access rights to the page. The table 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.

Location

The page management table may be accessed with load and store hexi-byte instructions, LDH, STH at the address of $FFDxxxxx. The low order bits of the base address are supplied by the region table.

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

PMTE Description

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

Page Management Table Entry

V	N	M	$\sim_{10}$	E	AL_2	PCI_{16}
ACL_{16}						Share Count$_{16}$
Access Count$_{32}$						
PL_8			Key_{24}			

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. 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

PCI

The PCI indicator bits tell which slices of the page are compressed.

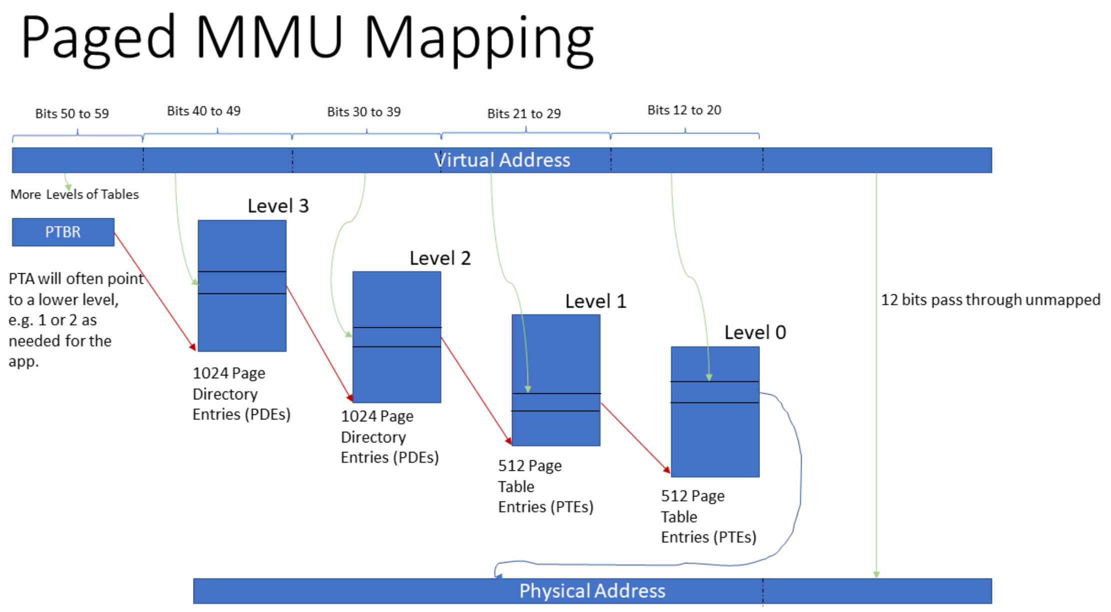
Page Tables

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 are capable of mapping the entire virtual address range. At the topmost level a register points to a page directory, that page directory points to a page directory at a lower level until finally a page directory points to a page containing page table entries. To map an entire 64-bit virtual address range approximately five levels of tables are required.

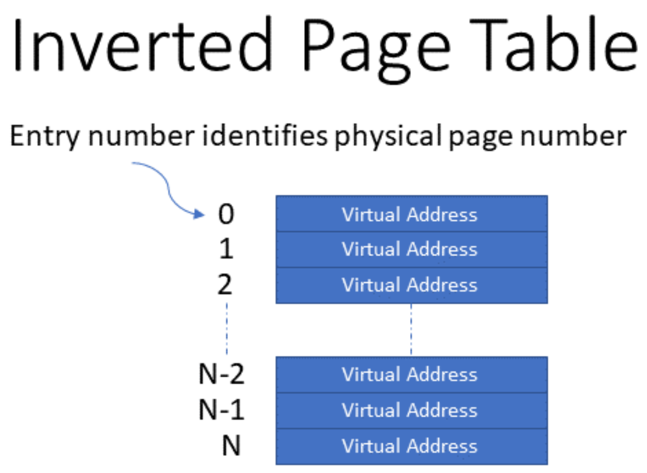

Paged MMU Mapping
Bits 50 to 59
Bits 40 to 49
Bits 30 to 39
Bits 21 to 29
Bits 12 to 20
Virtual Address
More Levels of Tables
PTBR
PTA will often point to a lower level, e.g. 1 or 2 as needed for the app.
Level 3
1024 Page Directory Entries (PDEs)
Level 2
1024 Page Directory Entries (PDEs)
Level 1
512 Page Table Entries (PTEs)
Level 0
512 Page Table Entries (PTEs)
12 bits pass through unmapped
Physical Address

Inverted Page Tables

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

The Simple Inverted Page Table

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



Hashed Page Tables

Hashed Table Access

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

Shared Memory

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

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

Thor2022 Page Tables

Thor2022 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 64/128-bit HPTE format. Note that the first part of the HPTE has the same format as a PTE.

31			20	19	18 17	16	15	0
V	~$_3$	MB$_3$	ME$_3$	M	RWX $_3$	A	~	Physical Page Number$_{15..0}$
ASID$_{10}$				G	BC$_4$		~	Virtual Page Number$_{15..0}$
Physical Page Number$_{47..16}$								
Virtual Page Number$_{47..16}$								

Small Hash Page Table Entry – SHPTE

For systems with less than 4GB of physical memory the small hash page table entry may be used. This is a configuration option.

31			20	19	18 17	16	15	0
V	~$_3$	MB$_3$	ME$_3$	M	RWX $_3$	A	~	Physical Page Number$_{15..0}$
ASID$_{10}$				G	BC$_4$		~	Virtual Page Number$_{15..0}$

Fields Description

V	translation Valid
G	global translation
MB	page access mask begin
ME	page access mask end
RWX	readable, writeable, executable
ASID	address space identifier
BC	bounce count

MB$_4$	ME$_4$	
15	0	1 MB page
15	1	16 MB page
15	2	256 MB page
15	3	4 GB page

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

The virtual to physical address mapping is for a 64kB page. But the entire 64kB page does not need to be accessible by the process. The page mask begin and end fields allow access with a 4kB granularity.

The page mask begin field is added to bits 12 through 15 of the virtual address. The effect is to rotate a 4kB block of memory so that it begins at start of the 64kB block. This field is used to allocate less than 64kB to a process. It allows the 64kB page to be shared by different virtual addresses.

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 2048 bits was chosen as the amount of memory to fetch.

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

127 0

PTE0
PTE1
PTE2
PTE3
PTE4
PTE5
PTE6
PTE7

Size of Page Table

There are several conflicting elements to deal with, with regards to the size of the page table. Ideally, the page table is small enough to fit into the block RAM resources available in the FPGA. So, about 1/3 of the block RAMs available are dedicated to MMU use. At the same time a multiple of the number of physical pages of memory should be supported to support page sharing and swapping pages to secondary storage. To support swapping pages, double the number of physical entries were chosen. To support page sharing, double that number again. Therefore, a minimum size of a page table would contain at least four times the number of physical pages for entries. By setting the size of the page table instead of the size of pages, it can be worked backwards how many pages of memory can be supported.

For a system using 512k block RAM to store PTEs. 512k / 32 = 16384 entries. 16384 / 4 = 4096 physical pages. Since the RAM size is 512MB, each page would be 512MB/4096 = 128kB. Since half the pages may be in secondary storage, 1GB of address range is available.

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

Hash Function

The hash function needs to reduce the size of a virtual address down to a 11-bit number. The asid should be considered part of the virtual address. Including the asid an address is 76 bits. The first thing to do is to throw away the lowest sixteen bits as they pass through the MMU unaltered. We now have 60-bits to deal with. We can probably throw away some high order bits too, as a process is not likely to use the full 64-bit address range.

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

Collision Handling

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

Finding a Match

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

Locality of Reference

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

Access Rights

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

Location of Page Table

Thor2022's hash page table is in the physical address space at $FFAxxxxx. It is a specially dedicated block RAM memory which has two sides. One side is updateable and readable via the load hexi-byte pair and store hexi-byte pair LDHP, STHP instructions. The other side is updateable and readable in terms of page groups by the hash page table control logic.

Thor2022 Hierarchical Page Table Setup

Page Table Entries - PTE

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

Page Table Entry Format - PTE

V	LVL_3	MB_3	ME_3	M	RWX_3	A	~	Physical Page $Number_{15..0}$
Physical Page $Number_{47..16}$								

Small Page Table Entry Format - SPTE

The small page table entry format may be selected as a configuration option for systems with limited physical memory. If physical memory is limited to less than 4GB the small page table entry format may be used. 16384 SPTEs fit into a 64kB page.

V	LVL_3	MB_3	ME_3	M	RWX_3	A	~	Physical Page $Number_{15..0}$

Field	Size	Purpose
PPN	16/48	Physical page number
A	1	1=accessed
X	1	1=executable
W	1	1=writeable
R	1	1=readable
M	1	1=modified
ME	3	page slice end
MB	3	page slice begin
V	1	1 if entry is valid, otherwise 0
LVL	3	the page table level of the entry pointed to

Note the LVL field for both PTEs and PDEs is in the same position.

Page Directory Entries - PDE

A hierarchical table usually consists of multiple levels of pages for the page table. The leaf entries are the PTEs non-leaf entries are page directory entries or PDEs. PDEs are 64-bits in size, therefore 8192 PDEs fit in one 64kB page of memory.

V	LVL$_3$	~$_4$	~$_4$	~	Physical Page Number$_{15..0}$
Physical Page Number$_{47..16}$					

Small Page Directory Entry Format – SPDE

Small page directory entries are used for systems with less than 4GB of physical memory, and conserve space over PDEs. 16,384 SPDEs fit into one 64kB page of memory, This allows 14-bits of the virtual address to be absorbed per table.

V	LVL$_3$	~$_4$	~$_4$	~	Physical Page Number$_{15..0}$

Field	Size	Purpose
PPN	16/48	Page number of next lower page table
~	12	reserved
V	1	1 if entry is valid, otherwise 0
LVL	3	the page table level of the entry pointed to

MMU Cache

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

TLB – Translation Lookaside Buffer

Overview

The 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. Thor2022’s TLB is quite large being five way associative with 1024 entries per way. This choice of size was based on the minimum number of block RAMs that could be used to implement the TLB. On a TLB miss the page table is searched for a translation and if found the translation is stored in one of the ways of the TLB. The way selected is determined either randomly or in a least-recently-used fashion as one of the first four ways. The fifth way may not be updated automatically by a page table search, it must be updated by software.

Size / Organization

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

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

TLB Entries - TLBE

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

V	LVL_3	MB_3	ME_3	M	RWX_3	A	~	Physical Page Number$_{15..0}$	
$ASID_{10}$				G	BC_4		~	Virtual Page Number$_{15..10}$	~$_{10}$
Physical Page Number$_{47..16}$									
Virtual Page Number$_{47..16}$									

V	N	M	~$_{10}$	E	AL_2	PCI_{16}
ACL_{16}						Share Count$_{16}$
Access Count$_{32}$						
PL_8			Key$_{24}$			

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

What is Translated

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

Page Size

Because the TLB caches address translations it can get away with a much smaller page size than the page map can for a larger memory system. 4kB is a common size for many systems. There are some indications in contemporary documentation that a larger page size would be better. In this case the TLB uses 64kB. For a 512MB system (the size of the memory in the test system) there are 8192 64kB pages.

Management

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

Accessing the TLB

A TLB entry contains too much information to be updated with a single register write. Since the information must also be updated atomically to ensure correct operation, the TLB update occurs in an indirect fashion. First holding registers are loaded with the desired values, then all the holding registers are written to the TLB in a single atomic cycle. The TLB is addressed in the physical memory space in the address range $FFE000x0. There are five buckets which must be filled with TLB info using STH instructions. Then address $FFE00070 is written to causing the TLB to be updated.

The low order bits of the bucket four determine which way to update in the TLB if the algorithm is a fixed or LRU way algorithm. Otherwise, a way to update will be selected randomly. The data is hexi-byte aligned. When the LRU algorithm is active the most recently used entry is placed in way #0. It may be desirable to bump out entry #3 and replace it with the new entry for LRU operation.

Bucket	127 32	31 16	15	14 5	4 3	2 0
00	PTE$_{128}$					
10	PMT$_{128}$					
20	▨	PTE Address				
30	▨	PMT Address				
40	▨	▨	0	Entry Num$_{10}$	~$_{2}$	way$_{3}$
50	▨					
60	▨					
70	▨					

Example TLB Update Routine

```
_TLBMap:
        ldhs        a0,0[sp]
        sth         a0,0xFFE00040            # entry number
        ldhs        a0,16[sp]
        sth         a0,0xFFE00000            # PTE value
        ldhs        a0,32[sp]
        sth         a0,0xFFE00010            # PMT value
        ldhs        a0,48[sp]
        sth         a0,0xFFE00020            # PTE address
        ldhs        a0,64[sp]
        sth         a0,0xFFE00030            # PMT address
        sth         r0,0xFFE00070            # triggers a TLB update
        add         sp,sp,80
        rts
```

TLB Entry Replacement Policies

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

For fixed replacement, the way to update must be specified by a software instruction. Least recently used replacement, LRU, rotates the most recent address translation to the first way and updates by over-writing the value in the third way. Random replacement chooses a way to replace at random.

Flushing the TLB

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

Reset

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

Global Bit

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

Page Management

There are typically at least three primary lists that a page may be on. These are the active list, the inactive list, and the free list. Typically, a list link field would be present in the PMTE.

Key Cache

Overview

Associated with each page of memory is a memory key. To access a page of memory the memory key must match with one of the keys in the applications keyset. The keyset is maintained in the keys CSRs. The key size of 20 bits is a minimum size recommended for security purposes.

The key associated with each memory page is stored in a table in main memory. Each key occupies a tetra-byte of memory to keep caching simple. So that two memory accesses are not required to access a page of memory this table of keys is cached. When a page of memory is accessed the key cache is accessed in parallel.

The key cache is a direct mapped cache organized as 256 lines of 16 keys. Key values are stored in LUT rams. 256 address tags are stored in LUT ram.

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

Memory is divided into 512-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 1MB card memory is required at the outermost layer. The outer most 1MB card memory layer is itself divided into 2048 512-byte card pages. Note that each byte represents the pointer store status for a 512B region. The 2048B memory is further resolved to single octa indicating if any pointer store has taken place. Thus, for a 512MB memory system a three-level card memory is used.

Layer	Resolving Power	
0	1 MB	512B regions
1	2 kB	256kB regions
2	4 B	128 MB regions

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 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.
        ; Usable with up to 64-bit memory systems.
        ; Three level card memory
        ;
GCWriteBarrier:
        STO     a0,[a1]         ; store the value to memory at a1
        SRL     a1,a1,#8        ; compute card address
        STB     x0,[a2+a1]      ; clear byte in card memory
        SRL     a1,a1,#8        ; repeat for each table level
        STB     x0,[a2+a1]
        SRL     a1,a1,#8
        STB     x0,[a2+a1]
        ;… more stores as needed
        JMP     lk2
```

System Memory Map

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

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

Debugging Unit

Overview

The Thor2021 has several debug features including debug exceptions on address matches and instruction tracing. Instruction trace trigger registers are shared with the debug address registers. Which function is triggered on an address match is controlled in the debug control register.

Instruction Tracing

Instruction tracing is enabled by setting the trace enable bit (bit 32 to 35) for the corresponding debug address match register. Tracing will begin when an address match occurs and continue until the trace buffer is full. The trace queue is 8kB in size allowing thousands of instructions to be traced.

Trace Queue Entry Format

The trace queue stores both complete instruction pointer addresses and branch taken-not-taken (TNT) history. The low order two bits of the trace entry indicate the type of record stored by the entry. There are currently two record types. Record type zero is an instruction pointer address. Record type one is a history record for branches.

111		2	1 0
			Rectype$_2$
~$_{14}$	Selector Value$_{32}$	Instruction Pointer bits 0 to 63	00

111 9	8 2	1 0
Branch Taken-Not-Taken History$_{103}$	count$_7$	01

Up to 103 bits of branch TNT history may be stored in a single record. The number of bits stored is recorded in bits 2 to 8 of the record. After four full branch TNT history records, the trace will record the current instruction address in whole.

Trace Readback

A trace of instructions executed may be read back from the trace queue using the PEEKQ and POPQ instructions. The processor trace queue is accessible as queue number 15. Queue 15 contains the raw history record. Software should get the status using the STATQ instruction to see if data is available, then pop queue 15 to get the data record.

Address Matching

The debug and trace unit use selectors to select an address range to match against.

Extended Precision Arithmetic

Extended precision arithmetic is supported by the CARRY instruction and carry storage registers P0 to P7. The CARRY instruction specifies which of the following instructions input or output a carry value. The instruction group covered by the CARRY instruction is executed with interrupts disabled. Because the instructions are treated as a group there is no need to store or retrieve intermediate carry values through context switches.

Instruction Set Architecture Description

Overview

Like the original Thor core, the instruction set is variable length. Instructions vary from two to eight bytes in length (2, 4, 6 or 8). Commonly used instructions often have short forms. While adding complexity to the processor, variable length instructions make better use of the cache.

Thor2021 uses 16-bit instruction parcels as a minimum. All instructions are a multiple of two bytes in length. Formats were considered that varied by a single byte in size. An issue with using instructions that vary by only a single byte in size is that there are twice as many instruction length decoders required in hardware. The instruction aligner would be larger and more complex with byte aligned instructions. The 16-bit parcel is an engineering compromise between minimizing instruction size and minimizing processor complexity. Seriously considered was packing three 41-bit instructions into 128-bit bundles. One issue with this approach is software complexity of the assembler and linker. Byte aligned instructions are handled well by existing software, bit aligned instructions are not as popular.

Thor2021 has the capacity to represent any instruction as a vector instruction. A single bit in the instruction indicates if it is a vector instruction. This avoids the issue many instruction set architectures have of having large numbers of instructions duplicated for operation with vectors. There are literally hundreds of instructions that make sense to have vector versions. Having two sets of opcodes would add too much complexity to the processor. Because of the large number of vector instructions it ends up consuming an opcode bit anyways.

Root Opcode

The root opcode determines the class of instructions executed. Some commonly executed instructions are also encoded at the root level to make more bits available for the instruction. The root opcode is always present in all instructions as the lowest eight bits of the instruction. The instruction length is determined entirely by the value of the root opcode.

Byte 3	Byte 2	Byte 1	Byte 0

▼

31 21	20 15	14 9	8	7 0
Constant$_{11}$	Ra$_6$	Rt$_6$	v	Opcode$_8$

Vector Instruction Indicator

The processing core needs to know if an instruction is a vector instruction before it is fully decoded. Depending on if the instruction is a vector instruction, it may be re-decoded and sent into the pipeline multiple times. The processor needs to know very quickly and simply at the instruction fetch stage if the instruction is a vector operation. So, to help things along Thor2021 encodes this information in bit 8 of all instructions. If bit 8 is a '1' then the instruction is a vector instruction. See the sample instruction below.

▼

31 21	20 15	14 9	8	7 0
Constant$_{11}$	Ra$_6$	Rt$_6$	v	Opcode$_8$

Target Register Spec

Most instructions have a target register. The register spec for the target register is usually in the same position, bits 9 to 14 of an instruction. If the instruction is a vector instruction, then the target register will be a vector register, otherwise it is a scalar register.

		▼		
31 21	20 15	14 9	8	7 0
Constant$_{21}$	Ra$_6$	Rt$_6$	V	Opcode$_8$

Register Formats

R1 (one source register)

31 25	24 22	21	20 19	18 14	13 9	8	7 0
Func$_7$	m$_3$	z	Sz$_2$	Ra$_5$	Rt$_5$	v	Opcode$_8$

R2 (two source register)

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz$_2$	m$_3$	z	P	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	Opcode$_8$

R3 (three source register)

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
Func$_7$	m$_3$	z	~$_2$	Sz$_3$	P	Tc	Rc$_5$	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	Opcode$_8$

Immediate Format

RIL (with prefix)

The immediate prefix replaces the sign extension from bit 23 to the machine width with bits from the prefix instruction.

Bytes 9 to 11	Byte 8	Byte 7	Byte 6	Bytes 1 to 5	Byte 0

Scalar Immediate

Instruction					Prefix		
47 19	18 14	13 9	8	7 0	47 9	8	7 0
Immediate$_{28..0}$	Ra$_5$	Rt$_5$	0	Opcode$_8$	Immediate$_{63..25}$	v	1Fh$_6$

The instruction prefix may be 2, 4, 6 or 8 bytes in size.

Vector Immediate

Instruction							Prefix		
47 45	44	43 19	18 14	13 9	8	7 0	47 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	Ra$_5$	Rt$_5$	1	Opcode$_8$	Immediate$_{63..25}$	v	1Fh$_6$

Clock Cycles

Clock cycles given for instructions are approximate and represent a relative relationship between instructions. An instruction with a given clock cycle count of two will take approximately twice as long to execute as an instruction with a given clock cycle count of one. Generally, a clock cycle count of one means the instruction requires only a single clock to execute. However, the effective execution time of the instruction may be less if multiple execution units can execute the instruction or if execution of the instruction may be overlapped with execution of other instructions.

Execution Units

Most instructions execute on a particular type of execution unit. For instance, the ADD instruction is executed on an ALU while the LDB instruction is executed on a memory unit. The execution unit for the instruction is listed in the instruction's description. The execution of some instructions is supported on only a single execution unit even if multiple execution units of the same type are available. For instance, infrequently used instructions like bitfield manipulations or sub-word shifts and rotates are supported on only a single ALU. Instructions that execute on two different execution units may execute at the same time. This is important for instruction scheduling.

Sub-word Operations

Mnemonics

Operation sizes are specified by adding a size qualifier to the mnemonic as in ADD.T. The T stands for Tetra.

Possible size qualifiers:

.B	byte
.W	wyde
.T	tetra
.O	octa
.H	hexi
.BP	byte parallel
.WP	wyde parallel
.TP	tetra parallel
.OP	octa parallel
.HP	hexi parallel

Result Format

For sub-word operations which may be specified for instructions the results are sign extended to the word width of 128-bits.

Example:

SLL.W r1,r2,6

Will shift the first 16 bits of the register to the left by six bits, then sign extend the result from bit 15 to bit 127.

Instruction Encoding

Two Bit Size Code

Sz_2	
0	hexis
1	octas
2	tetras
3	wydes

Four Bit Size Code

Sz_4	
0	hexis
1	octas
2	tetras
3	wydes
4 to 7	reserved
8	hexi
9	octa parallel

10	tetra parallel
11	wyde parallel

Arithmetic / Logical / Shift

ABS – Absolute Value

Description:

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

Integer Instruction Format: R1

31 25	24 22	21	20 19	18 14	13 9	8	7 0
06h$_7$	m$_3$	z	Sz$_2$	Ra$_5$	Rt$_5$	v	01h$_8$

v: 0 = scalar, 1 = vector op

Operation:

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

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Rt[x] = Ra[x] < 0 ? -Ra[x] : Ra[x]
    else if (z) Rt[x] = 0
    else Rt[x] = Rt[x]
```

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. The mask register is ignored for scalar instructions. All registers are integer registers.

Unless the CARRY instruction is used, the input carry will be zero.

Instruction Format: R2

Results are sign extended to 128-bits.

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz$_2$	m$_3$	z	P	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	19h$_8$

Operation:

Rt = Ra + Rb + carry

Instruction Format: R3

For this format, if the CARRY instruction is in use Rc should be zero.

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
04h$_7$	m$_3$	z	~$_2$	Sz$_3$	P	Tc	Rc$_5$	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	02h$_8$

Operation:

Rt = Ra + Rb + (Rc|carry)

Operation:

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.

Unless the CARRY instruction is used, the input carry will be zero.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	Ra$_5$	Rt$_5$	0	04h$_8$

Instruction Format: RIL

47 19	18 14	13 9	8	7 0
Immediate$_{28..0}$	Ra$_5$	Rt$_5$	0	D4h$_8$

Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	Ra$_5$	Rt$_5$	1	D4h$_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra + immediate + carry

Exceptions:

Notes:

AND – Bitwise And

Description:

Perform a bitwise ‘and’ operation between operands.

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	~	Tb	Rb_5	Ra_5	Rt_5	v	$1Ah_8$

Operation:

Rt = Ra & Rb

Integer Instruction Format: R3

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
$08h_7$	m_3	z	$\sim_2$	Sz_3	P	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

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

Operation:

Rt = Ra & Rb & Rc

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

ANDC – Bitwise ‘And’ with Complement

Description:

Perform a bitwise ‘and’ with complement operation between operands.

Integer Instruction Format: R3

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
0Bh$_7$	m$_3$	z	~$_2$	Sz$_3$	P	Tc	Rc$_5$	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	02h$_8$

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

Operation:

Rt = Ra & ~Rb & Rc

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

ANDI – Bitwise ‘And’ Immediate

Description:

Bitwise ‘and’ a register and immediate value and place the result in the target register. The immediate is one extended to the machine width.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	Ra$_5$	Rt$_5$	v	08h$_8$

Instruction Format: RIL

47 19	18 14	13 9	8	7 0
Immediate$_{28..0}$	Ra$_5$	Rt$_5$	0	D8h$_8$

Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	Ra$_5$	Rt$_5$	1	D8h$_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra & immediate

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions:

Notes:

ANDM – Bitwise ‘And’ with Mask

Description:

Generates a mask consisting of ‘1’s from the mask begin, Rb_5, to the mask end Rc_5, inclusive. Perform a bitwise ‘and’ operation between register Ra and the mask and store the result in register Rt. The immediate constant is one extended before use.

Instruction Format: RM

47 41	40 38	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$08h_7$	m_3	z	c_3	b_3	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	AAh_8

b_3: if b_3 is 0xxb then Rb is a five bit register specifier.

if b_3 is 1xxb then Rb is low order five-bits of a seven-bit constant, the least significant two bits of b_3 is bit 5 and 6 of the constant

c_3: works the same way as b_2 except for the Rc field

Operation

Rt = Ra & Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] & Immediate
    else Vt[x] = Vt[x]
```

Clock Cycles: 1

Execution Units: First Integer ALU

Exceptions: none

AND_OR – Bitwise And then Or

Description:

Perform a bitwise ‘and’ operation between the first two operands, Ra, Rb, then bitwise or the result with a third operand, Rc. Place the result in the target register Rt.

Integer Instruction Format: R3

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
78h$_7$	m$_3$	z	~$_2$	Sz$_3$	P	Tc	Rc$_5$	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	02h$_8$

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

Operation:

Rt = (Ra & Rb) | Rc

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

BCDADD – BCD Add

Description:

Adds two registers using BCD arithmetic and places the result in a target register. Unless the CARRY modifier is used the carry input will be zero.

Instruction Format: R3

47 41	49 38	37	36 33	32 31	30 25	24	23 19	18 14	13 9	8	7 0
00h$_7$	m$_3$	z	~$_4$	0$_2$	0$_7$	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	F5h$_8$

Clock Cycles: 5

Execution Units: First Integer ALU

Operation:

Rt = Ra + Rb + carry

Exceptions: none

BCDMUL – BCD Multiply

Description:

Multiply two registers using BCD arithmetic shift the result right by Rc digits round and place the result in a target register. This instruction has been designed to implement fixed point arithmetic.

Instruction Format:

47 41	49 38	37	36 31	30	29 25	24	23 19	18 14	13 9	8	7 0
02h$_7$	m$_3$	z	~$_6$	Tc	Rc$_5$	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	F5h$_8$

Clock Cycles: 150

Execution Units: First Integer ALU

Operation:

Rt = (Ra * Rb) >> (Rc * 4)

Exceptions: none

BCDSUB – BCD Subtract

Description:

Subtract two registers using BCD arithmetic and places the result in a target register. Unless the carry modifier is used the carry input will be zero.

Instruction Format:

47 41	49 38	37	36 33	32 31	30 25	24	23 19	18 14	13 9	8	7 0
$01h_7$	m_3	z	$\sim_4$	0_2	0_7	Tb	Rb_5	Ra_5	Rt_5	v	$F5h_8$

Clock Cycles: 5

Execution Units: First Integer ALU

Operation:

Rt = Ra – Rb – carry

Exceptions: none

BFALIGN – Bitfield Align

Description:

Align the contents of register Ra with a bitfield specified between mask begin and mask end. This instruction can be used in combination with BFCLR and OR to perform a bitfield insert.

Integer Instruction Format: BF

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$00h_7$	m_3	z	c_3	b_3	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	AAh_8

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

b_3: if b_3 is 0xxb then Rb is a five bit register specifier.

if b_3 is 1xxb then Rb is low order five-bits of a seven-bit constant, the least significant two bits of b_3 is bit 5 and 6 of the constant

c_3: works the same way as b_2 except for the Rc field.

Operation

Rt = (Ra << Mb) & Immediate Mask

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = (Va[x] << Mb) & Immediate Mask
    else Vt[x] = Vt[x]
```

Execution Units: First Integer ALU

Exceptions: none

BFCHG – Bitfield Change

Description:

Flip the bits of a bitfield specified between mask begin and mask end.

Integer Instruction Format: BF

Generates a mask consisting of ‘1’s between the mask begin, Rb, and mask end, Rc, inclusive. Bitwise exclusive ‘or’ the mask with the value in register Ra and store the result in register Rt. This instruction is useful for flipping bit fields.

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$0Ah_7$	m_3	z	c_3	b_3	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	AAh_8

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

b_3: if b_3 is 0xxb then Rb is a five bit register specifier.

if b_3 is 1xxb then Rb is low order five-bits of a seven-bit constant, the least significant two bits of b_3 is bit 5 and 6 of the constant

c_3: works the same way as b_2 except for the Rc field.

Operation

Rt = Ra ^ Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] ^ Immediate
    else Vt[x] = Vt[x]
```

Execution Units: First Integer ALU

Exceptions: none

BFCLR – Bitfield Clear

Description:

Clear the bits of a bitfield specified between mask begin and mask end.

Integer Instruction Format: BF

Generates a mask consisting of ‘1’s between the mask begin, Rb, and mask end, Rc, inclusive. Bitwise and with the complement of the mask with the value in register Ra and store the result in register Rt.

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$0Bh_7$	m_3	z	c_3	b_3	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	AAh_8

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

b_3: if b_3 is 0xxb then Rb is a five bit register specifier.

if b_3 is 1xxb then Rb is low order five-bits of a seven-bit constant, the least significant two bits of b_3 is bit 5 and 6 of the constant

c_3: works the same way as b_2 except for the Rc field.

Operation

Rt = Ra & ~Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] & ~Immediate
    else Vt[x] = Vt[x]
```

Execution Units: First Integer ALU

Exceptions: none

BFEXT – Bitfield Extract

Description:

Extract the bits of a bitfield specified beginning at mask begin in Rb and extending by mask width in Rc and sign extend and right align the result in the target register. The value in Rc is one less than the field width.

Integer Instruction Format: BF

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$05h_7$	m_3	z	c_3	b_3	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	AAh_8

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

b_3: if b_3 is 0xxb then Rb is a five bit register specifier.

if b_3 is 1xxb then Rb is low order five-bits of a seven-bit constant, the least significant two bits of b_3 is bit 5 and 6 of the constant

c_3: works the same way as b_2 except for the Rc field.

Operation

Rt = sign extend (ror(Ra,Mb) & Mask)

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = sign extend (ror(Va[x],Mb) & mask)
    else Vt[x] = Vt[x]
```

Execution Units: First Integer ALU

Exceptions: none

BFEXTU – Bitfield Extract Unsigned

Description:

Extract the bits of a bitfield specified beginning at mask begin in Rb and extending by mask width in Rc and zero extend and right align the result in the target register. The value in Rc is one less than the field width.

Integer Instruction Format: BF

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
04h$_7$	m$_3$	z	c$_3$	b$_3$	Tc	Rc$_5$	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	AAh$_8$

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

b$_3$: if b$_3$ is 0xxb then Rb is a five bit register specifier.

if b$_3$ is 1xxb then Rb is low order five-bits of a seven-bit constant, the least significant two bits of b$_3$ is bit 5 and 6 of the constant

c$_3$: works the same way as b$_2$ except for the Rc field.

Operation

Rt = zero extend ((ror(Ra,Mb)) & Mask)

Vector Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = zero extend ((ror(Va[x],Mb)) & Mask)
    else if (z) Vt[x]  = 0
    else Vt[x] = Vt[x]
```

Execution Units: First Integer ALU

Exceptions: none

BFFFO –Find First One

Description:

A bitfield contained in Ra is searched beginning at the most significant bit to the least significant bit for a bit that is set. The index into the bitfield of the bit that is set is stored in Rt. If no bits are set, then Rt is set equal to -1.

Integer Instruction Format: BF

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$01h_7$	m_3	z	c_3	b_3	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	AAh_8

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

b_3: if b_3 is 0xxb then Rb is a five bit register specifier.

if b_3 is 1xxb then Rb is low order five-bits of a seven-bit constant, the least significant two bits of b_3 is bit 5 and 6 of the constant

c_3: works the same way as b_2 except for the Rc field.

Clock Cycles:

Execution Units: First Integer ALU

Execution Units: Integer

Exceptions: none

BFSET – Bitfield Set

Description:

Set the bits of a bitfield specified between mask begin and mask end.

Integer Instruction Format: BF

Generates a mask consisting of ‘1’s between the mask begin, Mb_6, and mask end, Me_6, inclusive. Bitwise ‘or’ the mask with the value in register Ra and store the result in register Rt.

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$09h_7$	m_3	z	c_3	b_3	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	AAh_8

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

b_3: if b_3 is 0xxb then Rb is a five bit register specifier.

if b_3 is 1xxb then Rb is low order five-bits of a seven bit constant, the least significant two bits of b_3 is bit 5 and 6 of the constant

c_3: works the same way as b_2 except for the Rc field.

Operation

Rt = Ra | Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] | Immediate
    else Vt[x] = Vt[x]
```

Execution Units: First Integer ALU

Exceptions: none

BMAP – Byte Map

Description:

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. Bytes which are not mapped will remain unchanged in the target register.

Integer Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
0_2	m_3	z	~	Tb	Rb_5	Ra_5	Rt_5	v	$44h_8$

Rb value:

63 36	3532	3128	2724	2320	1916	1512	11 8	7 4	3 0	
…	$B8_4$	$B7_4$	$B6_4$	$B5_4$	$B4_4$	$B3_4$	$B2_4$	$B1_4$	$B0_4$	<= Target byte = Bn

Operation:

Vector Operation

Execution Units: First Integer ALU

Clock Cycles: 1

Exceptions: none

Notes:

BMAPZ – Byte Map Zero

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. Bytes which are not mapped will end up as zero in the target register.

Integer Instruction Format: R2

30 31	29 27	26	2524	23 19	18 14	13 9	8	7 0
1_2	m_3	z	Tb_2	Rb_5	Ra_5	Rt_5	v	$44h_8$

Rb value:

63 36	3532	3128	2724	2320	1916	1512	11 8	7 4	3 0	
…	$B8_4$	$B7_4$	$B6_4$	$B5_4$	$B4_4$	$B3_4$	$B2_4$	$B1_4$	$B0_4$	<= Target byte = Bn

Operation:

Vector Operation

Execution Units: First Integer ALU

Clock Cycles: 1

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

47 41	49 38	37	36 33	32 31	30 26	25 24	23 19	18 14	13 9	8	7 0
$Func_7$	m_3	z	$\sim_4$	0_2	0_5	Tb_2	Rb_5	Ra_5	Rt_5	v	$02h_8$

Fn_7	Function
30h	MOR
31h	MXOR
32h	MORT (MOR transpose)
33h	MXORT (MXOR transpose)

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.

BYTNDX – Byte Index

Description:

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

If a vector BYTNDX instruction is issued and the target is a scalar register then the instruction searches all the vector elements and returns a value which varies from -1 to +511 in the scalar register. Thus, BYTNDX may be used to determine the length of a null termination string in the vector register.

Instruction Format: RI

47 41	49 38	37	3635	3433	32 31	30 25	24	23 19	18 14	13 9	8	7 0
$55h_7$	m_3	z	$\sim_2$	b_2	0_2	0_6	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

b2: upper 2 bits of eight bit constant specified by Rb

Clock Cycles: 1

Execution Units: First Integer ALU

Operation:

Rt = Index of (Rb in Ra)

Exceptions: none

BYTNDXI – Byte Index

Description:

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

If a vector BYTNDX instruction is issued and the target is a scalar register then the instruction searches all the vector elements and returns a value which varies from -1 to +511 in the scalar register. Thus, BYTNDX may be used to determine the length of a null termination string in the vector register.

Instruction Format: RI

31 27	26 19	18 14	13 9	8	7 0
~$_5$	Immediate$_{7..0}$	Ra$_5$	Rt$_5$	v	55h$_8$

Clock Cycles: 1

Execution Units: First Integer ALU

Operation:

Rt = Index of (Imm$_8$ in Ra)

Exceptions: none

CARRY – Carry

Description:

This instruction controls carry propagation between instructions. There is a bit pair indicating carry-in and carry-out for each following instruction. The first following instruction is controlled by the bit-pair 15,16, the second by bit-pair 17,18, and so on for up to eight instructions.

Interrupts are disabled while carry propagation is in effect.

The following instructions may generate or consume carry bits.

ADD, SUB, OR, EOR, SLL, SRL, SRA, MUL, LLA.

Instruction Format: R3

31	30 29	28 27	26 25	24 23	22 21	20 19	18 17	16 15	14 11	10 9	8	7 0
~	IO$_2$	IO$_2$	IO$_2$	IO$_2$	IO$_2$	IO$_2$	IO$_2$	IO$_2$	~$_4$	Pn$_2$	0	F3h$_8$

IO$_2$	Meaning
0	no input or output carry
1	output carry
2	input carry
3	both input and output carry

Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Exceptions: none

Example:

```
# Get at high-order product bits

CARRY P1,{O}{I}

MUL a2,a1,a0

ADD a3,r0,r0
```

CHK – Check Register Against Bounds

Description:

A register is compared to two values. If the register is outside of the bounds defined by Rb and Rc then an exception will occur.

Instruction Format: R3

47 41	49 38	37	36 33	32 31	30 26	25 24	23 19	18 14	13 12	11 9	8	7 0
$19h_7$	m_3	z	$\sim_4$	Tc_2	Rc_5	Tb_2	Rb_5	Ra_5	$\sim_2$	cn_3	v	$02h_8$

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

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.

CHKI – Check Register Against Bounds

Description:

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

Instruction Format: CHKI

47 26	25 24	23 19	18 14	13 12	11 9	8	7 0
Immediate$_{22}$	Tb$_2$	Rb$_5$	Ra$_5$	Imm$_2$	cn$_3$	v	42h$_8$

cn$_3$	exception when not
0	Ra >= Rb and Ra < Imm
1	Ra >= Rb and Ra <= Imm
2	Ra > Rb and Ra < Imm
3	Ra > Rb and Ra <= Imm
4	not (Ra >= Rb and Ra < Imm)
5	not (Ra >= Rb and Ra <= Imm)
6	not (Ra > Rb and Ra < Imm)
7	not (Ra > Rb and Ra <= Imm)

Clock Cycles: 1

Execution Units: Integer ALU

Exceptions: bounds check

Notes:

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

A six-bit immediate value may be specified by Tb$_2$ and Rb.

CLMUL – Carry-less Multiply

Description:

Compute the low order product bits of a carry-less multiply. Both operands must be in registers.

Integer Instruction Format: R3

47 41	39 38	37	36 33	32 31	30 26	25 24	23 19	18 14	13 9	8	7 0
$2Eh_7$	m_3	z	$\sim_4$	0_2	0_5	Tb_2	Rb_5	Ra_5	Rt_5	v	$02h_8$

4 clock cycles

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

CLMULH – Carry-less Multiply High

Description:

Compute the high order product bits of a carry-less multiply. Both operands must be in registers.

Integer Instruction Format: R2

47 41	39 38	37	36 33	32 31	30 26	25 24	23 19	18 14	13 9	8	7 0
$2Fh_7$	m_3	z	$\sim_4$	0_2	0_5	Tb_2	Rb_5	Ra_5	Rt_5	v	$02h_8$

4 clock cycles

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

Description:

CMOVNZ moves a value from Rb or Rc depending on the value in Ra. If Ra is non-zero then Rb is moved to Rt, otherwise Rc is moved to Rt.

For the vector version of the instruction Ra is a scalar. Each bit of Ra is used corresponding to the vector element to determine if the move takes place. This instruction may be used to convert a set of bits in Ra into a Boolean vector.

Instruction Format: BF

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$10h_7$	m_3	z	c_3	b_3	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	AAh_8

Vector Operation

```
For x = 0 to VL-1
    if (Vm[x]) Vt[x] = Ra.bit[x] ? Rb : Rc
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

Exceptions: none

Execution Units: integer ALU

CMP – Compare

Description

Compare two registers and return the relationship between them. The size field applies to only integer compares.

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	P	Tb	Rb_5	Ra_5	Rt_5	v	$0Ch_8$

1 clock cycle

Rt bit	Meaning
	Integer Compare Results
0	= equal
1	< less than
2	<= less than or equal
3	< magnitude less than
4	unordered
5	< unsigned less than
6	<= unsigned less than or equal
8	< > not equal
9	>= greater than or equal
10	> greater than
11	>= magnitude greater than or equal
12	ordered
13	unsigned greater than or equal
14	unsigned greater than
	Float Compare Results
16	= equal
17	< less than
18	<= less than or equal
19	< magnitude less than
20	unordered
21	<> not equal
22	>= greater than or equal
23	> greater than
24	>= magnitude greater than or equal
25	ordered
	Decimal Floating Point Compare Results
32	= equal
33	< less than
34	<= less than or equal
35	< magnitude less than
36	unordered
37	<> not equal
38	>= greater than or equal
39	> greater than

40	>= magnitude greater than or equal
41	ordered
13 to 62	zero (reserved)
63	less than

Operation:

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]
```

Execution Units: Integer ALU, Floating Point Unit

Exceptions: none

CMPI – Compare Immediate

Description

Compare a register and an immediate value and return the relationship between them in a bit vector. See CMP. The immediate is sign extended to the machine width.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	Ra$_5$	Rt$_5$	v	0Bh$_8$

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

Instruction Format: RIL

47 45 44 43 19	18 14	13 9	8	7 0
Immediate$_{28..0}$	Ra$_5$	Rt$_5$	0	D0h$_8$

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

Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	Ra$_5$	Rt$_5$	1	D0h$_8$

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

Operation:

Rt = Ra ? Imm

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] ? Imm
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

Execution Units: Integer ALU, Floating Point Unit

Exceptions: none

CNTLZ – Count Leading Zeros

Description:

Count the number of leading zeros (starting at the MSB) in Ra and place the count in the target register.

Instruction Format: R1

31 25	24 22	21	20 19	18 14	13 9	8	7 0
$00h_7$	m_3	z	Sz_2	Ra_5	Rt_5	v	$01h_8$

R1 Supported Formats: .o

Clock Cycles: 1

Execution Units: First Integer ALU

Exceptions: none

CNTPOP – Count Population

CNTPOP r1,r2
CNTPOP v1,v2
CNTPOP r1,vm2

Description:

Count the number of ones and place the count in the target register.

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = popcnt(Va[x])
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

Instruction Format: R1

31 25	24 22	21	20 19	18 14	13 9	8	7 0
$02h_7$	m_3	z	Sz_2	Ra_5	Rt_5	v	$01h_8$

Execution Units: First Integer ALU

Exceptions: none

COM – Ones Complement

Description:

Bitwise complement all the bits in the register. 1’s become 0’s and 0’s become 1’s. This is an alternate mnemonic for the BFCHG function.

Instruction Format: RM

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$0Ah_7$	m_3	z	3_2	63_6	2_2	0_6	Ra_6	Rt_6	v	AAh_8

1 clock cycle

Operation

Rt = ~Ra

Vector Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = ~Va[x]
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

Exceptions: none

CPUID – CPU Identification

Description:

This instruction returns general information about the core. Register Ra is used as a table index to determine which row of information to return.

Instruction Format:

31 19	18 14	13 9	8	7 0
$\sim_{13}$	Ra_5	Rt_5	v	$41h_8$

Clock Cycles: 1

Execution Units: First Integer ALU

Operation:

Rt = Info[Ra]

Exceptions: none

Index	bits	Information Returned
0	63 to 0	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	63 to 0	Manufacturer name first eight chars “Finitron”
3	63 to 0	Manufacturer name last eight characters
4	63 to 0	CPU class “64BitSS”
5	63 to 0	CPU class
6	63 to 0	CPU Name “Thor2021”
7	63 to 0	CPU Name
8	63 to 0	Model Number “M1”
9	63 to 0	Serial Number “1234”
10	63 to 0	Features bitmap
11	31 to 0	Instruction Cache Size (32kB)
11	63 to 32	Data cache size (16kB)

DIF – Difference

Description:

This instruction is an alternate mnemonic for the PTRDIF instruction where Rc is zero. This instruction computes the difference between two signed values in registers Ra and Rb and places the result in a target Rt register. The difference is calculated as the absolute value of Ra minus Rb.

Instruction Format: R2

47 41	49 38	37	36 35	34 25	24	23 19	18 14	13 9	8	7 0
$14h_7$	m_3	z	0_2	0_{10}	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

Supported Formats: .o

Clock Cycles: 1

Execution Units: Integer

Operation:

Rt = Abs(Ra - Rb)

Exceptions: none

DIV – Division

Description:

Divide two operand values and place the result in the target register. Both operands are in registers. Both operands are treated as signed values. This instruction may cause a divide by zero or overflow exception if enabled. The exception is enabled by the ‘X’ bit of the instruction.

Instruction Format: R2

47 41	49 38	37	36 33	32 31	30 26	25 24	23 19	18 14	13 9	8	7 0
$10h_7$	m_3	z	Sz_4	0_2	0_5	Tb_2	Rb_5	Ra_5	Rt_5	v	$02h_8$

Execution Units: First Integer ALU

Clock Cycles: 150

Exceptions: DBZ

DIVI – Divide by Immediate

Description:

Divide two operand values and place the result in the target register. The first operand must be in a register specified by the Ra field of the instruction. The second operand is an immediate value. Both operands are treated as signed values.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	Ra$_5$	Rt$_5$	v	40h$_8$

Instruction Format: RIL

47 45 44 43 19	18 14	13 9	8	7 0
Immediate$_{28..0}$	Ra$_5$	Rt$_5$	0	DDh$_8$

Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	Ra$_5$	Rt$_5$	1	DDh$_8$

Execution Units: ALU

Clock Cycles: 130

Exceptions: none

Notes:

Divide by zero is not checked for because the divisor is a constant.

DIVU – Divide Unsigned

Description:

Divide two operand values and place the result in the target register. Both operands are in registers. Both operands are treated as unsigned values.

Instruction Format: R2

47 41	49 38	37	36 33	32 31	30 26	25 24	23 19	18 14	13 9	8	7 0
$11h_7$	m_3	z	Sz_4	0_2	0_5	Tb_2	Rb_5	Ra_5	Rt_5	v	$02h_8$

Execution Units: First Integer ALU

Clock Cycles: 130

Exceptions: none

DIVUI – Divide Unsigned by Immediate

Description:

Divide two operand values and place the result in the target register. The first operand must be in a register specified by the Ra field of the instruction. The second operand is an immediate value. Both operands are treated as unsigned values.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
$Immediate_{12..0}$	Ra_6	Rt_6	v	$4Fh_8$

Execution Units: ALU

Clock Cycles: 130

Exceptions: none

Notes:

DIVSU – Divide Signed by Unsigned

Description:

Divide two operand values and place the result in the target register. Both operands are in registers. The register operand Ra is a signed value, Rb is an unsigned value.

Instruction Format: R2

47 41	49 38	37	36 33	32 31	30 26	25 24	23 19	18 14	13 9	8	7 0
$12h_7$	m_3	z	Sz_4	0_2	0_5	Tb_2	Rb_5	Ra_5	Rt_5	v	$02h_8$

Execution Units: ALU

Clock Cycles: 130

Exceptions: none

ENOR – Bitwise Exclusive Nor

Description:

Perform a bitwise exclusive ‘nor’ operation between operands.

Integer Instruction Format: R3

47 41	49 38	37	36 33	32 31	30 26	25 24	23 19	18 14	13 9	8	7 0
$02h_7$	m_3	z	Sz_4	Tc_2	Rc_5	Tb_2	Rb_5	Ra_5	Rt_5	v	$02h_8$

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

Operation:

Rt = ~(Ra ^ Rb ^ Rc)

Execution Units: All Integer ALU’s

Exceptions: none

EOR – Bitwise Exclusive Or

Description:

Perform a bitwise ‘or’ operation between operands. The carry input will be zero unless the CARRY instruction is in use.

Instruction Format: R2

30 31	29 27	26	2524	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	Tb_2	Rb_5	Ra_5	Rt_5	v	$14h_8$

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

Operation:

Rt = Ra ^ Rb ^ carry

Integer Instruction Format: R3

47 41	49 38	37	36 33	32 31	30 26	25 24	23 19	18 14	13 9	8	7 0
$0Ah_7$	m_3	z	Sz_4	Tc_2	Rc_5	Tb_2	Rb_5	Ra_5	Rt_5	v	$02h_8$

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

Operation:

Rt = Ra ^ Rb ^ Rc ^ carry

Clock Cycles: 1

Execution Units: All Integer ALU’s

Exceptions: none

EORI – Bitwise Exclusive ‘Or’ Immediate

Description:

Perform a bitwise exclusive ‘or’ operation between operands. The immediate is zero extended to the machine width.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	Ra$_5$	Rt$_5$	v	0Ah$_8$

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

Instruction Format: RIL

47 45 44 43 19	18 14	13 9	8	7 0
Immediate$_{28..0}$	Ra$_5$	Rt$_5$	0	DAh$_8$

Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	Ra$_5$	Rt$_5$	1	DAh$_8$

Operation

Rt = Ra ^ Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] ^ Immediate
    else Vt[x] = Vt[x]
```

Exceptions: none

EXI8 – Extend Immediate by 8 Bits

Description:

Extend the immediate field of the following instruction by 8 bits beginning at bit 24. The 32-bit immediate result is sign extended to the machine width.

Instruction Format: EXI8

15 9	8	7 1	0
Imm$_{31..25}$	v	23h$_7$	I$_{24}$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Exceptions:

Notes:

EXI24 – Extend Immediate by 24 Bits

Description:

Extend the immediate field of the following instruction by 24 bits beginning at bit 24. The 48-bit immediate result is sign extended to the machine width.

Instruction Format: EXI23

31 9	8	7 1	0
Imm$_{47..25}$	v	24h$_7$	I$_{24}$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Exceptions:

Notes:

EXI40 – Extend Immediate by 40 Bits

Description:

Extend the immediate field of the following instruction by 40 bits beginning at bit 24. The 64-bit immediate result is sign extended to the machine width.

Instruction Format: EXI40

47	9	8	7	2	0
Immediate$_{63..25}$		v	25h$_7$		I$_{24}$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Exceptions:

Notes:

EXI56 – Extend Immediate by 56 Bits

Description:

Extend the immediate field of the following instruction by 56 bits beginning at bit 24. The 80-bit immediate result is sign extended to the machine width.

Instruction Format: EXI56

63	9	8	7	1	0
Immediate$_{79..25}$		v	26h$_7$		I$_{24}$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Exceptions:

Notes:

EXIM – Extend Immediate Middle

Description:

Extend the immediate field of the following instruction by 55 bits beginning at bit 80. The EXIM prefix should be used before other immediate extension prefixes to ensure the constant is properly built.

Instruction Format: EXIM

63 9	8	7 0
Immediate$_{134..80}$	v	50h$_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Exceptions:

Example:

```
EXIM 0x12345678
EXI56 0x90123456
ADDI r1,r2,0x123
```

Notes:

LDI – Load Immediate

Description:

This is an alternate mnemonic for the ADDI instruction where register Ra is zero. Add register zero and a sign extended immediate value and place the sum in the target register. If the Ra register field is zero then the value zero is used for Ra.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	0$_6$	Rt$_5$	0	04h$_8$

Instruction Format: RIL

47 45 44 43 19	18 14	13 9	8	7 0
Immediate$_{28..0}$	0$_5$	Rt$_5$	0	D4h$_8$

Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	0$_5$	Rt$_5$	1	D4h$_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = immediate

Exceptions:

Notes:

MAX – 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

47 41	39 38	37	3635	34 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$29h_7$	m_3	z	$\sim_2$	Sz_4	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

Clock Cycles: 1

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

Instruction Format: R3

47 41	39 38	37	3635	34 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$28h_7$	m_3	z	$\sim_2$	Sz_4	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Operation:

```
IF (Ra < Rb and Ra < Rc)
        Rt = Ra
else if (Rb < Rc)
        Rt = Rb
else
        Rt = Rc
```

MKBOOL – Make Boolean

Description:

This instruction is an alternate mnemonic for the CMOVNZ instruction. This instruction places a zero in the target register if the source register is zero, otherwise one is placed in the target register. This instruction reduces the source operand to a Boolean value.

Integer Instruction Format: R3

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
10h$_7$	m$_3$	z	4$_3$	4$_3$	0	0$_5$	0	1$_5$	Ra$_5$	Rt$_5$	v	AAh$_8$

Operation:

Rt = Ra ==0 ? 0 : 1

Execution Units: Integer ALU

Clock Cycles: 1

Exceptions: none

Notes:

MOV – Move Register-Register

Description:

This is an alternate mnemonic for the OR instruction. This instruction moves one general purpose register to another.

Instruction Format:

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
0_2	m_3	z	~	0	0_5	Ra_5	Rt_5	v	$13h_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra

MOVSXB – Move Byte, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a byte from one general purpose register to another.

Instruction Format:

47 41	4038	37	36 34	33 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$05h_7$	m_3	z	4_3	4_3	0	7_5	0	0_5	Ra_5	Rt_5	v	AAh_8

0A120E0000AA

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVSXO – Move Octa, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a octa from one general purpose register to another.

Instruction Format:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	m_3	z	2_2	63_6	2_2	0_6	Ra_6	Rt_6	v	AAh_8

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVSXT – Move Tetra, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a tetra from one general purpose register to another.

Instruction Format:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	m_3	z	2_2	31_6	2_2	0_6	Ra_6	Rt_6	v	AAh_8

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

MOVSXW – Move Wyde, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a wyde from one general purpose register to another.

Instruction Format:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
05h$_7$	m$_3$	z	2$_2$	15$_6$	2$_2$	0$_6$	Ra$_6$	Rt$_6$	v	AAh$_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra

MOVZXB – Move Byte, Zero Extend

Description:

This is an alternate mnemonic for the BFEXTU instruction. This instruction moves a byte from one general purpose register to another.

Instruction Format:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
04h$_7$	m$_3$	z	2$_2$	7$_6$	2$_2$	0$_6$	Ra$_6$	Rt$_6$	v	AAh$_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra

MOVZXT – Move Tetra, Zero Extend

Description:

This is an alternate mnemonic for the BFEXTU instruction. This instruction moves a tetra-byte from one general purpose register to another.

Instruction Format:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	m_3	z	2_2	31_6	2_2	0_6	Ra_6	Rt_6	v	AAh_8

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra

MOVZXW – Move Wyde, Zero Extend

Description:

This is an alternate mnemonic for the BFEXTU instruction. This instruction moves a wyde from one general purpose register to another.

Instruction Format:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	m_3	z	2_2	15_6	2_2	0_6	Ra_6	Rt_6	v	AAh_8

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra

MUL – Multiply

Description:

Multiply two values add a third value. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. All the operands are treated as signed 128-bit values, the result is a signed result.

The carry will be zero unless made available by the CARRY instruction.

Integer Instruction Format: R3

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$06h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

20 clock cycles

Exceptions: none

Execution Units: ALU

Operation

Rt = Ra * Rb + Rc + carry

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] * Vb[x] + Vc[x]
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

Exceptions: overflow, if enabled

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.

Integer Instruction Format: R3

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$0Fh_7$	m_3	z	0_2	0_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

4 clock cycles

Exceptions: none

Execution Units: ALU

Operation

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

MULI – Multiply Immediate

Description:

Multiply two values. The first operand must be in a register. The second operand is an immediate value specified in the instruction. Both the operands are treated as signed values, the result is a signed result.

Integer Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	Ra$_5$	Rt$_5$	0	06h$_8$

20 clock cycles

Integer Instruction Format: RIL

47 45 44 43 19	18 14	13 9	8	7 0
Immediate$_{28..0}$	Ra$_5$	Rt$_5$	0	D2h$_8$

20 clock cycles

Integer Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	Ra$_5$	Rt$_5$	1	D2h$_8$

20 clock cycles

Execution Units: ALU

Operation:

Rt = Ra * Immediate

Vector Operation

```
for x = 0 to VL - 1
    if (Vm0[x]) Vt[x] = Va[x] * Vb[x]
    else Vt[x] = Vt[x]
```

Exceptions: none

MULF – Fast Unsigned Multiply

Description:

Multiply two values located in registers Ra and Rb and add the value of Rc. All operands are treated as unsigned values. The result is an unsigned result. The fast multiply multiplies only the low order 24 bits of the first operand times the low order 16 bits of the second. The result is a 40-bit unsigned product.

Integer Instruction Format: R2

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$15h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Operation:

Rt = Ra * Rb + Rc + carry

Execution Units: ALU

Clock Cycles: 1

Exceptions: none

MULFI – Fast Unsigned Multiply Immediate

Description:

Multiply two values. The first operand is in register Ra. The second operand is an immediate value specified in the instruction. Both the operands are treated as unsigned values. The result is an unsigned result. The fast multiply multiplies only the low order 24 bits of the first operand times the low order 13 bits of the second. The result is a 37-bit unsigned product.

Integer Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	Ra$_5$	Rt$_5$	0	15h$_8$

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

Execution Units: ALU

Clock Cycles: 1

Exceptions: none

MULU – Multiply Unsigned

Description:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as signed values, the result is a signed result.

Integer Instruction Format: R3

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
1Eh$_7$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

20 clock cycles

Exceptions: none

Execution Units: ALU

Operation

Rt = Ra * Rb + Rc + carry

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] * Vb[x] + Vc[x]
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

Exceptions: overflow, if enabled

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:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
34h$_7$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

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:

Perform a bitwise ‘nand’ operation between operands.

Integer Instruction Format: R2

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
00h$_7$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

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

Operation:

Rt = ~(Ra & Rb & Rc)

Exceptions: none

NEG - Negate

Description:

This instruction takes the negative of a value contained in a register Ra.

Instruction Format: R1

31 25	24 22	21	2019	18 14	13 9	8	7 0
$05h_7$	m_3	z	Sz_2	Ra_5	Rt_5	v	$01h_8$

Scalar Operation

Rt = - Rb

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = -Vb[x]
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

Notes

Exceptions:

NOR – Bitwise Nor

Description:

Perform a bitwise ‘nor’ operation between operands.

Integer Instruction Format: R2

47 41	40 38	37	36 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$01h_7$	m_3	z	$\sim_6$	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

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

Operation:

Rt = ~(Ra | Rb | Rc)

Exceptions: none

NOT – Logical Not

Description:

This instruction places a one in the target register if the source register is zero, otherwise zero is placed in the target register. This instruction reduces the source operand to a Boolean value.

Integer Instruction Format: R1

31 25	24 22	21	2019	18 14	13 9	8	7 0
$04h_7$	m_3	z	Sz_2	Ra_5	Rt_5	v	$01h_8$

Operation:

Rt = Ra==0 ? 1 : 0

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

OR – Bitwise Or

Description:

Perform a bitwise ‘or’ operation between operands. The carry input will be zero unless the CARRY instruction is in use.

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	~	Tb	Rb_5	Ra_5	Rt_5	v	$13h_8$

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

Operation:

Rt = Ra | Rb | carry

Integer Instruction Format: R3

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
$09h_7$	m_3	z	$\sim_2$	Sz_3	P	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

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

Operation:

Rt = Ra | Rb | Rc | carry

Execution Units: All Integer ALU’s

Exceptions: none

ORC – Bitwise ‘Or’ with Complement

Description:

Perform a bitwise ‘or’ with complement operation between operands.

Integer Instruction Format: R2

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
$03h_7$	m_3	z	$\sim_2$	Sz_3	P	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

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

Operation:

Rt = Ra | ~Rb | Rc

Execution Units: All Integer ALU’s

Exceptions: none

ORI – Bitwise 'Or' Immediate

Description:

Perform a bitwise or operation between operands. The immediate value is shifted to the left by a multiple of 16 bits before the operation takes place. The immediate is zero extended to the machine width and padded with zeros on the right. This instruction may be used to 'or' a large constant or build a 64-bit constant in a register.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	Ra$_5$	Rt$_5$	v	09h$_8$

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

Instruction Format: RIL

47 45 44 43 19	18 14	13 9	8	7 0
Immediate$_{28..0}$	Ra$_5$	Rt$_5$	0	D9h$_8$

Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	Ra$_5$	Rt$_5$	1	D9h$_8$

Operation

Rt = Ra | Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] | Immediate
    else Vt[x] = Vt[x]
```

Execution Units: All Integer ALU's

Exceptions: none

ORN – Bitwise ‘Or’ with Complement

Description:

This is an alternate mnemonic for the ORC instruction. Perform a bitwise ‘or’ with complement operation between operands.

Integer Instruction Format: R2

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
03h$_7$	m$_3$	z	~$_2$	Sz$_3$	P	Tc	Rc$_5$	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	02h$_8$

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

Operation:

Rt = Ra | ~Rb | Rc

Exceptions: none

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: R3

47 41	40 38	37	36 31	30	29 25	24	23 19	18 14	13 9	8	7 0
$14h_7$	m_3	z	$\sim_6$	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

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. Bits may be reversed in individual bytes, wydes, tetras or octas.

Integer Instruction Format: R1

31 25	24 22	21	20 19	18 14	13 9	8	7 0
$28h_7$	m_3	z	Sz_2	Ra_5	Rt_5	v	$01h_8$

v: 0 = scalar, 1 = vector op

Sz_2	Ext.	Meaning
0	.BP	reverse order within bytes (byte parallel)
1	.WP	reverse order within wydes (wyde parallel)
2	.TP	reverse order within tetras (tetra parallel)
3	.OP	reverse order within octas (octa parallel)

Operation:

Vector Operation

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

ROL – Rotate Left

Description:

Left rotate an operand value by an operand value and place the result in the target register. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	P	Tb	Rb_5	Ra_5	Rt_5	v	$5Bh_8$

0=128 bits,1=64 bits,2=32 bits

Integer Instruction Format: SHIFTI

47 41	49 38	37	36 29	28 26	25	24	23 19	18 14	13 9	8	7 0
$43h_7$	m_3	z	Amt_8	Sz_3	P	0	0_5	Ra_5	Rt_5	v	$02h_8$

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

0=128 bits,1=64 bits,2=32 bits,3=16 bits

Operation Size: .w, .t, .o, .h, .wp, .tp, .op, .hp

Execution Units: integer ALU

Exceptions: none

Example:

ROR – Rotate Right

Description:

Right rotate an operand value by an operand value and place the result in the target register. The first operand must be in a register specified by the Ra. The second operand may be either a register specified by the Rc field of the instruction, or an immediate value.

This is an alternate mnemonic for the SRLP instruction.

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	P	Tb	Rb_5	Ra_5	Rt_5	v	$5Ch_8$

0=128 bits,1=64 bits,2=32 bits

Integer Instruction Format: SHIFTI

47 41	49 38	37	36 29	28 26	25	24	23 19	18 14	13 9	8	7 0
$44h_7$	m_3	z	Amt_8	Sz_3	P	0	0_5	Ra_5	Rt_5	v	$02h_8$

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

0=128 bits,1=64 bits,2=32 bits,3=16 bits

Operation Size: .w, .t, .o, .h, .wp, .tp, .op, .hp

Execution Units: integer ALU

Exceptions: none

Example:

SEQ – Set if Equal

Description:

The set instruction places 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is equal to a second operand in register (Rb) then the target register is set to one, otherwise the target register is set to a zero.

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	P	Tb	Rb_5	Ra_5	Rt_5	v	$44h_8$

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

Operation:

```
if (Ra == Rb)
        Rt = Rc
else
        Rt = 0
```

SEQI – Set if Equal Immediate

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is equal to a second operand, which is an immediate constant then the target register is set to a one, otherwise the target register is set to a zero. The immediate constant is sign extended to the machine width.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
Immediate$_{12..0}$	Ra$_5$	Rt$_5$	v	16h$_8$

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

Instruction Format: RIL

47 45 44 43 19	18 14	13 9	8	7 0
Immediate$_{28..0}$	Ra$_5$	Rt$_5$	0	D6h$_8$

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

Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m$_3$	z	Immediate$_{24..0}$	Ra$_5$	Rt$_5$	1	D6h$_8$

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

SGE / SLE – Set if Greater Than or Equal

Description:

The set instruction places Rc or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than or equal to a second operand in register (Rb) then the target register is set to the value in Rc, otherwise the target register is set to a zero. The operands are treated as signed values.

This instruction may also be used to test less than or equal to by swapping the operands.

Instruction Format: R3

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

Operation:

```
if (Ra >= Rb)
        Rt = Rc
else
        Rt = 0
```

SGEU / SLEU – Set if Greater Than or Equal Unsigned

Description:

The set instruction places Rc or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than or equal to a second operand in register (Rb) then the target register is set to the value in Rc, otherwise the target register is set to a zero. The operands are treated as unsigned values.

This instruction may also be used to test less than or equal to by swapping the operands.

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$23h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Operation:

```
if (Ra >= Rb)
        Rt = Rc
else
        Rt = 0
```

SGTI – Set if Greater Than Immediate

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is equal to a second operand, which is an immediate constant then the target register is set to a one, otherwise the target register is set to a zero.

Instruction Format: RI

31 21	20 15	14 9	8	7 0
Immediate$_{10..0}$	Ra$_6$	Rt$_6$	v	1Bh$_8$

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

SGTIL – Set if Greater Than Immediate Low

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than a second operand, which is an immediate constant then the target register is set to a one, otherwise the target register is set to a zero. The immediate constant is sign extended to the machine width.

Instruction Format: RIL

47 45 44 43 21	20 15	14 9	8	7 0
Immediate$_{26..0}$	Ra$_6$	Rt$_6$	0	DBh$_8$

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

Instruction Format: RILV

47 45	44	43 21	20 15	14 9	8	7 0
m$_3$	z	Immediate$_{22..0}$	Ra$_6$	Rt$_6$	1	DBh$_8$

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

SGTUI – Set if Greater Than Immediate Unsigned

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is equal to a second operand, which is an immediate constant then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as unsigned values.

Instruction Format: RI

31 21	20 15	14 9	8	7 0
Immediate$_{10..0}$	Ra$_6$	Rt$_6$	v	1Fh$_8$

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

SGTUIL – Set if Greater Than Immediate Unsigned Low

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than a second operand, which is an immediate constant then the target register is set to a one, otherwise the target register is set to a zero. The immediate constant is zero extended to the machine width. The operands are treated as unsigned values.

Instruction Format: RIL

47 45 44 43 21	20 15	14 9	8	7 0
Immediate$_{26..0}$	Ra$_6$	Rt$_6$	0	DFh$_8$

1 clock cycle

Instruction Format: RILV

47 45	44	43 21	20 15	14 9	8	7 0
m$_3$	z	Immediate$_{22..0}$	Ra$_6$	Rt$_6$	1	DFh$_8$

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

SLEI – Set if Less Than or Equal Immediate

Description:

This instruction is an alternate mnemonic for the SLTI instruction where the constant has been adjusted by one. For instance, SLEI $t0,$a0,#4 is the same as SLTI $t0,$a0,#5. The assembler will adjust the constant and use the SLTI instruction.

Instruction Format: RI

31 21	20 15	14 9	8	7 0
Immediate$_{10..0}$	Ra$_6$	Rt$_6$	v	18h$_8$

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

SLL –Shift Left Logical

Description:

Left shift an operand value by an operand value and place the result in the target register. Zeros 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

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	P	Tb	Rb_5	Ra_5	Rt_5	v	$58h_8$

0=128 bits,1=64 bits,2=32 bits

Instruction Format: RI6

31 30	29 27	26	25	24 19	18 14	13 9	8	7 0
Sz_2	m_3	z	P	Imm_6	Ra_5	Rt_5	v	$6Ch_8$

0=128 bits,1=64 bits,2=32 bits

Integer Instruction Format: SHIFTI

47 41	49 38	37	36 29	28 26	25	24	23 19	18 14	13 9	8	7 0
$40h_7$	m_3	z	Amt_8	Sz_3	P	0	0_5	Ra_5	Rt_5	v	$02h_8$

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

0=128 bits,1=64 bits,2=32 bits,3=16 bits,5=2x64 bits,6=4x32 bits,7=8x16 bits

Operation Size: .w, .t, .o, .h, .wp, .tp, .op, .hp

Operation:

Rt = Ra << Rb

Operation Size: .o

Execution Units: integer ALU

Exceptions: none

Example:

SLT / SGT – Set if Less Than

Description:

The set instruction places 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than a second operand in register (Rb) then the target register is set to one, otherwise the target register is set to a zero. The operands are treated as signed values.

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	P	Tb	Rb_5	Ra_5	Rt_5	v	$4Eh_8$

Operation:

```
if (Ra < Rb)
        Rt = 1
else
        Rt = 0
```

SLTI – Set if Less Than Immediate

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than a second operand, which is an immediate constant then the target register is set to a one, otherwise the target register is set to a zero. The immediate constant is sign extended to the machine width.

Instruction Format: RI

31 21	20 15	14 9	8	7 0
Immediate$_{10..0}$	Ra$_6$	Rt$_6$	v	18h$_8$

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

Instruction Format: RIL

47 45 44 43 21	20 15	14 9	8	7 0
Immediate$_{26..0}$	Ra$_6$	Rt$_6$	0	D3h$_8$

1 clock cycle

Instruction Format: RILV

47 45	44	43 21	20 15	14 9	8	7 0
m$_3$	z	Immediate$_{22..0}$	Ra$_6$	Rt$_6$	1	D3h$_8$

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

Operation:

```
if (Ra < Rb)
        Rt = 1
else
        Rt = 0
```

SLTU / SGTU – Set if Less Than Unsigned

Description:

The set instruction places Rc or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than a second operand in register (Rb) then the target register is set to the value of Rc, otherwise the target register is set to a zero. The operands are treated as unsigned values.

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$22h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Operation:

```
if (Ra < Rb)
        Rt = Rc
else
        Rt = 0
```

SNE – Set if Not Equal

Description:

The set instruction places 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is not equal to a second operand in register (Rb) then the target register is set to one, otherwise the target register is set to a zero.

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	P	Tb	Rb_5	Ra_5	Rt_5	v	$45h_8$

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

Operation:

```
if (Ra <> Rb)
        Rt = Rc
else
        Rt = 0
```

SNEI – Set if Not Equal Immediate

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is not equal to a second operand, which is an immediate constant then the target register is set to a one, otherwise the target register is set to a zero. The immediate constant is sign extended to the machine width.

Instruction Format: RI

31 19	18 14	13 9	8	7 0
$\text{Immediate}_{12..0}$	Ra_5	Rt_5	v	17h_8

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

Instruction Format: RIL

47 45 44 43 19	18 14	13 9	8	7 0
$\text{Immediate}_{28..0}$	Ra_5	Rt_5	0	D7h_8

1 clock cycle

Instruction Format: RILV

47 45	44	43 19	18 14	13 9	8	7 0
m_3	z	$\text{Immediate}_{24..0}$	Ra_5	Rt_5	1	D7h_8

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

SRA –Shift Right Arithmetic

Description:

Right shift an operand value by an operand value and place the result in the target register. The most significant bit is preserved. 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

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	P	Tb	Rb_5	Ra_5	Rt_5	v	$5Ah_8$

0=128 bits,1=64 bits,2=32 bits

Integer Instruction Format: SHIFTI

47 41	49 38	37	36 29	28 26	25	24	23 19	18 14	13 9	8	7 0
$42h_7$	m_3	z	Amt_8	Sz_3	P	0	0_5	Ra_5	Rt_5	v	$02h_8$

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

0=128 bits,1=64 bits,2=32 bits,3=16 bits,5=2x64 bits,6=4x32 bits,7=8x16 bits

Operation Size: .w, .t, .o, .h, .wp, .tp, .op, .hp

Execution Units: integer ALU

Exceptions: none

Example:

SRL –Shift Right Logical

Description:

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

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz$_2$	m$_3$	z	P	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	59h$_8$

0=128 bits,1=64 bits,2=32 bits

Integer Instruction Format: SHIFTI

47 41	49 38	37	36 29	28 26	25	24	23 19	18 14	13 9	8	7 0
41h$_7$	m$_3$	z	Amt$_8$	Sz$_3$	P	0	0$_5$	Ra$_5$	Rt$_5$	v	02h$_8$

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

0=128 bits,1=64 bits,2=32 bits,3=16 bits,5=2x64 bits,6=4x32 bits,7=8x16 bits

Operation Size: .w, .t, .o, .h, .wp, .tp, .op, .hp

Execution Units: integer ALU

Exceptions: none

Example:

SUB – Subtract

Description:

Subtract Rb from Ra and place the difference in the target register Rt. 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 mask register is ignored for scalar instructions.

Instruction Format: R2

30 31	29 27	26	2524	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	Tb_2	Rb_5	Ra_5	Rt_5	v	$0Dh_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra - Rb

Exceptions: none

Notes:

SUBFI – Subtract from Immediate

Description:

Subtract a register from a sign extended immediate value and place the difference in the target register. For the vector instruction both Ra and Rt are vector registers.

Instruction Format: RI

31 21	20 15	14 9	8	7 0
$\text{Immediate}_{10..0}$	Ra_6	Rt_6	v	05h_8

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Immediate - Ra

Exceptions: none

Notes:

SXB – Move Byte, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a byte from one general purpose register to another.

Instruction Format:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
05h_7	m_3	z	2_2	7_6	2_2	0_6	Ra_6	Rt_6	v	AAh_8

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra

SXT – Move Tetra, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a tetra from one general purpose register to another.

Instruction Format:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	m_3	z	2_2	31_6	2_2	0_6	Ra_6	Rt_6	v	AAh_8

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra

SXW – Move Wyde, Sign Extend

Description:

This is an alternate mnemonic for the BFEXT instruction. This instruction moves a wyde from one general purpose register to another.

Instruction Format:

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$05h_7$	m_3	z	2_2	15_6	2_2	0_6	Ra_6	Rt_6	v	AAh_8

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra

UTF21NDX – UTF21 Index

Description:

This instruction searches Ra, which is treated as an array of three UTF21 characters, for a 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 +2. The index of the first found character is returned (closest to zero).

If a vector UTF21NDX instruction is issued and the target is a scalar register then the instruction searches all the vector elements and returns a value which varies from -1 to +191 in the scalar register. Thus, UTF21NDX may be used to determine the length of a null termination string in the vector register.

Instruction Format: R2

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
1Ch$_7$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Rb in Ra)

Exceptions: none

UTF21NDXI – UTF21 Index

Description:

This instruction searches Ra, which is treated as an array of three UTF21 character, for a value specified by an immediate value and places the index of the value 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 +2. The index of the first found character is returned (closest to zero).

If a vector UTF21NDX instruction is issued and the target is a scalar register then the instruction searches all the vector elements and returns a value which varies from -1 to +191 in the scalar register. Thus, UTF21NDX may be used to determine the length of a null termination string in the vector register.

Only the first 2^{20} UTF21 characters may be searched for by the immediate form of the instruction.

Instruction Format: RI

47 45	44	43 21	20 15	14 9	8	7 0
m_3	z	$Immediate_{22..0}$	Ra_6	Rt_6	v	$57h_8$

Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Imm_{21} in Ra)

Exceptions: none

WYDENDX – WYDE Index

Description:

This instruction searches Ra, which is treated as an array of four 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 wyde. The index result may vary from -1 to +3. The index of the first found wyde is returned (closest to zero).

If a vector WYDENDX instruction is issued and the target is a scalar register then the instruction searches all the vector elements and returns a value which varies from -1 to +255 in the scalar register. Thus, WYDENDX may be used to determine the length of a null termination string in the vector register.

Instruction Format: R2

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
1Bh$_7$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Rb in Ra)

Exceptions: none

WYDENDXI – Wyde Index

Description:

This instruction searches Ra, which is treated as an array of four wydes, for a value specified by an immediate value and places the index of the value 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 wyde. The index result may vary from -1 to +3. The index of the first found wyde is returned (closest to zero).

If a vector WYDENDX instruction is issued and the target is a scalar register then the instruction searches all the vector elements and returns a value which varies from -1 to +255 in the scalar register. Thus, WYDENDX may be used to determine the length of a null termination string in the vector register.

Instruction Format: RI

47 45	44	43 21	20 15	14 9	8	7 0
m_3	z	$Immediate_{22..0}$	Ra_6	Rt_6	v	$56h_8$

Clock Cycles: 1

Execution Units: Integer ALU

Operation:

Rt = Index of (Imm_8 in Ra)

Exceptions: none

XNOR – Bitwise Exclusive Nor

Description:

Perform a bitwise ‘nor’ operation between operands. This is an alternate mnemonic for the ENOR instruction.

Integer Instruction Format: R2

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
$02h_7$	m_3	z	$\sim_2$	Sz_3	P	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

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

Exceptions: none

XOR – Bitwise Exclusive Or

Description:

Perform a bitwise ‘or’ operation between operands.

Instruction Format: R2

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
Sz_2	m_3	z	~	Tb	Rb_5	Ra_5	Rt_5	v	$1Eh_8$

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

Operation:

Rt = Ra ^ Rb ^ carry

Integer Instruction Format: R3

47 41	40 38	37	36 35	34 32	31	30	29 25	24	23 19	18 14	13 9	8	7 0
$0Ah_7$	m_3	z	$\sim_2$	Sz_3	P	Tc	Rc_5	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

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

Operation:

Rt = Ra ^ Rb ^ Rc

Execution Units: All Integer ALU’s

Exceptions: none

XORI – Bitwise Exclusive ‘Or’ Immediate

Description:

Perform a bitwise exclusive ‘or’ operation between operands. The immediate is zero extended to the machine width.

Instruction Format: RI

31 21	20 15	14 9	8	7 0
Immediate$_{10..0}$	Ra$_6$	Rt$_6$	v	0Ah$_8$

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

Operation

Rt = Ra ^ Immediate

Vector Operation

```
for x = 0 to VL-1
    if (Vm0[x]) Vt[x] = Va[x] ^ Immediate
    else Vt[x] = Vt[x]
```

Execution Units: All Integer ALU’s

Exceptions: none

XORIL – Bitwise Exclusive ‘Or’ Immediate Low

Description:

Bitwise 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: RIL

47 45 44 43 21	20 15	14 9	8	7 0
Immediate$_{26..0}$	Ra$_6$	Rt$_6$	0	DAh$_8$

Clock Cycles: 1

Instruction Format: RILV

47 45	44	43 21	20 15	14 9	8	7 0
m$_3$	z	Immediate$_{22..0}$	Ra$_6$	Rt$_6$	1	DAh$_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra ^ immediate

Exceptions:

Notes:

Floating-Point Instructions

Overview

All floating-point operations take place in double precision unless otherwise noted.

FABS – Absolute Value

Description:

This instruction takes the absolute value of a register and places the result in a target register. The values are treated as double precision floating-point values. The sign bit of the number is cleared, no rounding of the number takes place.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
20h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	61h$_8$

v: 0 = scalar, 1 = vector op

Operation:

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

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Rt[x] = Ra[x] < 0 ? -Ra[x] : Ra[x]
    else if (z) Rt[x] = 0
    else Rt[x] = Rt[x]
```

Execution Units: F

Clock Cycles: 1

Exceptions: none

Notes:

FADD – 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 mask register is ignored for scalar instructions. The values are treated as double precision floating-point values.

Instruction Format: R2

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	m_3	z	$\sim_5$	Rm_3	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

FCLASS – Classify Value

Description:

FCLASS classifies the value in register Ra and returns the information as a bit vector in the integer register Rt.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
1Eh$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	61h$_8$

v: 0 = scalar, 1 = vector op

Bit in Rt	Meaning
0	1 = negative infinity
1	1 = negative number
2	1 = negative subnormal number
3	1 = negative zero
4	1 = positive zero
5	1 = positive subnormal number
6	1 = positive number
7	1 = positive infinity
8	1 = signalling nan
9	1 = quiet nan
10 to 62	not used
63	1 = negative, 0 = positive number

FCMP – Compare

Description

Compare two registers and return the relationship between them.

Integer Instruction Format: R2

Both values are treated as double precision (64-bit) floating point numbers. The result is returned as a float value of -1.0, 0.0 or +1.0. If the comparison is unordered 2.0 is returned.

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$10h_7$	m_3	z	$\sim_5$	$\sim_3$	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

1 clock cycle

Operation:

Rt = Ra < Rb ? –1 : Ra = Rb ? 0 : 1

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] < Vb[x] ? –1 : Va[x]=Vb[x] ? 0 : 1
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

FCMPB – Compare

Description

Compare two registers and return the relationship between them.

Integer Instruction Format: R2

Both values are treated as double precision (64-bit) floating point numbers. The value returned is a bit vector as outlined in the table below. Note that the less than status is returned in both bits 1 and 63 so that a BLT may be used.

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
15h$_7$	m$_3$	z	~$_5$	~$_3$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	62h$_8$

1 clock cycle

The float comparison returns a bit vector containing the status of all possible relationships. This may then be tested with the BBS instruction.

Rt bit	Meaning
0	= equal
1	< less than
2	<= less than or equal
3	< magnitude less than
4	unordered
5 to 7	zero (reserved)
8	< > not equal
9	>= greater than or equal
10	> greater than
11	>= magnitude greater than or equal
12	ordered
13 to 62	zero (reserved)
63	less than

Operation:

Vector Operation

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: F1

31 25	24 22	21	20 15	14 9	8	7 0
$11h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$61h_8$

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 – Divide Register-Register

Description:

Divide two operand values and place the result in the target register. The first operand must be in a register specified by the Ra field of the instruction. The second operand may be a register specified by the Rb field of the instruction. The values are treated as double precision floating-point values.

Instruction Format: R2

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$09h_7$	m_3	z	$\sim_5$	Rm_3	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra / Rb

Exceptions:

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: F1

31 25	24 22	21	20 15	14 9	8	7 0
$13h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$61h_8$

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: F1

31 25	24 22	21	20 15	14 9	8	7 0
$12h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$61h_8$

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

FFINITE – Number is Finite

Description:

Test the value in Ra to see if it’s a finite number and return Z=1 or Z = 0 in register Rt.

Integer Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
0Fh$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	61h$_8$

v: 0 = scalar, 1 = vector op

Clock Cycles: 1

Execution Units: Floating Point

Example:

```
finite    $cr1,$f7
```

FMA – Floating Point Multiply Add

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$0h_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$63h_8$

Operation:

Rt = Ra * Rb + Rc

Clock Cycles: 30

Execution Units: All Floating Point

FNMA – Floating Point Negate Multiply Add

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$2h_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$63h_8$

Operation:

Rt = -Ra * Rb + Rc

Clock Cycles: 30

Execution Units: All Floating Point

FNMS – Floating Point Negate Multiply Subtract

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$3h_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$63h_8$

Operation:

Rt = -Ra * Rb - Rc

Clock Cycles: 30

Execution Units: All Floating Point

FMAN – Mantissa of Number

Description:

This instruction provides the mantissa of a double precision floating point number contained in a general-purpose register as a 52-bit zero extended result. The hidden bit of the floating-point number remains hidden.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
07h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	61h$_8$

v: 0

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

FMS – Floating Point Multiply Subtract

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$1h_4$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$63h_8$

Operation:

Rt = Ra * Rb - Rc

Clock Cycles: 30

Execution Units: All Floating Point

FMUL – Floating point multiplication

Description:

Multiply two double precision floating point numbers in registers Ra and Rb and place the result into target register Rt.

Instruction Format: R2

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
08h$_7$	m$_3$	z	~$_5$	Rm$_3$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	62h$_8$

Clock Cycles: 25

Execution Units: All Floating Point

FNEG – Negate Register

Description:

This instruction negates a double precision floating point number contained in a general-purpose register. The sign bit of the number is inverted. No rounding takes place.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
22h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	61h$_8$

v: 0 = scalar, 1 = vector op

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = -Ra

FRM – Set Floating Point Rounding Mode

Description:

This instruction sets the rounding mode bits in the floating-point control register (FPSCR). The rounding mode bits are set to the bitwise 'or' of an immediate field in the instruction and the contents of register Ra. Either Ra or the immediate field should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
14h$_7$	~$_3$	~	Ra$_6$	uimm$_6$	0	61h$_8$

Execution Units: All Floating Point

Operation:

FPSCR.RM = Ra | Immediate

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

31 25	24 22	21	20 15	14 9	8	7 0
01h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	61h$_8$

Clock Cycles: 5

Execution Units: Floating Point

Notes:

The estimate is only accurate to about 3%. The estimate is performed in single precision (32-bit) floating point, then converted to a 64-bit format. That means that input values must in the range of a 32-bit floating point number. Values outside of this range will return infinity or zero as a result.

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

FSEQ - Float Set if Equal

Description:

The register compare instruction compares two registers as floating-point values for equality and sets the compare results register as a result. Note that negative and positive zero are considered equal.

Instruction Format: R2

Both values are treated as double precision (64-bit) floating point numbers. The result is returned as an integer value of 1 or 0.

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$11h_7$	m_3	z	$\sim_5$	$\sim_3$	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

1 clock cycle

Clock Cycles: 1

Execution Units: Floating Point

Operation:

```
if Ra == Rb
        Rt= 1
else
        Rt= 0
```

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

31 25	24 22	21	20 15	14 9	8	7 0
$28h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

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

31 25	24 22	21	20 15	14 9	8	7 0
$06h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = sign of (Ra)

FSLT - Float Set if Less Than

Description:

The register compare instruction compares two registers as floating-point values for less than and sets the target register as a result.

Instruction Format: R2

Both values are treated as double precision (64-bit) floating point numbers. The result is returned as an integer value of 1 or 0.

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$12h_7$	m_3	z	$\sim_5$	$\sim_3$	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

1 clock cycle

Clock Cycles: 1

Execution Units: Floating Point

Operation:

```
if Ra < Rb
    Rt = 1
else
    Rt = 0
```

FSNE - Float Set if Not Equal

Description:

The register compare instruction compares two registers as floating-point values for inequality and sets the compare results register as a result. Note that negative and positive zero are considered equal.

Instruction Format: R2

Both values are treated as double precision (64-bit) floating point numbers. The result is returned as an integer value of 1 or 0.

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$14h_7$	m_3	z	$\sim_5$	$\sim_3$	Tb_2	Rb_6	Ra_6	Rt_6	v	$62h_8$

1 clock cycle

Clock Cycles: 1

Execution Units: Floating Point

Operation:

```
if Ra == Rb
        Rt= 1
else
        Rt= 0
```

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, R1L

31 25	24 22	21	20 15	14 9	8	7 0
$08h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

Operation:

Rt = sqrt (Ra)

Clock Cycles: 64 (est).

Execution Units: Floating Point

FSTAT – Get Floating Point Status and Control

Description:

The floating-point status and control register may be read using the FSTAT instruction. The format of the FPSCR register is outlined on the next page.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$0Ch_7$	$\sim_3$	~	$\sim_6$	Rt_6	0	$61h_8$

Execution Units: All Floating Point

Operation:

Rt = FPSCR

Floating Point Status And Control Register Format:

Bit		Symbol	Description
31:29	**RM**	rm	rounding mode (unimplemented)
28	**E5**	inexe	- inexact exception enable
27	**E4**	dbzxe	- divide by zero exception enable
26	**E3**	underxe	- underflow exception enable
25	**E2**	overxe	- overflow exception enable
24	**E1**	invopxe	- invalid operation exception enable
23	**NS**	ns	- non standard floating point indicator
Result Status			
22		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
21	**RA**	rawayz	rounded away from zero (fraction incremented)
20	**SC**	C	denormalized, negative zero, or quiet NaN
19	**SL**	neg <	the result is negative (and not zero)
18	**SG**	pos >	the result is positive (and not zero)
17	**SE**	zero =	the result is zero (negative or positive)
16	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
15	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
14	**X5**	inerx	- inexact result exception occurred (sticky)
13	**X4**	dbzx	- divide by zero exception occurred
12	**X3**	underx	- underflow exception occurred
11	**X2**	overx	- overflow exception occurred
10	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
9	**GX**	gx	- global exception indicator – set if any enabled exception has happened
8	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

Greyed out items are not implemented.

FSUB – 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. The mask register is ignored for scalar instructions. The values are treated as double precision floating-point values.

Instruction Format: R2

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
05h$_7$	m$_3$	z	~$_5$	Rm$_3$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	62h$_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

FTOI – Float to Integer

Description:

This instruction converts a floating-point double value to an integer value.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
02h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	61h$_8$

Clock Cycles: 2

Execution Units: All Floating Point

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

31 25	24 22	21	20 15	14 9	8	7 0
$15h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

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: F1

31 25	24 22	21	20 15	14 9	8	7 0
$10h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$61h_8$

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

ISNAN – Is Not a Number

Description:

Test the value in Ra to see if it’s a nan (not a number) and return true Z=1 or false Z=0 in register Rt.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$0Eh_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

Clock Cycles: 1

Execution Units: Floating Point

Example:

```
isnan    $cr1,$f7
```

ITOF – Integer to Float

Description:

This instruction converts an integer value to a double precision floating point representation.

Instruction Format: F1, F1L

31 25	24 22	21	20 15	14 9	8	7 0
$03h_7$	m_3	z	Ra_6	Rt_6	v	$61h_8$

Clock Cycles: 2

Execution Units: All Floating Point

Decimal Floating-Point Instructions

DFABS – Absolute Value

Description:

This instruction takes the absolute value of a register and places the result in a target register. The values are treated as quad precision decimal floating-point values. The sign bit of the number is cleared, no rounding of the number takes place.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$20h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

v: 0 = scalar, 1 = vector op

Operation:

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

Vector Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Rt[x] = Ra[x] < 0 ? -Ra[x] : Ra[x]
    else if (z) Rt[x] = 0
    else Rt[x] = Rt[x]
```

Execution Units: F

Clock Cycles: 1

Exceptions: none

Notes:

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 mask register is ignored for scalar instructions. The values are treated as quad precision decimal floating-point values.

Instruction Format: R2

47 41	49 38	37	36 35	34 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	m_3	z	$\sim_2$	$\sim_3$	Rm_3	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Clock Cycles: 1

Execution Units: All ALU's

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

DFCX – 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: F1

31 25	24 22	21	20 15	14 9	8	7 0
$11h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$65h_8$

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

DFDIV – Divide Register-Register

Description:

Divide two operand values and place the result in the target register. The first operand must be in a register specified by the Ra field of the instruction. The second operand must be a register specified by the Rb field of the instruction. The values are treated as quad precision decimal floating-point values. This instruction takes many clock cycles to complete but is interruptible.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$09h_7$	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Instruction Format: R2L

47 41	40 36	35 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
$09h_7$	$\sim_5$	Rm_3	m_3	z	Tb_2	Rb_6	Ra_6	Rt_6	v	$76h_8$

Clock Cycles: 3,000 est.

Execution Units: All ALU's

Operation:

Rt = Ra / Rb

Exceptions:

Notes:

DFDIVIDEND_ADJ – Dividend Adjustment for Division

Description:

Takes the dividend and divisor and returns the dividend with the exponent adjusted for a divisor between 0.5 and 1.0. This is in preparation for a Newton-Raphson division routine.

Instruction Format: R3

47 41	40 38	37	36 29	28 27	26 21	20 15	14 9	8	7 0
$20h_7$	m_3	z	$\sim_8$	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Exceptions:

Notes:

DFDIVISOR_ADJ – Divisor Adjustment for Division

Description:

Takes the divisor sets the exponent to the bias value, then scales the significand to between 0.5 and 1.0 adjusting the exponent as needed. This is in preparation for a Newton-Raphson division routine.

Instruction Format: R3

47 41	40 38	37	36 29	28 27	26 21	20 15	14 9	8	7 0
$21h_7$	m_3	z	$\sim_8$	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Exceptions:

Notes:

DFDIVISOR_BITADJ – Divisor Adjustment for Division

Description:

Returns the number of bits the divisor needed to be shifted to the left during scaling. This is in preparation for a Newton-Raphson division routine.

Instruction Format: R3

47 41	40 38	37	36 29	28 27	26 21	20 15	14 9	8	7 0
$21h_7$	m_3	z	$\sim_8$	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Exceptions:

Notes:

DFDX – 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: F1

31 25	24 22	21	20 15	14 9	8	7 0
$13h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$65h_8$

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

DFEX – 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: F1

31 25	24 22	21	20 15	14 9	8	7 0
12h$_7$	~$_3$	~	Ra$_6$	uimm$_6$	0	65h$_8$

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

DFMA – Floating Point Multiply Add

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$00h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$67h_8$

Instruction Format: R3L

55 49	48 44	43 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$00h_7$	$\sim_5$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$77h_8$

Operation:

Rt = Ra * Rb + Rc

Clock Cycles: 30

Execution Units: All Floating Point

DFNMA – Floating Point Negate Multiply Add

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
02h$_7$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	67h$_8$

Instruction Format: R3L

55 49	48 44	43 41	40 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
02h$_7$	~$_5$	Rm$_3$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	77h$_8$

Operation:

Rt = -Ra * Rb + Rc

Clock Cycles: 30

Execution Units: All Floating Point

DFNMS – Floating Point Negate Multiply Subtract

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$02h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$67h_8$

Instruction Format: R3L

55 49	48 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$02h_7$	$\sim_5$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$77h_8$

Operation:

Rt = -Ra * Rb - Rc

Clock Cycles: 30

Execution Units: All Floating Point

DFMAN – Mantissa of Number

Description:

This instruction provides the mantissa of a double precision floating point number contained in a general-purpose register as a 52-bit zero extended result. The hidden bit of the floating-point number remains hidden.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$07h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

v: 0

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = Ra

DFMS – Floating Point Multiply Subtract

Description:

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

Instruction Format: R3

47 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$01h_7$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$67h_8$

Instruction Format: R3L

55 49	48 44	43 41	40 38	37	3635	34 29	28 27	26 21	20 15	14 9	8	7 0
$01h_7$	$\sim_5$	Rm_3	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$77h_8$

Operation:

Rt = Ra * Rb - Rc

Clock Cycles: 30

Execution Units: All Floating Point

DFMUL – Floating point multiplication

Description:

Multiply two quad precision decimal floating point numbers in registers Ra and Rb and place the result into target register Rt.

Instruction Format: R2

47 41	49 38	37	36 35	34 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$08h_7$	m_3	z	$\sim_2$	$\sim_3$	Rm_3	Tb_2	Rb_6	Ra_6	Rt_6	v	$66h_8$

Clock Cycles: 25

Execution Units: All Floating Point

DFNEG – Negate Register

Description:

This instruction negates a double precision floating point number contained in a general-purpose register. The sign bit of the number is inverted. No rounding takes place.

Integer Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$22h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

v: 0 = scalar, 1 = vector op

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = -Ra

DFRES – Reciprocal Approximate

Description:

Returns an approximation of the reciprocal of a number which must be between 0.1 and 1.0. The reciprocal has approximately three decimal digits of accuracy. This is in preparation for a Newton-Raphson division routine.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
$17h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

Clock Cycles: 4

Execution Units: All ALU’s

Operation:

Exceptions:

Notes:

DFRM – Set Floating Point Rounding Mode

Description:

This instruction sets the rounding mode bits in the floating-point control register (FPSCR). The rounding mode bits are set to the bitwise ‘or’ of an immediate field in the instruction and the contents of register Ra. Either Ra or the immediate field should be zero.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$14h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$65h_8$

Execution Units: All Floating Point

Operation:

FPSCR.RM = Ra | Immediate

DFSIGN – 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: F1

31 25	24 22	21	20 15	14 9	8	7 0
$06h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

Clock Cycles: 1

Execution Units: All Floating Point

Operation:

Rt = sign of (Ra)

DFSTAT – Get Floating Point Status and Control

Description:

The floating-point status and control register may be read using the FSTAT instruction. The format of the FPSCR register is outlined on the next page.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
$0Ch_7$	$\sim_3$	~	$\sim_6$	Rt_6	0	$65h_8$

Execution Units: All Floating Point

Operation:

Rt = FPSCR

Floating Point Status And Control Register Format:

Bit		Symbol	Description
31:29	**RM**	rm	rounding mode (unimplemented)
28	**E5**	inexe	- inexact exception enable
27	**E4**	dbzxe	- divide by zero exception enable
26	**E3**	underxe	- underflow exception enable
25	**E2**	overxe	- overflow exception enable
24	**E1**	invopxe	- invalid operation exception enable
23	**NS**	ns	- non standard floating point indicator
Result Status			
22		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
21	**RA**	rawayz	rounded away from zero (fraction incremented)
20	**SC**	C	denormalized, negative zero, or quiet NaN
19	**SL**	neg <	the result is negative (and not zero)
18	**SG**	pos >	the result is positive (and not zero)
17	**SE**	zero =	the result is zero (negative or positive)
16	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
15	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
14	**X5**	inerx	- inexact result exception occurred (sticky)
13	**X4**	dbzx	- divide by zero exception occurred
12	**X3**	underx	- underflow exception occurred
11	**X2**	overx	- overflow exception occurred
10	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
9	**GX**	gx	- global exception indicator – set if any enabled exception has happened
8	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

Greyed out items are not implemented.

DFSUB – 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. The mask register is ignored for scalar instructions. The values are treated as double precision floating-point values.

Instruction Format: R2

39 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
05h$_7$	m$_3$	z	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	66h$_8$

Instruction Format: R2L

47 41	40 36	35 33	32 30	29	28 27	26 21	20 15	14 9	8	7 0
05h$_7$	~$_5$	Rm$_3$	m$_3$	z	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	76h$_8$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

DFTOI – Float to Integer

Description:

This instruction converts a quad precision decimal floating-point value to an integer value.

Instruction Format: F1

31 25	24 22	21	20 15	14 9	8	7 0
02h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	65h$_8$

Clock Cycles: 2

Execution Units: All Floating Point

DFTX – 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: F1

31 25	24 22	21	20 15	14 9	8	7 0
$10h_7$	$\sim_3$	~	Ra_6	$uimm_6$	0	$65h_8$

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

ITODF – Integer to Float

Description:

This instruction converts an integer value to a quad precision decimal floating point representation.

Instruction Format: F1, F1L

31 25	24 22	21	20 15	14 9	8	7 0
$03h_7$	m_3	z	Ra_6	Rt_6	v	$65h_8$

Clock Cycles: 2

Execution Units: All Floating Point

Posit Instructions

PADD – 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 mask register is ignored for scalar instructions. The values are treated as posit values.

Instruction Format: R2

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$04h_7$	m_3	z	$\sim_5$	Rm_3	Tb_2	Rb_6	Ra_6	Rt_6	v	$6Ah_8$

Clock Cycles: 25

Execution Units: Posit Arithmetic

Operation:

Rt = Ra + Rb

Exceptions:

Notes:

PMUL – Posit multiplication

Description:

Multiply two posit numbers in registers Ra and Rb and place the result into target register Rt.

Instruction Format: R2

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
$08h_7$	m_3	z	$\sim_5$	Rm_3	Tb_2	Rb_6	Ra_6	Rt_6	v	$6Ah_8$

Clock Cycles: 25

Execution Units: Posit Arithmetic

PSUB – 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. The mask register is ignored for scalar instructions. The values are treated as posit values.

Instruction Format: R2

47 41	49 38	37	36 32	31 29	28 27	26 21	20 15	14 9	8	7 0
05h$_7$	m$_3$	z	~$_5$	Rm$_3$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	6Ah$_8$

Clock Cycles: 1

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. Note that register indirect with displacement addressing is limited to scalar instructions.

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.

47 19	18 14	13 9	8	7 0
$\text{Displacement}_{28..0}$	Ra_5	Rt_5	0	Opcode_8

Indexed Format

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rt_5	v	Opcode_8

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.

47 19	18 14	13 9	8	7 0
$\text{Displacement}_{28..0}$	Ra_5	Rs_5	0	Opcode_8

Indexed Format

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rs_5	v	Opcode_8

CACHE – Cache Command

Description:

This instruction commands the cache controller to perform an operation. Commands are summarized in the command table below. Commands may be issued to both the instruction and data cache at the same time. The address of the cache line to be invalidated is passed in Ra if needed.

Instruction Format:

47 19	18 14	13 11	10 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	DC$_3$	IC$_2$	0	9Fh$_8$

Commands:

IC$_2$	Mne.	Operation
0	NOP	no operation
1	invline	invalidate line associated with given address
2	invall	invalidate the entire cache (address is ignored)
3		reserved

DC$_3$	Mne.	Operation
0	NOP	no operation
1	enable	enable cache (instruction cache is always enabled)
2	disable	not valid for the instruction cache
3	invline	invalidate line associated with given address
4	invall	invalidate the entire cache (address is ignored)
5 to 7		reserved

Clock Cycles: 3

Execution Units: All ALU’s / Memory

Operation:

Exceptions: DBG

CACHEX – Cache Command

Description:

This instruction commands the cache controller to perform an operation. Commands are summarized in the command table below. Commands may be issued to both the instruction and data cache at the same time.

Instruction Format:

31 29	28	27	26	2524	23 19	18 14	13 11	10 9	8	7 0
m_3	z	C	~	Tb_2	Rb_5	Ra_5	DC_3	IC_2	v	CFh_8

Commands:

IC_2	Mne.	Operation
0	NOP	no operation
1	invline	invalidate line associated with given address
2	invall	invalidate the entire cache (address is ignored)
3		reserved

DC_3	Mne.	Operation
0	NOP	no operation
1	enable	enable cache (instruction cache is always enabled)
2	disable	not valid for the instruction cache
3	invline	invalidate line associated with given address
4	invall	invalidate the entire cache (address is ignored)
5 to 7		reserved

Clock Cycles:

Execution Units: All ALU’s / Memory

Operation:

Exceptions: DBG

CAS – Compare and Swap

Description:

If the contents of the addressed memory cell is equal to the contents of Rb then a sixty-four bit value is stored to memory from the source register Rc. The original contents of the memory cell are loaded into register Rt. The memory address is contained in register Ra. The memory address must be word aligned. If the operation was successful then Rt and Rb will be the same value. The compare and swap operation is an atomic operation; the bus is locked during the load and potential store operation. This operation assumes that the addressed memory location is part of the volatile region of memory and bypasses the data cache.

The stack pointer cannot be used as the target register.

Instruction Format:

47 45	44	43 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
Sel_3	~	$\sim_3$	m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$97h_8$

Operation:

```
Rt = memory [Ra]
if memory[Ra] = Rb
        memory[Ra] = Rc
```

Assembler:

CAS Rt,Rb,Rc,[Ra]

LDB – Load Byte

Description:

An eight-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. For vector instructions Ra is a scalar register.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format: RIL

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rt$_5$	0	80h$_8$

Instruction Format: RILV

C	44	43	43 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rt$_5$	1	80h$_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (memory$_8$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDBU – Load Byte, Unsigned

Description:

An eight-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. For vector instructions Ra is a scalar register.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format: RIL

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rt$_5$	0	81h$_8$

Instruction Format: RILV

C	44	43	43 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rt$_5$	1	81h$_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend (memory$_8$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDBUX – Load Byte Unsigned Indexed

Description:

An eight-bit value is loaded from memory zero extended and placed in the target register. The memory address is the sum of register Ra and register Rb.

Instruction Format:

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rt_5	v	$B1h_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory$_8$[Ra+Rb])

Exceptions: DBE, DBG, LMT, TLB

LDBX – Load Byte Indexed

Description:

An eight-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb. For vector instructions Ra is a scalar register.

Instruction Format:

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rt_5	v	$B0h_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory$_8$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDCTX – Load Context

Description:

All sixty-four general purpose registers are loaded from the context area. The context area is specified by a descriptor selected by the contents of the specified selector register.

Instruction Format:

15 13	12 9	8	7 0
Sel_3	0_4	0	$8Dh_8$

Clock Cycles: 320 (32 memory access) (64 for 8 cache access)

Execution Units: All ALU’s / Memory

Operation:

Exceptions: DBE, DBG, TLB, LMT

LDH – Load Hexi

Description:

A 128-bit value is loaded from memory then placed in the target register. The memory address is the sum of the sign extended displacement and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format: RIL

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rt$_5$	0	88h$_8$

Instruction Format: RIVL

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rt$_5$	1	88h$_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (memory$_{128}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDHS – Load Hexi, Short Format

Description:

A 128-bit value is loaded from memory then placed in the target register. The memory address is the sum of the sign extended displacement and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

31 19	18 14	13 9	8	7 0
Disp$_{12..0}$	Ra$_5$	Rt$_5$	0	89h$_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (memory$_{128}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDHX – Load Hexi-byte Indexed

Description:

A 128-bit value is loaded from memory and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

31 29	28	27	26	2524	23 19	18 14	13 9	8	7 0
m_3	z	C	~	Tb_2	Rb_5	Ra_5	Rt_5	v	$B8h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory$_{64}$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDO – Load Octa

Description:

A sixty-four-bit value is loaded from memory then placed in the target register. The memory address is the sum of the sign extended displacement and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn’t in the cache provided the current memory page is cacheable.

Instruction Format:

47 19	18 14	13 9	8	7 0
$\text{Displacement}_{28..0}$	Ra_5	Rt_5	0	86h_8

Instruction Format: RIVL

47 45	44	43	42 19	18 14	13 9	8	7 0
m_3	z	C	$\text{Displacement}_{23..0}$	Ra_5	Rt_5	1	86h_8

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory_{64}[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDOU – Load Octa Unsigned

Description:

A sixty-four-bit value is loaded from memory then placed in the target register. The memory address is the sum of the sign extended displacement and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rt$_5$	0	87h$_8$

Instruction Format: RIVL

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rt$_5$	1	87h$_8$

Clock Cycles: 10 (one memory access)

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (memory$_{64}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDOX – Load Octa Indexed

Description:

A sixty-four-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

31 29	28	27	26	2524	23 19	18 14	13 9	8	7 0
m_3	z	C	~	Tb_2	Rb_5	Ra_5	Rt_5	v	$B6h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory$_{64}$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDOUX – Load Octa Unsigned Indexed

Description:

A sixty-four-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

31 29	28	27	26	2524	23 19	18 14	13 9	8	7 0
m_3	z	C	~	Tb_2	Rb_5	Ra_5	Rt_5	v	$B7h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (memory$_{64}$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDT – Load Tetra

Description:

A thirty-two-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rt$_5$	0	84h$_8$

Instruction Format: RIVL

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rt$_5$	1	84h$_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (memory$_{32}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDTU – Load Tetra Unsigned

Description:

A thirty-two-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rt$_5$	0	85h$_8$

Instruction Format: RIVL

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rt$_5$	1	85h$_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend (memory$_{32}$[Ra+displacement])

Exceptions: DBE, DBG, LMT, TLB

LDTUX – Load Tetra Unsigned Indexed

Description:

A thirty-two-bit value is loaded from memory zero extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

47 45	44	43	42 35	34 32	31 29	2827	26 21	20 15	14 9	8	7 0
m_3	z	C	$Disp_8$	Sel_3	Sc_3	Tb_2	Rb_6	Ra_6	Rt_6	v	$B5h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend ($memory_{32}$[Ra+Rb*Sc])

Exceptions: DBE, DBG, LMT, TLB

LDTX – Load Tetra Indexed

Description:

A thirty-two-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and register Rb.

Instruction Format:

31 29	28	27	26	2524	23 19	18 14	13 9	8	7 0
m_3	z	C	~	Tb_2	Rb_5	Ra_5	Rt_5	v	$B4h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend ($memory_{32}$[Ra + Rb])

Exceptions: DBE, DBG, TLB

LDW – Load Wyde

Description:

A sixteen-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rt$_5$	0	82h$_8$

Instruction Format: RIVL

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rt$_5$	1	82h$_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = sign extend (memory$_{16}$[Ra+displacement])

Exceptions: DBE, DBG, TLB

LDWU – Load Wyde Unsigned

Description:

A sixteen-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn't in the cache provided the current memory page is cacheable.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rt$_5$	0	83h$_8$

Instruction Format: RIVL

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rt$_5$	1	83h$_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

Rt = zero extend (memory$_{16}$[Ra+offset])

Exceptions: DBE, DBG, TLB

LDWX – Load Wyde Indexed

Description:

A sixteen-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and register Rb.

Instruction Format:

31 29	28	27	26	2524	23 19	18 14	13 9	8	7 0
m_3	z	C	~	Tb_2	Rb_5	Ra_5	Rt_5	v	$B2h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory$_{16}$[Ra + Rb])

Exceptions: DBE, DBG, LMT, TLB

LDWUX – Load Wyde Unsigned Indexed

Description:

A sixteen-bit value is loaded from memory zero extended and placed in the target register. The memory address is the sum of register Ra and register Rb.

Instruction Format:

31 29	28	27	26	2524	23 19	18 14	13 9	8	7 0
m_3	z	C	~	Tb_2	Rb_5	Ra_5	Rt_5	v	$B3h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory$_{16}$[Ra + Rb])

Exceptions: DBE, DBG, TLB

LEA – Load Effective Address

Description:

This instruction is an alternate mnemonic for the ADDI instruction. The virtual address is calculated and placed in the target register. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format: RIL

47 19	18 14	13 9	8	7 0
$\text{Displacement}_{28..0}$	Ra_5	Rt_5	0	$04h_8$

Instruction Format: RIVL

47 45	44	43 19	18 14	13 9	8	7 0
m_3	z	$\text{Displacement}_{24..0}$	Ra_5	Rt_5	1	$04h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = Ra + displacement

Exceptions: none

LEAX – Load Effective Address Indexed

Description:

This instruction is an alternate mnemonic for the ADD instruction. The virtual address is calculated and placed in the target register. The memory address is the sum of register Ra and register Rb.

Instruction Format:

31 30	29 27	26	25	24	23 19	18 14	13 9	8	7 0
0_2	m_3	z	0	Tb	Rb_5	Ra_5	Rt_5	v	$19h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = Ra + Rb

Exceptions: none

VIRT2PHYS – Convert Virtual to Physical Address

Description:

The virtual address in register Ra is converted to the physical address.

Instruction Format:

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$36h_7$	m_3	z	~	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

Clock Cycles: 3

Execution Units: Memory

Operation:

Rt = Virt2Phys(Ra)

Exceptions: none

STB – Store Byte

Description:

An eight-bit value is stored to memory from the source register Rs. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rs$_5$	0	90h$_8$

Instruction Format: RILV

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rs$_5$	1	90h$_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory$_8$[Ra + displacement] = Rs$_{[7..0]}$

Exceptions: DBE, DBG, TLB, LMT

STBX – Store Byte Indexed

Description:

An eight-bit value is stored to memory from register Rs. The memory address is the sum of register Ra and register Rb.

Instruction Format:

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rs_5	v	$C0h_8$

C: 1=compressed vector store

Clock Cycles: 10 (one memory access)

Execution Units: All AGU’s / Memory

Operation:

$memory_8[Ra+Rb] = Rs_{[7:0]}$

Vector Operation:

```
y = 0
for x = 0 to VL – 1
    if (Rb is scalar)
        n = Rb * y
    else
        n = Rb[n]
    IF (Vm[n])
        $memory_8[Ra+n] = Rs[n]_{[7:0]}$
    IF (C)
        y = y + Vm[n]
    ELSE
        y = y + 1
```

Exceptions: DBE, DBG, LMT, TLB

STCTX – Store Context

Description:

All sixty-four general purpose registers are stored to the context area. The context area is specified by a descriptor selected by the contents of the specified selector register.

Instruction Format:

15 13	12 9	8	7 0
Sel$_3$	0$_4$	0	9Dh$_8$

Clock Cycles: 320 (32 memory access)

Execution Units: All ALU's / Memory

Operation:

Exceptions: DBE, DBG, TLB, LMT

Example;

```
# setup the FS to point to the TCB area
MFSEL t0,11              # TCB selector
MTSEL FS,t0
…
# store context
STCTX FS
```

STH – Store Hexi-byte

Description:

A register is stored to memory. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rs$_5$	0	94h$_8$

Instruction Format: RILV

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rs$_5$	1	94h$_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory$_{128}$[Ra+displacement] = Rs

Exceptions: DBE, DBG, TLB, LMT

STHC – Store Hexi, Clear Reservation

Description:

A 128-bit value from the source register Rs is stored to memory if a reservation exists at the target address. The reservation at the target address is cleared. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rs$_5$	0	96h$_8$

Instruction Format: RILV

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rt$_5$	1	96h$_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory$_{128}$[Ra+displacement] = Rb

Exceptions: DBE, DBG, TLB, LMT

STHP – Store Hexi-byte Pair

Description:

A pair of registers is stored to memory. The memory address is the sum of the sign extended displacement and register Ra. The address must be hexi-byte pair aligned. The register pair must be an even, odd pair of registers.

Instruction Format:

47 19	18 14	13 10	9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rs$_4$	0	0	94h$_8$

Instruction Format: RILV

47 45	44	43	42 19	18 14	13 10	9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rs$_4$	0	1	94h$_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory$_{128}$[Ra+displacement] = Rs

memory$_{128}$[Ra+displacement+16] = Rs+1

Exceptions: DBE, DBG, TLB, LMT

STHS – Store Hexi, Short Addressing

Description:

A 128-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

31 19	18 14	13 9	8	7 0
$\text{Disp}_{12..0}$	Ra_5	Rs_5	0	95h_8

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory_{64}[Ra+displacement] = Rb

Exceptions: DBE, DBG, TLB, LMT

STHX – Store Hexi-byte Indexed

Description:

A 128-bit value is stored to memory from register Rs. The memory address is the sum of register Ra and register Rb.

Instruction Format:

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rs_5	v	$C4h_8$

C: 1=compressed vector store

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{128}[Ra+Rb] = Rs$

Exceptions: DBE, DBG, LMT, TLB

STO – Store Octa

Description:

A sixty-four-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rs$_5$	0	93h$_8$

Instruction Format: RILV

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rs$_5$	1	93h$_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory$_{64}$[Ra+displacement] = Rb

Exceptions: DBE, DBG, TLB, LMT

STHCX – Store Hexi, Clear Reservation Indexed

Description:

A 128-bit value from the source register Rs is stored to memory if a reservation exists at the target address. The reservation at the target address is cleared. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rs_5	v	$C4h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory$_{128}$[Ra + Rc] = Rb

Exceptions: DBE, DBG, LMT, TLB

STOO – Store Octa Octet

Description:

An octet of registers is stored to memory. The memory address must be 64-byte aligned. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 45	44 27	26 21	20 15	14 12	11 9	8	7 0
Sel_3	$Displacement_{23..6}$	0_6	Ra_6	Rs_3	1_3	0	$96h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

$memory_{64}$[Ra+displacement+00] = Rs+0

$memory_{64}$[Ra+displacement+08] = Rs+1

$memory_{64}$[Ra+displacement+16] = Rs+2

$memory_{64}$[Ra+displacement+24] = Rs+3

$memory_{64}$[Ra+displacement+32] = Rs+4

$memory_{64}$[Ra+displacement+40] = Rs+5

$memory_{64}$[Ra+displacement+48] = Rs+6

$memory_{64}$[Ra+displacement+56] = Rs+7

Exceptions: DBE, DBG, TLB, LMT

STOOX – Store Octa Octet Indexed

Description:

An octet of registers is stored to memory. The memory address must be 64-byte aligned. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

47 45	44	43	42 35	34 32	31 29	2827	26 21	20 15	14 9	8	7 0
m_3	z	C	$Disp_8$	Sel_3	Sc_3	Tb_2	Rb_6	Ra_6	Rs_6	v	$C6h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{64}$[Ra+Rb*Sc+00] = Rs+0

$memory_{64}$[Ra+Rb*Sc+08] = Rs+1

$memory_{64}$[Ra+Rb*Sc+16] = Rs+2

$memory_{64}$[Ra+Rb*Sc+24] = Rs+3

$memory_{64}$[Ra+Rb*Sc+32] = Rs+4

$memory_{64}$[Ra+Rb*Sc+40] = Rs+5

$memory_{64}$[Ra+Rb*Sc+48] = Rs+6

$memory_{64}$[Ra+Rb*Sc+56] = Rs+7

Exceptions: DBE, DBG, LMT, TLB

STOX – Store Octa Indexed

Description:

A sixty-four-bit value is stored to memory from register Rs. The memory address is the sum of register Ra and scaled register Rb.

Instruction Format:

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rs_5	v	$C3h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$memory_{64}[Ra+Rb*Sc] = Rs$

Exceptions: DBE, DBG, LMT, TLB

STPTR – Store Pointer

Description:

A register is stored to memory. The memory address is the sum of the sign extended displacement and register Ra. A telescopic write of zero bytes to the card table takes place.

Instruction Format:

47 19	18 14	13 9	8	7 0
$\text{Displacement}_{28..0}$	Ra_5	Rs_5	0	A8h_8

Instruction Format: RILV

4745	44	43	42 19	18 14	13 9	8	7 0
m_3	z	C	$\text{Displacement}_{23..0}$	Ra_5	Rt_5	1	A8h_8

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Operation:

```
tmp = Ra + displacement – region.start
do
    tmp = tmp >> 8
    $\text{memory}_8$[tmp+region.cta] = 0
until (tmp = 0)
```

Execution Units: All ALU’s / Memory

Operation:

```
$\text{memory}_{128}$[Ra+displacement+00] = Rs
```

Exceptions: DBE, DBG, TLB, LMT

STPTRX – Store Pointer Indexed

Description:

A register is stored to memory from register Rs. The memory address is the sum of register Ra and register Rb. A telescopic write of zero bytes to the card table takes place.

Instruction Format:

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rs_5	v	$A9h_8$

C: 1=compressed vector store

Clock Cycles: 10 (one memory access)

Execution Units: All AGU's / Memory

Operation:

```
memory8[Ra+Rb] = Rs
tmp = Ra + Rb – region.start
do
    tmp = tmp >> 8
    memory8[tmp+region.cta] = 0
until (tmp = 0)
```

Vector Operation:

```
y = 0
for x = 0 to VL – 1
    if (Rb is scalar)
        n = Rb * y
    else
        n = Rb[n]
    IF (Vm[n])
        memory8[Ra+n] = Rs[n][7:0]
    IF (C)
        y = y + Vm[n]
    ELSE
        y = y + 1
```

Exceptions: DBE, DBG, LMT, TLB

STT – Store Tetra

Description:

A thirty-two-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 19	18 14	13 9	8	7 0
Displacement$_{28..0}$	Ra$_5$	Rs$_5$	0	92h$_8$

Instruction Format: RILV

47 45	44	43	42 19	18 14	13 9	8	7 0
m$_3$	z	C	Displacement$_{23..0}$	Ra$_5$	Rs$_5$	1	92h$_8$

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory$_{32}$[Ra+displacement] = Rs$_{[31..0]}$

Exceptions: DBE, DBG, TLB, LMT

STTX – Store Tetra Indexed

Description:

A thirty-two-bit value is stored to memory from register Rs. The memory address is the sum of register Ra and register Rb.

Instruction Format:

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	$\sim_2$	Tb	Rb_5	Ra_5	Rs_5	v	$C2h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

$memory_{32}[Ra+Rb*Sc] = Rs_{[31:0]}$

Exceptions: DBE, DBG, LMT, TLB

STW – Store Wyde

Description:

A sixteen-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

Instruction Format:

47 19	18 14	13 9	8	7 0
$\text{Displacement}_{28..0}$	Ra_5	Rs_5	0	91h_8

Instruction Format: RILV

47 45	44	43	42 19	18 14	13 9	8	7 0
m_3	z	C	$\text{Displacement}_{23..0}$	Ra_5	Rs_5	1	91h_8

C: 1=compressed vector load

Clock Cycles: 10 (one memory access)

Execution Units: All ALU's / Memory

Operation:

$\text{memory}_{16}[\text{Ra+displacement}] = \text{Rs}_{[15..0]}$

Exceptions: DBE, DBG, TLB

STWX – Store Wyde Indexed

Description:

A sixteen-bit value is stored to memory from register Rs. The memory address is the sum of register Ra and register Rb.

Instruction Format:

31 29	28	27	2625	24	23 19	18 14	13 9	8	7 0
m_3	z	C	~	Tb	Rb_5	Ra_5	Rs_5	v	$C1h_8$

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

$memory_{16}[Ra+Rb*Sc] = Rs_{[15:0]}$

Exceptions: DBE, DBG, LMT, TLB

Block Instructions

BCMPx – 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 independently specified amounts. This instruction is interruptible. The data must be in the same segment and appropriately aligned.

Instruction Format:

47 45	44	43 41	39 38	37	36 29	28 25	24	23 19	18 14	13	12 9	8	7 0
Sel_3	Ce	Sz_3	m_3	z	$\sim_8$	Bi_4	Tb	Rb_5	Ra_5	~	Ai_4	v	$9Ah_8$

Ai_4/Bi_4	Adjustment Amount
0	0
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
BCMPO [Ra]+,[Rb]+
SUBF LC,LC,200          ; get index of difference
```

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0 and mem[Rb] = mem[Ra]
        Ra = Ra + amt
        Rb = Rb + amt
        LC = LC – 1
```

BFNDx – Block Find

Description:

This instruction compares data from the memory location addressed by Ra to the data in register Rb until the loop counter LC reaches zero or until a match occurs. This instruction is interruptible. The data must be appropriately aligned.

Instruction Format:

47 45	44	43 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
Sel$_3$	~	O$_3$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rc$_6$	v	9Bh$_8$

O$_3$	Assembler Mnemonic	
0	STFND.BI	byte incrementing
1	STFND.WI	wyde incrementing
2	STFND.TI	tetra incrementing
3	STFND.OI	octa incrementing
4	STFND.BD	byte decrementing
5	STFND.WD	wyde decrementing
6	STFND.TD	tetra decrementing
7	STFND.OD	octa decrementing

Execution Units: Memory

Operation:

```
temp = 0
while LC <> 0
        if  (mem[Ra] = Rb)
            stop
        Ra = Ra + amt
        LC = LC – 1
```

BMOVx –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. The data moved must be in the same segment and appropriately aligned. During the move, if the Ce bit is set data will be moved into the cache this may improve performance of the move due to fewer external memory cycles.

This instruction may be used to stream data to or from an I/O port by keeping one of the addresses fixed.

Instruction Format:

47 45	44	43 41	49 38	37	36 33	32 29	28 27	26 21	20 15	1413	12 9	8	7 0
Sel$_3$	Ce	Sz$_3$	m$_3$	z	~$_4$	Bi$_4$	Tb$_2$	Rb$_6$	Ra$_6$	~$_2$	Ai$_4$	v	99h$_8$

Ai$_4$/Bi$_4$	Adjustment Amount
0	0
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
BMOVB [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
```

BSETx – Block Set

Description:

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

Instruction Format:

47 45	44	43	42 31	30 28	27 25	24	23 19	18 14	13	12 9	8	7 0
m_3	z	C	$\sim_{12}$	Sel_3	Sz_3	0	Rb_5	Ra_5	~	Ai_4	0	$98h_8$

Ai_4	Adjustment Amount
0	0
1	1
2	2
3	4
4	8
5	16
15	-1
14	-2
13	-4
12	-8
11	-16
others	reserved

Sz_3	
0	byte
1	wyde
2	tetra
3	octa
4	hexi
5 to 7	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]+
```

Branch / Flow Control Instructions

Overview

Mnemonics

There are two sets of mnemonics for branch instructions. Branch instructions that are IP relative, specify Ca = 7 in the branch instruction and are referred to with a 'B' as in BEQ. Using mnemonics that begin with 'B' imply the code address register is the instruction pointer, C7. Branch instructions that are relative to other code address registers are referred to as jump instructions and begin with a 'J' in the mnemonic.

There are further 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'.

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 three-bit field in the instruction. There are four comparison types that may be performed as outlined in the table below.

Cm_3	Comparison Type
0	signed integer comparisons
1	quad float comparison
2	quad decimal float comparison
3	posit comparison
4	unsigned integer comparisons
5 to 7	reserved

Conditional Branch Format

Branches use 48-bit opcodes.

A 32-bit opcode would not leave a large enough target field 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.

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	Ca$_{3}$	Rb$_{5}$	Ra$_{5}$	Cm$_{3}$	Lk$_{2}$	0	2xh$_{8}$

Branch Conditions

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

				▼			▼
47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	Ca_3	Rb_5	Ra_5	Cm_3	Lk_2	0	$2xh_8$

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		magnitude less than	
2Dh			
2Eh			
2Fh			
26h	equal	equal	equal
27h	not equal	not equal	not equal
24h		ordered	
25h	bit set or clear	unordered	
22h	bit set or clear immediate	bit set or clear immediate	bit set or clear immediate

Linkage

Branches may specify a linkage register which is updated with the address of the next instruction. This allows subroutines to be called. There are two link registers in the architecture.

					▼		
47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	Ca_3	Rb_5	Ra_5	Cm_3	Lk_2	0	$2xh_8$

Lk_2	Meaning
0	do not store return address
1	use Lk1 / Ca1
2	use Lk2 / Ca2
3	reserved

Branch Target

For conditional branches, the target address is formed as the sum of a code address register and a 21-bit constant specified in the instruction. Branches may be IP relative with a range of ±2MB.

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.

▼	▼						
47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	Ca_3	Rb_5	Ra_5	Cm_3	Lk_2	0	$2xh_8$

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. Branching to a value in a register allows all bits of the instruction pointer to be set. Since addresses are formed as the sum of a code address register and a constant in the instruction, branching to a register is inherent in the instruction. The target constant may be set to zero. Specifying Ca = 0 will use the value zero rather than the contents of Ca zero. This allows absolute address branches to be formed.

▼	▼						
47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	Ca_3	Rb_5	Ra_5	Cm_3	Lk_2	0	$2xh_8$

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. Only the first 64-bits of a register may be tested.

Formats Supported: B

47 27	26 24	23 19	18 14	13 12	11	10 9	8	7 0
Target_{21}	7_3	Rb_5	Ra_5	0_2	0	Lk_2	0	25h_8

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 set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 12	11	10 9	8	7 0
Target_{21}	7_3	Imm_5	Ra_5	Imm_2	0	Lk_2	0	22h_8

Operation:

```
Lk = next IP
If (Ra.bit[$\text{Imm}_7$] == 1)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

BBS – Branch if Bit Set

Description:

This instruction branches to the target address if a bit of Ra is set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions. Only the first 64-bits of a register may be tested.

Formats Supported: B

47 27	26 24	23 19	18 14	13 12	11	10 9	8	7 0
Target$_{21}$	7$_3$	Rb$_5$	Ra$_5$	0$_2$	1	Lk$_2$	0	25h$_8$

Operation:

```
Lk = next IP
If (Ra.bit[Rb] == 1)
    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: J

47 27	26 24	23 19	18 14	13 12	11	10 9	8	7 0
Target$_{21}$	7$_3$	Imm$_6$	Ra$_5$	Imm$_2$	1	Lk$_2$	0	22h$_8$

Operation:

```
Lk = next IP
If (Ra.bit[Imm$_7$] == 1)
    IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

BEQ – Branch if Equal

Description:

This instruction branches to the target address if the contents of the Ra equals the contents of Rb, otherwise program execution continues with the next instruction. For a further description see Branch Instructions. Note that for float comparisons positive and negative zero are considered equal.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	7_3	Rb$_5$	Ra$_5$	Cm$_3$	Lk$_2$	0	26h$_8$

Operation:

```
Lk = next IP
If (Ra==Rb)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

For a floating-point comparison positive and negative zero are considered equal.

BEQZ – Branch if Equal to Zero

Description:

This instruction branches to the target address if the contents of the Ra equals zero, otherwise program execution continues with the next instruction. For a further description see Branch Instructions. In many cases it is sufficient to test and branch if a register is zero. Since this is a common case there is a short-hand instruction format available.

Formats Supported: B

31 29	28 19	18 14	13 9	8	7 0
7_3	Tgt$_{10}$	Ra$_5$	Tgt$_5$	0	10h$_8$

Operation:

```
If (Ra==0)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

BGE – Branch if Greater Than or Equal

Description:

This instruction branches to the target address if the contents of the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	7_3	Rb_5	Ra_5	0_3	Lk_2	0	$29h_8$

Operation:

```
Lk = next IP
If (Ra>=Rb)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

BGEU – Branch if Greater Than or Equal Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	7_3	Rb_5	Ra_5	4_3	Lk_2	0	$29h_8$

Operation:

```
Lk = next IP
If (Ra>=Rb)
    IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

BGT – Branch if Greater Than

Description:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	7$_{3}$	Rb$_{5}$	Ra$_{5}$	Cm$_{3}$	Lk$_{2}$	0	2Bh$_{8}$

Operation:

```
Lk = next IP
If (Ra > Rb)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

BGTU – Branch if Greater Than Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target_{21}	7_3	Rb_5	Ra_5	4_3	Lk_2	0	2Bh_8

Operation:

```
Lk = next IP
If (Ra > Rb)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

BLE – Branch if Less Than or Equal

Description:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	7_3	Rb_5	Ra_5	Cm_3	Lk_2	0	$2Ah_8$

Operation:

```
Lk = next IP
If (Ra <= Rb)
    IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

BLEU – Branch if Less Than or Equal Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	7$_3$	Rb$_5$	Ra$_5$	4$_3$	Lk$_2$	0	2Ah$_8$

Operation:

```
Lk = next IP
If (Ra <= Rb)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

BLT – Branch if Less Than

Description:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	7_3	Rb_5	Ra_5	Cm_3	Lk_2	0	28h$_8$

Operation:

```
Lk = next IP
If (Ra < Rb)
    IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

BLTU – Branch if Less Than Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	7_3	Rb_5	Ra_5	4_3	Lk_2	0	$28h_8$

Operation:

```
Lk = next IP
If (Ra < Rb)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

BNE – Branch if Not Equal

Description:

This instruction branches to the target address if the contents of the Ra is not equal to the contents of Rb, otherwise program execution continues with the next instruction. For a further description see Branch Instructions. Float comparisons treat positive and negative zero as equal.

Formats Supported: B

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	7_3	Rb_5	Ra_5	Cm_3	Lk_2	0	$27h_8$

Operation:

```
Lk = next IP
If (Ra != Rb)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

BNEZ – Branch if Not Equal to Zero

Description:

This instruction branches to the target address if the contents of the Ra is not equal to zero, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: B

31 29	28 19	18 14	13 9	8	7 0
7_3	Tgt_{10}	Ra_5	Tgt_5	0	$12h_8$

Operation:

```
If (Ra!=0)
        IP = IP + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

BRA – Branch Always

Description:

This instruction always branches to the target address. The target address range is ±8GB.

Formats Supported: B

31 29	2827 26 21 20 15 14 11	10 9	8	7 0
7_3	Target_{18}	0_2	0	$23h_8$

Formats Supported: B

47 32	31 29	2827 26 21 20 15 14 11	10 9	8	7 0
Target_{16}	7_3	Target_{18}	0	0	$20h_8$

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.

Formats Supported: B

31 29	2827 26 21 20 15 14 11	10 9	8	7 0
7_3	Target_{18}	Lk_2	0	$23h_8$

Formats Supported: B

47 32	31 29	2827 26 21 20 15 14 11	10 9	8	7 0
Target_{16}	7_3	Target_{18}	Lk_2	0	$20h_8$

Operation:

Lk = next IP
IP = IP + Constant

Execution Units: Branch

Exceptions: none

Notes:

DBRA – Decrement and Branch

Description:

This instruction always branches to the target address if the loop count is non-zero. The target address range is ±8GB. The register used as the loop counter is r26.

Formats Supported: B

47 32	31 29	2827 26 21 20 15 14 11	10 9	8	7 0
$Target_{16}$	7_3	$Target_{18}$	0	0	$21h_8$

Operation:

```
If (LC != 0)
        IP = IP + Constant
LC = LC - 1
```

Execution Units: Branch

Exceptions: none

Notes:

DJMP – Decrement and Jump

Description:

This instruction always jumps to the target address. The target address range is ±8GB.

Instruction Format: JMP

47 32	31 29	2827 26 21 20 15 14 11	10 9	8	7 0
$Target_{16}$	Ca_3	$Target_{18}$	0	0	$21h_8$

Operation:

```
If (LC != 0)
        Lk = next IP
        IP = Ca + sign extend (Constant$_{27}$)
LC = LC - 1
```

Execution Units: Branch

Exceptions: none

Notes:

JBC – Jump if Bit Clear

Description:

This instruction branches to the target address if a bit of Ra is clear, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	Ca$_{3}$	Rb$_{5}$	Ra$_{5}$	Cm$_{3}$	Lk$_{2}$	0	24h$_{8}$

Operation:

```
Lk = next IP
If (Ra.bit[Rb] == 0)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

JBCI – Jump 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 set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: J

47 27	2624	23 19	18 14	13 12	11	10 9	8	7 0
Target$_{21}$	Ca$_{3}$	Imm$_{5}$	Ra$_{5}$	Imm$_{2}$	0	Lk$_{2}$	0	22h$_{8}$

Operation:

```
Lk = next IP
If (Ra.bit[Imm$_7$] == 0)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

JBS – Jump if Bit Set

Description:

This instruction branches to the target address if a bit of Ra is set, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	Ca$_3$	Rb$_5$	Ra$_5$	Cm$_3$	Lk$_2$	0	25h$_8$

Operation:

```
Lk = next IP
If (Ra.bit[Rb] == 1)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

JBSI – Jump 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: J

47 27	26 24	23 19	18 14	13 12	11	10 9	8	7 0
Target$_{21}$	Ca$_3$	Imm$_5$	Ra$_5$	Imm$_2$	1	Lk$_2$	0	22h$_8$

Operation:

```
Lk = next IP
If (Ra.bit[$Imm_7$] == 1)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

JEQ – Jump if Equal

Description:

This instruction branches to the target address if the contents of the Ra equals the contents of Rb, otherwise program execution continues with the next instruction. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	Ca$_3$	Rb$_5$	Ra$_5$	Cm$_3$	Lk$_2$	0	26h$_8$

Operation:

```
Lk = next IP
If (Ra==Rb)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

JGE – Jump if Greater Than or Equal

Description:

This instruction branches to the target address if the contents of the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target_{21}	Ca_3	Rb_5	Ra_5	Cm_3	Lk_2	0	29h_8

Operation:

```
Lk = next IP
If (Ra>=Rb)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

JGEU – Jump if Greater Than or Equal Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target_{21}	Ca_3	Rb_5	Ra_5	4_3	Lk_2	0	29h_8

Operation:

```
Lk = next IP
If (Ra >= Rb)
    IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

JGT – Jump if Greater Than

Description:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	Ca$_3$	Rb$_5$	Ra$_5$	Cm$_3$	Lk$_2$	0	2Bh$_8$

Operation:

```
Lk = next IP
If (Ra > Rb)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

JGTU – Jump if Greater Than Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	Ca$_3$	Rb$_5$	Ra$_5$	4$_3$	Lk$_2$	0	2Bh$_8$

Operation:

```
Lk = next IP
If (Ra > Rb)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

JLE – Jump if Less Than or Equal

Description:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	Ca$_3$	Rb$_5$	Ra$_5$	Cm$_3$	Lk$_2$	0	2Ah$_8$

Operation:

```
Lk = next IP
If (Ra <= Rb)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

JLEU – Jump if Less Than or Equal Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	Ca$_3$	Rb$_5$	Ra$_5$	4$_3$	Lk$_2$	0	2Ah$_8$

Operation:

```
Lk = next IP
If (Ra <= Rb)
    IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

JLT – Jump if Less Than

Description:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	Ca_3	Rb_5	Ra_5	Cm_3	Lk_2	0	$28h_8$

Operation:

```
Lk = next IP
If (Ra < Rb)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

JLTU – Jump if Less Than Unsigned

Description:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see Branch Instructions.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
$Target_{21}$	Ca_3	Rb_5	Ra_5	4_3	Lk_2	0	$28h_8$

Operation:

```
Lk = next IP
If (Ra < Rb)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

JNE – Jump if Not Equal

Description:

This instruction branches to the target address if the contents of the Ra is not equal to the contents of Rb, otherwise program execution continues with the next instruction. For a further description see Branch Instructions. For floats, positive and negative zero are considered equal.

Formats Supported: J

47 27	26 24	23 19	18 14	13 11	10 9	8	7 0
Target$_{21}$	Ca$_3$	Rb$_5$	Ra$_5$	Cm$_3$	Lk$_2$	0	27h$_8$

Operation:

```
Lk = next IP
If (Ra != Rb)
        IP = Ca + Constant
```

Execution Units: Branch

Exceptions: none

Notes:

JMP – Jump

Description:

This instruction always jumps to the target address. The target address range is ±8GB.

Instruction Format: JMP

47 32	31 29	2827 26 21 20 15 14 11	10 9	8	7 0
Target_{16}	Ca_3	Target_{18}	0	0	20h_8

Operation:

Lk = next IP
IP = Ca + sign extend (Constant_{27})

Execution Units: Branch

Exceptions: none

Notes:

JSR – Jump 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 ±8GB.

Formats Supported: JMP

47 32	31 29	2827 26 21 20 15 14 11	10 9	8	7 0
$Target_{16}$	Ca_3	$Target_{18}$	Lk_2	0	$20h_8$

Operation:

Lk = next IP
IP = Ca + Constant

Execution Units: Branch

Exceptions: none

Notes:

NOP – No Operation

Description:

This instruction does not do anything.

Integer Instruction Format:

15 9	8	7 0
$\sim_7$	v	$F1h_8$

Operation: none

Vector Operation

Execution Units: I

Clock Cycles: 1

Exceptions: none

Notes:

RTS – Return from Subroutine

Description:

This instruction returns from a subroutine by transferring program execution to the address calculated as the sum of a link register and a constant. The const field is shifted left once before use.

Formats Supported: RTS

15 11	10 9	8	7 0
$Const_5$	Lk_2	0	$F2h_8$

Flags Affected: none

Operation:

Execution Units: Branch

Exceptions: none

Notes:

A branch instruction may also be used to return from a subroutine

Return address prediction hardware may make use of the RTS instruction.

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 register, C6.

Instruction Format: BRK

15 9	8	7 0
$\sim_7$	0	$00h_8$

Operation:

PMSTACK = (PMSTACK << 8) | 3Eh
PLSTACK = (PLSTACK << 8) | FFh
CAUSE = FLT_BRK
C[8+stk_depth] = IP
IP = tvec[3]

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

CSRx – Control and Special / Status Access

Description:

The CSR instruction group provides access to control and special or status registers in the core. For the read operation the current value of the CSR is placed in the target register Rt.

Instruction Format: CSR

47 45	44	43 41	40 35	34 19	18 14	13 9	8	7 0
m_3	z	O_3	$\sim_6$	$Immediate_{15..0}$	Ra_5	Rt_5	v	$0Fh_8$

O_3		Operation
0	CSRRD	Only read the CSR, no update takes place, Ra should be x0.
1	CSRRW	Read/Write to CSR
2	CSRRS	Read/Set CSR bits
3	CSRRC	Read/Clear CSR bits
4 to 7		Reserved

CSRRS and CSRRC operations are only valid on registers that support the capability.

The $Regno_{[15..12]}$ field is reserved to specify the operating mode. Note that registers cannot be accessed by a lower operating mode.

Execution Units: Integer, the instruction may be available on only a single execution unit (not supported on all available integer units).

Clock Cycles: 1

Exceptions: privilege violation attempting to access registers outside of those allowed for the operating mode.

DI – Disable Interrupts

Description:

This instruction disables asynchronous interrupts for a short period of time. The contents of Ra or'd with the N field specifies the number of following instructions for which interrupts are disabled. Interrupts may be disabled for the execution of a maximum of thirty-one instructions. Note that macro instructions count as the number of instructions making up the macro instruction.

Instruction Format: DI

31 25	24 22	21	20 19	18 14	13 9	8	7 0
41h$_7$	~	~	~$_2$	Ra$_5$	N$_5$	v	01h$_8$

Operation:

Execution Units: ALU

Clock Cycles:

Exceptions: none

Notes:

INT – 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:

31 29	28 25	24 9	8	7 0
Lvl_3	$\sim_4$	Cause_{16}	0	A6h_8

Operation:

PMSTACK = (PMSTACK << 8) | 30h + (Lvl_3 * 2)
PLSTACK = (PLSTACK << 8) | FFh
CAUSE = Cause_{16}
C[8+stk_depth] = IP
IP = tvec[3]

MEMDB – Memory Data Barrier

Description:

All memory accesses before the MEMDB command are completed before any memory accesses after the data barrier are started.

Instruction Format:

15 9	8	7 0
$\sim_7$	0	$F9h_8$

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.

Instruction Format:

15 9	8	7 0
$\sim_7$	0	F8h$_8$

Clock Cycles: 1

Execution Units: Memory

MFLK – Move from Link Register

Description:

This instruction moves a link register to a general-purpose register.

Instruction Format: MFLK

1514	13 9	8	7 0
L_2	Rt_5	v	$5Eh_8$

MTLK – Move to Link Register

Description:

This instruction moves a general-purpose register to a link register.

Instruction Format: MFLK

1514	13 9	8	7 0
L_2	Rs_5	v	$5Fh_8$

PEEKQ – Peek at Queue / Stack

Description:

This instruction returns the top value into Rt from the hardware queue specified in Rb. The hardware queue position is not advanced. Unused value bits should read as zero. Used the STATQ instruction to get the queue status.

Instruction Format: OSR2

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
0Ah$_7$	m$_3$	z	~$_2$	~$_6$	Tb$_2$	Rb$_6$	~$_6$	Rt$_6$	v	07h$_8$

Exceptions: none

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 eight bits, and the interrupt level in bits eight to eleven of the register.

Instruction Format: OSR2

15 14	13 9	8	7 0
$\sim_2$	Rt_5	v	FBh_8

Clock Cycles: 1 (if no exception present)

Operation:

```
if (irq <> 0)
        Rt[7:0] = cause code
        Rt[11:8] = irq level
        PMSTACK = (PMSTACK << 4) | 6
        CAUSE = Const₈
        C6 = IP
        IP = tvec[3]
```

Execution Units: Branch

POPQ – Pop from Queue / Stack

Description:

This instruction pops a value into Rt from the hardware queue specified in Rb. The hardware queue position is advanced. Unused value bits should read as zero. To check the queue status, use the STATQ instruction.

63 0
Value

Value: the value that was pushed to the queue

Instruction Format: OSR2

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$09h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	$\sim_6$	Rt_6	v	$07h_8$

Exceptions: none

Notes:

Queue #15 is the instruction trace que

PUSHQ – Push on Queue / Stack

Description:

This instruction pushes an N-bit value in Ra onto the hardware queue specified in Rb. Where N is implementation defined between 1 and 64 bits. To check the queue status, use the STATQ instruction.

Instruction Format: OSR2

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$08h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	$\sim_6$	v	$07h_8$

Exceptions: none

RESETQ – Reset Queue / Stack

Description:

This instruction resets the hardware queue specified by Rb.

Instruction Format: OSR2

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
0Ch$_7$	m$_3$	z	~$_2$	~$_6$	Tb$_2$	Rb$_6$	~$_6$	~$_6$	v	07h$_8$

Exceptions: none

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: REX

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 11	10 9	8	7 0
$10h_7$	m_3	z	$\sim_2$	$\sim_6$	$\sim_2$	$\sim_6$	Ra_6	$\sim_4$	Tm_2	v	$07h_8$

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.

RTI – Return from Interrupt or Exception

Description:

Restore the previous interrupt enable setting, operating mode, and privilege level and transfer program execution back to the address in the exception address register (one of C8 to C15). Semaphore register zero is always cleared by this instruction.

This instruction may be encoded to return a short distance past the exception address point. This may be useful to return to the next instruction or return to a point past inline parameters. The constant$_5$ field specifies a return offset in terms of bytes.

There is only a single instruction to return from any mode for an exception. Although there are several additional mnemonics.

Instruction Format: RTI

15 11	10 9	8	7 0
Const$_5$	0	v	F0h$_8$

Flags Affected: none

Operation:

```
PMSTACK = PMSTACK >> 8
PLSTACK = PLSTACK >> 8
Semaphore[0] = 0
IP = C[7+stk_depth] + Constant
```

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

SEI – Set Interrupt Level

Description:

The interrupt mask is set, disabling lower level maskable interrupts. The current interrupt mask level is stored in the target register Rt. The new interrupt mask level is set to the contents of Ra bitwise or'd with a three-bit level field in the instruction. This instruction is available only in machine mode.

Instruction Format:

31 25	24 22	21	2019	18 14	13 9	8	7 0
$40h_7$	Im_3	0	0_2	Ra_5	Rt_5	v	$01h_8$

Clock Cycles: 1

Operation:

im = $Rb_{[2:0]}$

Exceptions: none

STATQ – Get Status of Queue / Stack

Description:

This instruction returns a queue status value into Rt from the hardware queue specified in Rb. The hardware queue position is not advanced. Unused value bits should read as zero.

63	62	61 48	47 0
Qe	Dv	Data Count	Extended Data (XD)

Fields

Qe: empty.If set, this bit indicates that the queue/stack is empty.

Dv: data valid. If this bit is set it indicates that valid data is present at the queue.

Dc: data count: The number of items left in the queue

XD: The high order 48 bits of the data stored by the queue if the queue is wider than 64 bits.

Instruction Format: OSR2

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
0Bh$_7$	m$_3$	z	~$_2$	~$_6$	Tb$_2$	Rb$_6$	~$_6$	Rt$_6$	v	07h$_8$

Exceptions: none

SYNC -Synchronize

Description:

All instructions for a particular unit before the SYNC are completed and committed to the architectural state before instructions of the unit type after the SYNC are issued. This instruction is used to ensure that the machine state is valid before subsequent instructions are executed.

Instruction Format:

15 9	8	7 0
$\sim_7$	0	$F7h_8$

SYS – Call system routine

Description:

This instruction invokes the system exception handler. The return address is stored in the EIP register (code address register #8 to 15). This instruction causes the core to switch to machine mode.

Instruction Format:

31 25	24 9	8	7 0
$\sim_7$	$Cause_{16}$	0	$A5h_8$

Operation:

CAUSE = $Cause_{16}$
C[8+stk_depth] = IP
IP = tvec[3]

TLBRW – Read / Write TLB

Description:

This instruction both reads and writes the TLB. Which translation entry to update and the value to use comes from the value in register pair Ra, Rb which contains the virtual page number, ASID, and physical page number. The current value of the entry selected by Ra is copied to register pair Rt. The TLB will be written only if bit 31 of Ra is set. A random way may be selected for write by setting the ‘r’ bit in the Ra value.

The entry number for Ra comes from virtual address bits 12 to 21.

Page numbers are in terms of a 4kB page size.

Instruction Format: OSR3

47 41	49 38	37	36 34	33	32 31	30 26	25 24	23 19	18 14	13 10	9	8	7 0
$1Eh_7$	m_3	z	$\sim_3$	~	$\sim_2$	$\sim_5$	$\sim_2$	Rb_5	Ra_5	Rt_4	0	v	$07h_8$

Clock Cycles: 5

Execution Units: Memory

Ra Value Format

31	20	19	18	17	16	15	14	13	12	11	10	8	7	6	5	4	3	2	1	0
WR	$\sim_{14}$				~	AL_2		$\sim_2$		way		Entry No_{10}								
$ASID_{12}$		SC	SR	SW	SX	V	G	Av	▨ ~	BC_4			D	U	S	A	C	R	W	X
Key_{32}																				
Access $Count_{32}$																				

AL_2: update algorithm 0 = fixed, 1 = LRU, 2 = random

Rb Value Format

Physical Page $Number_{31..0}$				
EN_3	N	PL_8	▨ $\sim_4$	Physical Page $Number_{47..32}$
Virtual Page $Number_{31..0}$				
MB_6		ME_6	▨ $\sim_4$	Virtual Page $Number_{47..32}$

	Meaning	
PPN	Physical page number (address bits 12 to 63)	
VPN	Virtual page number high address order bits 12 to 63	
X	1 = page is executable	These three combined indicate page present (P) 0 = not present
W	1 = page is writeable	
R	1 = page is readable	
C	1 = page is cachable	
A	Accessed, set if translation was used	
S	1 = 4MB page	
U	reserved for system usage	
D	Dirty, set if a write occurred to the page	
BC	bounce count	
~	reserved	
V	1 = translation valid (page is present)	
G	Global, global translation indicator	
SX	1 = page is executable by system	
SW	1 = page is writeable by system	
SR	1 = page is readable by system	

SC	1 = page is cacheable by system
ASID	ASID address space identifier

Exceptions: none

WAIT – Wait for Signal

Description:

The WAIT instruction waits for a signal to occur before proceeding.

Instruction Format: WAIT

15 11	10	9	8	7 0
~$_5$	D	0	0	FAh$_8$

Clock Cycles: 1 (if no exception present)

Execution Units: Branch

WFI – Wait for Interrupt

Description:

The WFI instruction waits for an external interrupt to occur before proceeding. While waiting for the interrupt, the processor clock is stopped placing the processor in a lower power mode.

Instruction Format: WAIT

15 10	9	8	7 0
0$_6$	1	0	FAh$_8$

Clock Cycles: 1 (if no exception present)

Execution Units: Branch

Macro Instructions

DEFCAT – Default Catch Handler

Description:

DEFCAT reaches into the stack frame of the calling function to get the last registered catch handler address. It then moves this value to the return address fields of the return block so that the current function will return to the catch handler in the caller. The values in registers t0 and t1 are overwritten.

Instruction Format: DEFCAT

15 9	8	7 0
$\sim_7$	0	$F0h_8$

Operation:

```
T0 = Memory[FP]
T1 = Memory32[T0]
Memory16[FP] = T1
T1 = Memory40[T0]
Memory32[FP] = T1
```

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 8MB may be allocated on the stack. An immediate prefix 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

31 9	8	7 0
Constant_{23}	0	AFh_8

Operation:

```
SP = SP - 64
Memory[SP] = FP
T0 = CA1
Memory16[SP] = T0
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 three-bit constant multiplied by two is added to the link register to form the return address. This allows returning up to 16 bytes past the normal return address.

Instruction Format: LEAVE

31 13	12 9	8	7 0
Constant$_{19}$	Cnst$_4$	0	BFh$_8$

Operation:

SP = FP
FP = Memory[SP]
T0 = Memory16[SP]
CA1 = T0
SP = SP + 64 + Constant$_{20}$ * 16
IP = CA1 + Cnst$_3$ * 2

POP – Pop Register from Stack

Description:

This instruction pops a register from the stack.

Instruction Format: POP

15 14	13 9	8	7 0
0	Rt$_5$	0	BCh$_8$

Operation:

Ra = Memory$_{128}$[SP]
SP = SP + 16

POP – Pop Registers from Stack

Description:

This instruction pops up to four registers from the stack. The number of registers to pop is specified in the N field of the instruction.

Instruction Format: POP3R

31 29	28 24	23 19	18 14	13 9	8	7 0
N_3	Rd_2	Rc_5	Rb_5	Ra_5	0	BEh_8

Operation:

```
if (N > 0) Ra = Memory[SP]
if (N > 1) Rb = Memory16[SP]
if (N > 2) Rc = Memory32[SP]
if (N > 3) Rd = Memory48[SP]
SP = SP + N*16
```

PUSH – Push Register on Stack

Description:

This instruction pushes a register onto a stack.

Instruction Format: PUSH

1514	13 9	8	7 0
0	Ra_5	0	ACh_8

Operation:

SP = SP – 8
Memory[SP] = Ra

PUSH – Push Registers on Stack

Description:

This instruction pushes up to four registers onto a stack. The number of registers to push is specified in the N field of the instruction.

Instruction Format: PUSH

31 29	28 24	23 19	18 14	13 9	8	7 0
N_3	Rd_5	Rc_5	Rb_5	Ra_5	0	AEh_8

Operation:

```
$SP = SP – N * 16$
if $(N > 3)$ $Memory_{16}[SP+(N-4)*16] = Rd$
if $(N > 2)$ $Memory_{16}[SP+(N-3)*16] = Rc$
if $(N > 1)$ $Memory_{16}[SP+(N-2)*16] = Rb$
if $(N > 0)$ $Memory_{16}[SP+(N-1)*16] = Ra$
```

Vector Specific Instructions

V2BITS

Description

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

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
3Dh$_7$	m$_3$	z	~$_2$	~$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

Operation

```
For x = 0 to VL-1
    if (Vm[x])
        Rt.bit[x] = Ra[x].bit[Rb]
    else if (z)
        Rt.bit[x] = 0
    else
        Rt.bit[x] = Rt.bit[x]
```

Exceptions: none

VBITS2V

Description

This is an alternate mnemonic for the CMOVNZ instruction where Rb = 1 and Rc = 0. The effect is to generate a Boolean vector from the contents of register Ra.

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$2Dh_7$	m_3	z	2_2	0_6	2_2	1_6	Ra_6	Rt_6	v	$02h_8$

Operation

```
For x = 0 to VL-1
    if (Vm[x]) Vt[x] = Ra.bit[x] ? 1 : 0
    else if (z) Vt[x] = 0
    else Vt[x] = Vt[x]
```

Exceptions: none

VGIDX – 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: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
3Ch$_7$	m$_3$	z	~$_2$	~$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

Operation

```
y = 0
for x = 0 to VL - 1
    if (Vm[x])
        Vt[y] = Ra * y + Rb
        y = y + 1
```

VCMPRSS – Compress Vector

Description

Selected elements from vector register Va are copied to elements of vector register Vt guided by a vector mask register.

Instruction Format: R1

31 25	24 22	21	20 19	18 14	13 9	8	7 0
$2Ch_7$	m_3	z	$\sim_2$	Ra_5	Rt_5	1	$01h_8$

Operation

```
y = 0
for x = 0 to VL - 1
    if (Vm[x])
        Vt[y] = Va[x]
        y = y + 1
```

VEINS / VMOVSV – Vector Element Insert

Synopsis

Vector element insert.

Description

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

Instruction Format: R2

47 41	49 38	37	36 35	34 25	24	23 19	18 14	13 9	8	7 0
$3Bh_7$	m_3	z	$\sim_2$	$\sim_{10}$	Tb	Rb_5	Ra_5	Rt_5	v	$02h_8$

Operation

Vt[Ra] = Rb

Exceptions: none

VEX / VMOVS – Vector Element Extract

Synopsis

Vector element extract.

Description

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

Instruction Format: R2

47 41	49 38	37	36 35	34 25	24	23 19	18 14	13 9	8	7 0
3Ah$_7$	m$_3$	z	~$_2$	~$_{10}$	Tb	Rb$_5$	Ra$_5$	Rt$_5$	v	02h$_8$

Operation

Rt = Vb[Ra]

Exceptions: none

MFVM – Move from Vector Mask

Description

Move a mask register to a general-purpose register.

Instruction Format: VMR2

31 27	26 18	17 15	14 9	8	7 0
$11h_5$	$\sim_9$	Vmb_3	Rt_6	0	$52h_8$

Operation

Rt = Vmb

Execution Units: ALUs

MFVL – Move from Vector Length

Description

Move vector length register to a general-purpose register.

Instruction Format: R1

31 27	26 18	17 15	14 9	8	7 0
$13h_5$	$\sim_9$	$\sim_3$	Rt_6	0	$52h_8$

Operation

Rt = VL

Execution Units: ALUs

MTVM – Move to Vector Mask

Description

Move a general-purpose register to a mask register.

Instruction Format: VMR2

31 27	26 21	20 15	14 12	11 9	8	7 0
$10h_5$	$\sim_6$	Ra_6	$\sim_3$	Vmt_3	0	$52h_8$

Operation

Vmt = Ra

Execution Units: ALUs

MTVL – Move to Vector Length

Description

Move a general-purpose register to the vector length register.

Instruction Format: VMR2

31 27	26 21	20 15	14 12	11 9	8	7 0
$12h_5$	$\sim_6$	Ra_6	$\sim_3$	$\sim_3$	0	$52h_8$

Operation

VL = Ra

Execution Units: ALUs

VMADD – Vector Mask Add

Description:

Add the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2

31 27	26 18	17 15	14 12	11 9	8	7 0
$04h_5$	$\sim_9$	Vmb_3	Vma_3	Vmt_3	0	$52h_8$

1 clock cycle

Exceptions: none

VMAND – Vector Mask And

Description:

Bitwise ‘and’ the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2

31 27	26 18	17 15	14 12	11 9	8	7 0
$08h_5$	$\sim_9$	Vmb_3	Vma_3	Vmt_3	0	$52h_8$

1 clock cycle

Exceptions: none

VMCNTPOP – Count Population

CNTPOP r1,vm2

Description:

Count the number of ones and place the count in the target register.

Instruction Format: VMR2

31 27	26 18	17 15	14 9	8	7 0
$0Dh_5$	$\sim_9$	Vmb_3	Rt_6	0	$52h_8$

Execution Units: integer ALU

Exceptions: none

VMFILL – Vector Mask Fill

Description:

Fill the contents of a vector mask register with a mask of ones beginning at Mb_6 and ending at Me_6 inclusive. Fill the remainder of the register with zeros.

Instruction Format:

31 24	23 18	17 12	11 9	8	7 0
$\sim_8$	Me_6	Mb_6	Vmt_3	0	$53h_8$

1 clock cycle

Exceptions: none

VMFIRST – Find First Set Bit

Description

The position of the first bit set in the mask register is copied to the target register. If no bits are set the value is 65536. The search begins at the least significant bit and proceeds to the most significant bit.

Instruction Format: R1

31 27	26 18	17 15	14 9	8	7 0
$0Eh_5$	$\sim_9$	Vmb_3	Rt_6	0	$52h_8$

Operation

Rt = first set bit number of (Vm)

Exceptions: none

Execution Units: ALUs

VMLAST – Find Last Set Bit

Description

The position of the last bit set in the mask register is copied to the target register. If no bits are set the value is 65536. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

Instruction Format: VMR2

31 27	26 18	17 15	14 9	8	7 0
$0Fh_5$	$\sim_9$	Vmb_3	Rt_6	0	$52h_8$

Operation

Rt = last set bit number of (Vm)

Exceptions: none

Execution Units: ALUs

VMOR – Vector Mask Or

Description:

Bitwise ‘or’ the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2

31 27	26 18	17 15	14 12	11 9	8	7 0
09h$_5$	~$_9$	Vmb$_3$	Vma$_3$	Vmt$_3$	0	52h$_8$

1 clock cycle

Exceptions: none

VMSLL – Vector Mask Shift Left Logical

Description:

Shift a vector mask register to the left up to 63 bits.

Instruction Format: VMR2

31 27	26 21	20 15	14 12	11 9	8	7 0
1Ch$_5$	~$_6$	Amount$_6$	Vma$_3$	Vmt$_3$	0	52h$_8$

1 clock cycle

Exceptions: none

VMSRL – Vector Mask Shift Right Logical

Description:

Shift a vector mask register to the right up to 63 bits.

Instruction Format: VMR2

31 27	26 21	20 15	14 12	11 9	8	7 0
1Eh$_5$	~$_6$	Amount$_6$	Vma$_3$	Vmt$_3$	0	52h$_8$

1 clock cycle

Exceptions: none

VMSUB – Vector Mask Subtract

Description:

Subtract the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2

31 27	26 18	17 15	14 12	11 9	8	7 0
$05h_5$	$\sim_9$	Vmb_3	Vma_3	Vmt_3	0	$52h_8$

1 clock cycle

Exceptions: none

VMXOR – Vector Mask Exclusive Or

Description:

Bitwise ‘or’ the contents of two vector mask registers and place the result in a vector mask register.

Instruction Format: VMR2

31 27	26 18	17 15	14 12	11 9	8	7 0
$0Ah_5$	$\sim_9$	Vmb_3	Vma_3	Vmt_3	0	$52h_8$

1 clock cycle

Exceptions: none

VSCAN

Description

Elements of Vt are set to the cumulative sum of a value in register Ra. The summation is guided by a vector mask register.

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
1Eh$_7$	m$_3$	z	Ra$_6$	Vt$_6$	1	01h$_8$

Operation

```
sum = 0
for x = 0 to VL - 1
    Vt[x] = sum
    if (Vm[x])
        sum = sum + Ra
```

VSLLV – Shift Vector Left Logical

Description

Elements of the vector are transferred upwards to the next element position. The first is loaded with the value zero. This is also called a slide operation.

Instruction Format: R3

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

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

VSRLV – Shift Vector Right Logical

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.

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$39h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

Cryptographic Accelerator Instructions

AES64DS – Final Round Decryption

Description:

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

Integer Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
50h$_7$	m$_3$	z	~$_2$	~$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

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

Operation:

Rt = Ra & Rb

Exceptions: none

AES64DSM – Middle Round Decryption

Description:

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

Integer Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
51h$_7$	m$_3$	z	~$_2$	~$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

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

Operation:

Rt = Ra & Rb

Exceptions: none

AES64ES – Final Round Encryption

Description:

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

Integer Instruction Format: R3

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

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

Operation:

Rt = Ra & Rb

Exceptions: none

AES64ESM – Middle Round Encryption

Description:

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

Integer Instruction Format: R3

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

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

Operation:

Rt = Ra & Rb

Exceptions: none

SHA256SIG0

Description:

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

Instruction Format: R1

31 27	24 22	21	20 15	14 9	8	7 0
30h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	01h$_8$

Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

Exceptions: none

SHA256SIG1

Description:

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

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
31h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	01h$_8$

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

31 25	24 22	21	20 15	14 9	8	7 0
$32h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

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

31 25	24 22	21	20 15	14 9	8	7 0
$33h_7$	m_3	z	Ra_6	Rt_6	v	$01h_8$

Operation:

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

Execution Units: ALU #0

Exceptions: none

SHA512SIG0

Description:

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

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
34h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	01h$_8$

Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

Exceptions: none

SHA512SIG1

Description:

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

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
35h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	01h$_8$

Clock Cycles: 1

Operation:

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

Execution Units: ALU #0

Exceptions: none

SHA512SUM0

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
36h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	01h$_8$

SHA512SUM1

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
37h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	01h$_8$

SM3P0

Description:

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
38h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	01h$_8$

SM3P1

Description:

Instruction Format: R1

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

SM4ED

Description:

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
56h$_7$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

SM4KS

Description:

Instruction Format: R3

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
57h$_7$	m$_3$	z	Tc$_2$	Rc$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

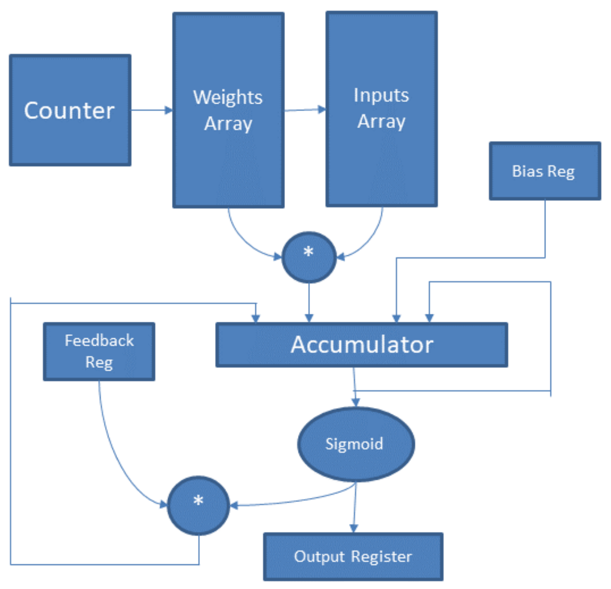
Neural Network Accelerator Instructions

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. 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.

Neural Network Accelerator – One Neuron



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

31 25	24 22	21	20 15	14 9	8	7 0
62h$_7$	m$_3$	z	Ra$_6$	Rt$_6$	v	01h$_8$

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

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
65h$_7$	m$_3$	z	~$_2$	~$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

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

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$62h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	$\sim_6$	v	$02h_8$

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

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$63h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	$\sim_6$	v	$02h_8$

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

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
61h$_7$	m$_3$	z	~$_2$	~$_6$	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

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

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$64h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	$\sim_6$	v	$02h_8$

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

47 41	49 38	37	36 35	34 29	28 27	26 21	20 15	14 9	8	7 0
$60h_7$	m_3	z	$\sim_2$	$\sim_6$	Tb_2	Rb_6	Ra_6	Rt_6	v	$02h_8$

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

31 25	24 22	21	20 15	14 9	8	7 0
61h$_7$	m$_3$	z	~$_6$	Rt$_6$	v	01h$_8$

Clock Cycles: 1

Execution Units: NNA

Notes:

NNA_TRIG – Trigger Calc

Description:

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

Instruction Format: R1

31 25	24 22	21	20 15	14 9	8	7 0
60h$_7$	m$_3$	z	Ra$_6$	~$_6$	v	01h$_8$

Clock Cycles: 1

Execution Units: NNA

Notes:

Graphics

Overview

Thor has an integrated graphics engine which performs graphics operations such as line or triangle drawing. The graphics engine runs asynchronously to other operations. The interface to the graphics engine is through hardware queue #0. The queue may be updated with the PUSHQ instruction. The queue acts as a worklist for the graphics engine. Software pushes graphics commands onto the queue and the graphics engine pops the commands during its processing.

Value for Ra

63 56	55 40	39 0
Graphics Register Number	~	Update Value

Co-ordinates

Co-ordinates are specified as 16.16 fixed point numbers.

31 16	15 0
coord.whole	coord.fract

Colors

Colors are represented using ZRGB8888 format. Colors are placed in the low order 32-bits of a register.

31 24	23 16	15 8	7 0
Z-order	Blue	Green	Red

Register Set

Graphics registers are 40 bits wide. Some registers are infrequently updated such as the target base address or font table address registers.

Reg no		Comment
0	Control Register	
1		
2	Alpha	
3	Transparent Color	
4	Target Base address (selector)	
5	Target Size X	See Target Coordinate registers
6	Target Size Y	

7	Texture Base Address (selector)	
8	Texture Size X	
9	Texture Size Y	
10	Source Pixel #0 X	
11	Source Pixel #0 Y	
12	Source Pixel #1 X	
13	Source Pixel #1 Y	
14	Destination Pixel X	
15	Destination Pixel Y	
16	Destination Pixel Z	
17	Transform Coefficient AA	
18	Transform Coefficient AB	
19	Transform Coefficient AC	
20	Transform Coefficient AT	
21	Transform Coefficient BA	
22	Transform Coefficient BB	
23	Transform Coefficient BC	
24	Transform Coefficient BT	
25	Transform Coefficient CA	
26	Transform Coefficient CB	
27	Transform Coefficient CC	
28	Transform Coefficient CT	
29	Clip Pixel #0 X	Defines a clipping region
30	Clip Pixel #0 Y	
31	Clip Pixel #1 X	
32	Clip Pixel #1 Y	
33	Color #0	Only color #0 is used unless interpolating colors
34	Color #1	
35	Color #2	
36	U0	
37	V0	
38	U1	
39	V1	
40	U2	
41	V2	
42	Z-Buffer Base Address (selector)	
43		
44	Target X0	Defines the drawing area
45	Target Y0	
46	Target X1	
47	Target Y1	
48	Font Table Base Address (selector)	
49	Font ID	
50	Character Code	
51 to 63	reserved	

CTRL – Reg #0

Bits moved and added for color depth.

Bit #	Access	Description
[39:32]	~	reserved
[31:30]	W	Bitset operation
[29:24]	~	Reserved
[23:20]	RW	Color Depth
[19]	W	Transform point
[18]	W	Forward point
[17:16]	RW	Active point
[15:14]	~	Reserved
[13]	W	Bézier inside shape
[12]	W	Interpolation
[11]	W	Curve write
[10]	W	Triangle write
[9]	W	Line write
[8]	W	Rect write
[7]	W	Point write
[6]	RW	Z-buffer enable
[5]	RW	Clipping enable
[4]	RW	Transparent Color enable
[3]	RW	Blending enable
[2]	RW	Texture0 enable
[1]	~	Reserved
[0]	W	Char write

Bits 0, 8, 9, 10, 18, and 19 automatically reset to zero.

Opcode	Operation
00	copy bits 0 to 29 to register
01	no operation, register unaffected
10	clear specified bits of register
11	set specified bits of register

Color depth is the number of bits per color component – 1.

Mode	Color Depth	Z,R,G,B	Pixels Per Strip
0000	4 bit	1,1,1,1	32
0001	8 bit	2,2,2,2	16
0010	12 bit	3,3,3,3	10
0011	16 bit	4,4,4,4	8
0100	20 bit	5,5,5,5	6
0101	24 bit	6,6,6,6	5
0110	28 bit	7,7,7,7	4
0111	32 bit	8,8,8,8	4
1000	36 bit	9,9,9,9	3
1001	40 bit	10,10,10,10	3

Alpha – reg #2

This register contains alpha values for global and point 0,1,2 alphas. 0xFF is full opacity, 0x00 is full transparency. The point alpha values are used to compute an interpolated alpha value from each point of a triangle. The interpolated alpha value is then multiplied by the global alpha value.

Bits	
0 to 7	Global Alpha
8 to 15	Point #2 alpha
16 to 23	Point #1 alpha
24 to 31	Point #0 alpha

Transparent Color – reg #3

Pixels colored matching the transparent color will not be drawn if the transparent color is enabled by bot4 of the control register.

Target Base Address – reg #4

This register contains the 32-bit selector identifying where in memory the graphics drawing target is located.

Texture Base Address – reg #7

This register contains the 32-bit selector identifying where in memory a texture is located.

Texture Size X, Y – reg #8, 9

This pair of registers defines the size of the texture area.

Source Pixel 0, 1 reg #10 to 13

These four registers define an area of the texture to draw from.

Destination Pixel X, Y, Z reg #14 to 16

This trio of registers contain the coordinates of the destination point. These registers are used in conjunction with the active point field of the control register. Three destination points are required to draw a triangle. Points are pushed into the graphics pipeline by setting the forward or transform bits of the control register.

Transform Coefficient reg #17 to 28

These registers are used to set the coefficients for a transformation matrix used to translate or rotate points.

Clip Region reg #29 to 32

These registers define a clipping region outside of which points will be discarded.

Color 0,1,2 reg #33 to 35

The number of bits used in the register depends on the color depth.

U0 to 2, V0 to 2 reg #36 to 41

These are texture coordinates.

Z-Buffer Base reg #42

This register contains a selector for the base address of the z-buffer.

Target Area reg #44 to 47

This set of registers defines the target area.

Font Table Base Address reg #48

This register contains the selector for the base address of the font table.

Font ID reg #49

Contains an identifier for the currently selected font.

Character Code reg #50

Contains the character code to draw.

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 an eight-bit value assumed to be a binary fraction less than one. The color values in Ra and Rb are assumed to be RGB888 format colors. The result is a RGB888 format color. The high order eight bits of the result register are set to the high order eight bits of Ra. Note that a close approximation to 1.0 – alpha is used. Each component of the color is blended independently.

Instruction Format: R3

40 38	37	3635	34 29	28 27	26 21	20 15	14 11	8	7 0
m_3	z	Tc_2	Rc_6	Tb_2	Rb_6	Ra_6	Rt_6	v	$44h_8$

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

CLIP – Clip Point

Description:

The clip instruction checks that the point in Ra is within the graphics target area always and clip region if enabled. The target and clip areas must have been previously set. If the point should be clipped a one is set in Rt, otherwise Rt is set to zero.

Points are represented in 16.16 fixed-point format for each coordinate.

Instruction Format: R1

40 34	33 31	30	29 21	20 15	14 9	8	7 0
20h$_7$	m$_3$	z	~$_9$	Ra$_6$	Rt$_6$	v	01h$_8$

Clock Cycles: 2

GFX_PUSH

PLOT – Plot Point

Description:

This instruction plots a point in the graphics target area. The point’s co-ordinates are in Ra, the color to use is in Rb.

Instruction Format: R2

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
70h$_7$	m$_3$	z	~	Tb$_2$	Rb$_6$	Ra$_6$	Rt$_6$	v	02h$_8$

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 to used in the calculation.

Points are represented in 16.16 fixed-point format.

Instruction Format: R1

40 34	33 31	30	29	28 27	26 21	20 15	14 9	8	7 0
$11h_7$	m_3	z	~	$\sim_2$	$\sim_6$	Ra_6	Rt_6	v	$01h_8$

Clock Cycles: 2

RW_COEFF – Read/Write Co-efficient

Description:

RW_COEFF reads and writes a coefficient value to be used for the transform matrix. Ra contains the number of the coefficient to read or write. Rb contains the new value for the coefficient. Coefficients are in 16.16 fixed point format.

Instruction Format: R2

35 29	2827	2625	24 20	19	18 14	13	12 8	7	6 0
77h$_7$	~$_2$	Tb$_2$	Rb$_5$	Ta	Ra$_5$	Tt	Rt$_5$	v	02h$_7$

Co-efficient Matrix:

AA	AB	AC	AT
BA	BB	BC	BT
CA	CB	CC	CT

Regno in Ra	Coefficient Accessed
0	AA
1	AB
2	AC
3	AT
4	BA
5	BB
6	BC
7	BT
8	CA
9	CB
10	CC
11	CT
12	CMD – bit 0, 1=transform, 0 = pass through

Opcode Maps

Thor2021 Root Opcode

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	{R1}	{R2}	{R3}	ADDI	SUBFI	MULI	{SYS}	ANDI	ORI	EORI				MULUI	{CSR}
1x	BEQZ	REP	BNEZ		JGATE	MULFI	SEQI	SNEI	SLTI	ADD	AND	SGTI	SLTUI	OR	EOR	SGTUI
2x	BRA	DBRA			BBC	BBS	BEQ	BNE	BLT	BGE	BLE	BGT	BLTU	BGEU	BLEU	BGTU
3x																
4x	DIVI	CPUID	BRA	BRA	BLEND	CHKI	EXI7	EXI23		EXI55	EXIM	CMPI	BMAP	CHK	SLT	DIVUI
5x	CMPI	MLO	{VM}	VMFILL	CMOVNZ	BYTNDX	WYDENDX	UTF21NDX	SLL						MFLK	MTLK
6x	{SIMD}	{FLT1}	{FLT2}	{FLT3}	MUX	{DFLT1}	{DFLT2}	{DFLT3}		{PST1}	{PST2}	{PST3}	EXI41			
7x																
8x	LDB	LDBU	LDW	LDWU	LDT	LDTU	LDO	LDOS	LLAL	LLAH	LEA	LDVOAR	LDOO	LDCTX	LDOU	LDH
9x	STB	STW	STT	STO	STOC	STOS	STOO / STH	CAS	STSET	STMOV	STCMP	STFND		STCTX		CACHE
Ax						SYS	INT	MOV			{BTFLD}	MOVS	PUSH	PUSH 2R	PUSH 3R	ENTER
Bx	LDBX	LDBUX	LDWX	LDWUX	LDTX	LDTUX	LDOX	LDOOX	LLALX	LLAHX	LEAX	LDORX	POP	POP 2R	POP 3R	LEAVE
Cx	STBX	STWX	STTX	STOX	STOCX	STHX	STOOX					LINK	UNLINK	LDHX	LDOUX	CACHEX
Dx	CMPIL		MULIL	SLTIL	ADDIL	SUBFIL	SEQIL	SNEIL	ANDIL	ORIL	EORIL	SGTIL	SLTUIL	DIVIL	MULUIL	SGTUIL
Ex																
Fx	DEFCAT	NOP	RTS	CARRY		{BCD}	STP	SYNC	MEMSB	MEMDB	WFI	SEI	MBNEZ			

Thor2022 Root Opcode

	0	1	2	3	4	5	6	7
0x	0 BRK	1 {R1}	2 {R2}	3 {R3}	4 ADDI	5 SUBFI	6 MULI	7 {SYS}
	8 ANDI	9 ORI	10 EORI	11 CMPI	12 CMP	13 SUB	14 MULUI	15 CSR
1x	16 JEQZ	17	18 JNEZ	19 OR	20 EOR	21 MULFI	22 SEQI	23 SNEI
	24 SLTI	25 ADD	26 AND	27 SGTI	28 SLTUI	29 SGE	30 SLTU	31 SGTUI
2x	32 JMP	33 DJMP	34 JBCI / JBSI	35 BRA	36	37 JBC / JBS	38 JEQ	39 JNE
	40 JLT	41 JGE	42 JLE	43 JGT	44 SEQ	45 SNE	46	47 SGEU
3x	48	49	50	51	52	53	54	55
	56	57	58	59	60	61	62	63
4x	64 DIVI	65 CPUID	66 CHKI	67	68 BMAP	69 BLEND	70 EXI8	71 EXI8
	72 EXI24	73 EXI24	74 EXI40	75 EXI40	76 EXI56	77 EXI56	78 SLT	79 DIVUI
5x	80 EXIM	81	82 {VM}	83 VMFILL	84 CMOVNZ	85 BYTNDX	86 WYDENDX	87 TETRANDX
	88 SLL	89 SRL	90 SRA	91 ROL	92 ROR	93 SGEIL	94 MFLK	95 MTLK
6x	96 SGEI	97 {FLT1}	98 {FLT2}	99 {FLT3}	100 MUX	101 {DFLT1}	102 {DFLT2}	103 {DFLT3}
	104 SLEI	105 {PST1}	106 {PST2}	107 {PST3}	108 SLLI	109 SRLI	110 SRAI	111
7x	112	113	114	1115	116	117	118	119
	120	121	122	123	124	125	126	127
8x	128 LDB	129 LDBU	130 LDW	131 LDWU	132 LDT	133 LDTU	134 LDO	135 LDOU
	136 LDH	137 LDHS	138 LDHP	139 LDHR	140	141	142	143
9x	144 STB	145 STW	146 STT	147 STO	148 STH	149 STHS	150 STHC	151 STHP
	152 BSET	152 BMOV	154 BCMP	155 BFND	156	157	158	159 CACHE
Ax	160	161	162	163	164	165 SYS	166 INT	167
	168 STPTR	169 STPTRX	170 {BTFLD}	171	172 PUSH1	173	174 PUSH	175 ENTER
Bx	176 LDBX	177 LDBUX	178 LDWX	179 LDWUX	180 LDTX	181 LDTUX	182 LDOX	183 LDOUX
	184 LDHX	185	186 LDHPX	187 LDHRX	188 POP1	189	190 POP	191 LEAVE
Cx	192 STBX	193 STWX	194 STTX	195 STOX	196 STHX	197	198 STHCX	199 STHPX
	200	201	202	203	204	205	206	207 CACHEX
Dx	208 CMPIL	209 SLEIL	210 MULIL	211 SLTIL	212 ADDIL	213 SUBFIL	214 SEQIL	215 SNEIL
	216 ANDIL	217 ORIL	218 EORIL	219 SGTIL	220 SLTUIL	221 DIVIL	222 MULUIL	223 SGTUIL
Ex	224	225	226	227	228	229	230 SLEUIL	231 SGEUIL
	232	233	234	235	236	237	238	239
Fx	240 RTI	241 NOP	242 RTS	243 CARRY	244	245 {BCD}	246 STP	247 SYNC
	248 MEMSB	249 MEMDB	250 WFI	251 PFI	252 MBNEZ	253	254	255

{R1 – 0x01} Integer Monadic Register Ops – Func7

	000	001	010	011	100	101	110	111
x000	CNTLZ	CNTLO	CNTPOP		NOT	NEG	ABS	NABS
x001	SQRT			TST				
x010	PTRINC	TRANSFORM						
x011					VCMPRSS		VSCAN	
x100	CLIP							
x101	REVBIT	REVBIT.HP						PTGHASH
x110	SHA256 SIG0	SHA256 SIG1	SHA256 SUM0	SHA256 SUM1	SHA512 SIG0	SHA512 SIG1	SHA512 SUM0	SHA512 SUM1
x111	SM3P0	SM3P1						
1000	SEI	DI						
1001								
1010	AES64IM							
1011								
1100	NNA_TRIG	NNA_STAT	NNA_MFACT					
1101								

{R2 – 0x02} Integer Dyadic Register Ops – Func$_7$

	000	001	010	011	100	101	110	111
0000	NAND	NOR	XNOR	ORC	ADD	SUB	MUL	
0001	AND	OR	XOR	ANDC			MULU	MULH
0010	DIV	DIVU	DIVSU		PTRDIF	MULF	MULSU	PERM
0011	DIF	CHK	BYTNDX	WYDNDX	U21NDX	MULSUH	MULUH	MYST
0100	SLT / SGT	SGE / SLE	SLTU / SGTU	SGEU / SLEU			SEQ	SNE
0101	MIN	MAX	CMP		BIT	CMOVNZ	CLMUL	CLMULH
0110	BMM.or	BMM.xor	BMM	BMM	MUX		LIN2PL	LIN2PH
0111	VSLLV	VSLRV	VEX	VEINS	VGIDX	V2BITS	VBITS2V	
1000	SLL	SRL	SRA	ROL	ROR		CRYA	CRYS
1001	SLLH	SRLH	SRAH	ROLH	RORH			
1010	AES64DS	AES64DSM	AES64ES	AES64ESM	AES64KS1I	AES64KS2	SM4ED	SM4KS
1011								
1100	NNA_MTWT	NNA_MTIN	NNA_MTBIAS	NNA_MTFB	NNA_MTMC	NNA_MTBC		
1101								
1110	PLOT	LINE						RW_COEFF
1111	AND_OR	OR_AND	ANDC_OR					

{R3/R4 – 0x03} Triadic Register Ops

	000	001	010	011	100	101	110	111
0								
1	SLLP	SLLPI			BLEND	FDP		

{F1 - 0x61} Floating-Point Monadic Ops – Funct$_7$

	000	001	010	011	100	101	110	111
x000	FMOV	FRSQRTE	FTOI	ITOF			FSIGN	FMAN
x001	FSQRT	FS2D	FS2Q	FD2Q	FSTAT		ISNAN	FINITE
x010	FTX	FCX	FEX	FDX	FRM	TRUNC	FSYNC	FRES
x011	FSIGMOID	FD2S	FQ2S	FQ2D			FCLASS	UNORD
x100	FABS	FNABS	FNEG					
x101								
x110								
x111								

{F2– 0x62} Floating-Point Dyadic Ops – Funct$_7$

	000	001	010	011	100	101	110	111
x000	SCALEB		FMIN	FMAX	FADD	FSUB		
x001	FMUL	FDIV	FREM	FNXT				
x010	FCMP	FSEQ	FSLT	FSLE	FSNE	FCMPB	FSETM	
x011	CPYSGN	SGNINV	SGNAND	SGNOR	SGNXOR	SGNXNOR	FCLASS	
x100								
x101								
x110								
x111								

{F3– 0x63} Floating-Point Triadic Ops – Funct$_7$

	000	001	010	011	100	101	110	111
0	FMA	FMS	FNMA	FNMS				
1								

{DF1 - 0x65} Decimal Floating-Point Monadic Ops – Funct$_7$

	000	001	010	011	100	101	110	111
x000	MOV	RSQRTE	FTOI	ITOF			FSIGN	FMAN
x001	SQRT	FS2D	FS2Q	FD2Q	FSTAT		ISNAN	FINITE
x010	FTX	FCX	FEX	FDX	FRM	TRUNC	FSYNC	RES
x011	SIGMOID	FD2S	FQ2S	FQ2D			FCLASS	UNORD
x100	ABS	NABS	NEG					
x101								
x110								
x111								

{DF2} Decimal Floating-Point Dyadic Ops – Funct$_7$

	000	001	010	011	100	101	110	111
x000	SCALEB		DFMIN	DFMAX	DFADD	DFSUB		
x001	DFMUL	DFDIV	DFREM	DFNXT				
x010	DFCMP	DFSEQ	DFSLT	DFSLE	DFSNE	DFCMPB	DFSETM	
x011	CPYSGN	SGNINV	SGNAND	SGNOR	SGNXOR	SGNXNOR	FCLASS	
x100	DIVIDEND_ADJ	DIVISOR_ADJ	DIV_BITADJ					
x101								
x110								
x111								

{VM – 0x52} Vector Mask Register Ops – Func5

	000	001	010	011	100	101	110	111
00					VMADD	VMSUB		
01	VMAND	VMOR	VMXOR			VMCNTPOP	VMFIRST	VMLAST
10	MTVM	MFVM	MTVL	MFVL	MTLC	MFLC		
11					VMSLL		VMSRL	

{OSR2} System Ops

	000	001	010	011	100	101	110	111
x000								EXEC
x001	PUSHQ	POPQ	PEEKQ	STATQ	RESETQ	POPQI	PEEKQI	STATQI
x010	REX	PFI	WAI	RTE	SETKEY		DI	SEI
x011	SETTO	GETTO	GETZL		RGNRW		TLBRW	SYNC
x100	CSAVE	CRESTORE			LDPTG	STPTG	BASE	
x101	MFSEL	MTSEL						
x110	MVCI							
x111								

MPU Hardware

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 32-bit peripheral as that is all that is needed. The PIT while powerful turns out to be one of the simpler peripherals in the system.

System Usage

One programmable timer component, which includes four 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 is a free timing channel on the second timer component.

Each PIT is given a 4kB-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 $FFFFFFF960xxx

PIT#2 is located at $FFFFFFF961xxx

Registers

The PIT has 16 registers addressed as 32-bit I/O cells. It occupies 64 consecutive I/O locations. All registers are read-write except for the current counts which are read-only. The control registers all refer to a single control register which is accessible at three different addresses. Current count, max count and on time are all 32-bit accessible; all 32 bits must be read or written. The control register is byte accessible. It is possible to update only part of the control register. There is a separate byte for control information for each counter in the control register.

Regno	Access	Moniker	Purpose
00	R	CC0	Current Count
04	RW	MC0	Max count
08	RW	OT0	On Time
0C	RW	CTRL	Control
10	R	CC1	Current Count
14	RW	MC1	Max count
18	RW	OT1	On Time
1C	RW		Control – this is the same register as 0C
20	R	CC2	Current Count

24	RW	MC2	Max count
28	RW	OT2	On Time
2C	RW		Control – this is the same register as 0C
30	R	CC3	Current Count
34	RW	MC3	Max count
38	RW	OT3	On Time
3C	RW		Control – this is the same register as 0C

Control Register

The control register is split into four independent bytes. Each byte controls an individual timer. All four timer control bytes work in the same fashion so only one is described below. The very same control register is accessible at three different I/O locations. By locating all four timer controls in a single register it is possible to precisely synchronize timing operations.

31 24	23 16	15 8	7 0
Timer#3 Control	Timer#2 Control	Timer#1 Control	Timer#0 Control

Timer Control Byte

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.
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.
5 to 7	~	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.

Programming

The PIT is a memory mapped i/o device. The PIT is programmed using 32-bit load and store instructions (LDT and STT). 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.

PIC – Programmable Interrupt Controller

Overview

The programmable interrupt controller manages interrupt sources in the system and presents an interrupt signal to the cpu. 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.

PIC#1 is located at $FF950xxx.

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.

Registers

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

Regno	Access	Moniker	Purpose
00	R	CAUSE	interrupt cause code for currently interrupting source
04	RW	RE	request enable, a 1 bit indicates interrupt requesting is enabled for that interrupt, a 0 bit indicates the interrupt request is disabled.
08	W	ID	Disables interrupt identified by low order five data bits.
0C	W	IE	enables interrupt identified by low order five data bits
10			reserved
14	W	RSTE	resets the edge-sense circuit for edge sensitive interrupts, 1 bit for each interrupt source. This register has no effect on level sensitive sources. This register automatically resets to zero.
18	W	TRIG	software trigger of the interrupt specified by the low order five data bits.
20	W	ESL	The low bit for edge sensitivity selection. ESL and ESH combine to form a two bit select of the edge sensitivity. ESH,EHL — Sensitivity 00 — level sensitive interrupt 01 — positive edge sensitive

			10 negative edge sensitive 11 either edge sensitive
24	W	ESH	The high bit for edge sensitivity selection
80	RW	CTRL0	control register for interrupt #0
84	RW	CTRL1	control register for interrupt #1
…		…	
FC	RW	CTRL31	control register for interrupt #31

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 11	IRQ	This register determines which signal lines of the cpu are activated for the interrupt.
12	IE	This is the interrupt enable bit, 1 enables the interrupt, 0 disables it. This is the same bit reflected in the IE register.
13,14	ES	This bit controls edge sensitivity for the interrupt 00 = level, 01 = pos. edge sensitive, 10 = neg. edge sensitive. 11 = either edge sensitive. These same bits are present in the ESL, ESH registers.
15 to 31		reserved

UART – Universal Asynchronous Receiver / Transmitter

Overview

A UART component (Universal Asynchronous Transmitter / Receiver) is used for the asynchronous transmission and reception of data. Asynchronous referring to the lack of a clock signal during transmission or reception.

uart6551 is a WDC6551 register compatible uart. The uart is a 32-bit peripheral device. It may be used as an eight-bit peripheral by connecting the high order 24-bit data input lines to ground, and grounding select lines one to three.

Baud rate is controlled by clock divider which assumes a 200MHz baud reference clock input. If a different clock frequency is used, then the divider table will need to be updated. The baud rate may also be controlled via a clock divider register. This register is 24 bits so gives a minimum frequency of 11.92 Hz assuming a 200MHz clock. (200MHz / 2^24).

Special Features

- WDC6551 register compatibility

System Usage

The uart is located at $FF930xxx

Registers

There are only four registers in the design. The function of the low order eight bits of the registers matches the 6551 function. The controller honors byte lane selects so only the portion of the register selected is written.

Reg	Moniker	Description
00	UART_TRB	Transmit and receive buffer. Data written is transmitted, on a read data available is read. Also reads / writes the clock multiplier if access to clock multiplier is enabled.
04	UART_STAT	Status Register. Returns status bits on a read, a write of any value will cause a reset of some of the command register bits
08	UART_CMD	Command register
0C	UART_CTRL	Control register

UART_TRB

This register is 32-bits wide of which only the lower eight bits are used to transmit or receive data by the uart. Data written to the register is transmitted. A register read returns data received by the uart. When the fifo’s are enabled writing to this register writes to the transmit fifo. Reading this register reads the receive fifo. If clock divider access is enabled (via control register bit 31) then this register allows modifying or reading the clock divider value. Writing a clock divider value to this register automatically switches the function back to transmit / receive.

UART_STAT

Uart status register. Writing any value to the status register resets some of the uart's command bits.

Bit	Status	
0	Parity Error	1 = parity error occurred, 0 = no error
1	Framing Error	1 = framing error
2	Overrun	1 = overrun
3	Rx Full	1 = receiver data available
4	Tx Empty	1 = open slot in transmit fifo
5	DCD	0 = data carrier present
6	DSR	0 = data set ready
7	IRQ	1 = irq occurred
	Additional Line Status Byte	
8	reserved	
9	reserved	
10	reserved	
11	reserved	
12	Break received	1 if a break signal is received
13	Tx Full	1 = transmit fifo full
14	reserved	
15	G Rcv Err	1 = global receiver error (set if any error status is set)
	Additional Modem Status Byte	
16	CTS	1 = CTS line changed state
17	DSR	1 = DSR line changed state
18	RI	1 = RI line changed state
19	DCD	1 = DCD line changed state
20	CTS	CTS state
21	reserved	
22	RI	RI state
23	reserved	
	IRQ Status	
24,25	zero	these two bits are zero
26 to 28	IRQENC	encoded irq value (0 to 7)
29 to 30	reserved	
31	irq	IRQ is set

UART_CMD

Bit		
0	DTR	output 1 = low, 0 = high
1	RxIe	receiver interrupt enable 0 = enabled, 1 = disabled
2,3	RTS Control	
	00	output RTS high
	01	output RTS low, enable transmit interrupt
	10	output RTS low,
	11	output RTS low, send a break signal
4	LLB	1 = local loopback (receiver echo)
5 to 7	Parity Control	

	000	no parity
	001	odd parity
	011	even parity
	101	transmit mark parity (parity error disabled)
	111	transmit space parity (parity error disabled)
8	LSIe	line status change interrupt enable 1 = enabled
9	MSIe	modem status change interrupt enable 1 = enabled
10	RxToIe	receiver timeout interrupt enable 1 = enabled
11 to 31	reserved	

UART_CTRL

Bit			
0 to 3	Baud Rate		
	0000	Use 16x external clock	This table is expanded using an extra control bit #27.
	0001	50	
	0010	75	
	0011	109.92	
	0100	134.58	
	0101	150	
	0110	300	
	0111	600	
	1000	1200	
	1001	1800	
	1010	2400	
	1011	3600	
	1100	4800	
	1101	7200	
	1110	9600	
	1111	19200	
4	Rx clock source		1 = external, 0 = baud rate generator
5,6	Word length		code for word length in bits
	00	8	
	01	7	
	10	6	
	11	5	
7	Stop Bit		
	0	1	
	1	1 if 8 bits and parity	
	1	1.5 if 5 bits and no parity	
	1	2 otherwise	
8 to 15	reserved		do not use
16	Fifo enable		1 = fifo's enabled
17	Rx Fifo Clear		1 = clear receiver fifo
18	Tx Fifo Clear		1 = clear transmit fifo
19	reserved		
20,21	Transmit Threshold		Threshold for DMA signal activation If the transit fifo count is less than the threshold then a DMA transfer is triggered.
	0	1 byte	

		1	¼ full	
		2	½ full	
		3	¾ full	
22, 23	Receive Threshold			Threshold for DMA signal activation. If the receive fifo count is greater than the threshold then a DMA transfer is triggered.
		0	1 byte	
		1	¼ full	
		2	½ full	
		3	¾ full	
24	hwfc			1 = automatic hardware flow control
25	reserved			
26	dmaEnable			1 = dma enabled
27	Baud Rate bit 4			Extended baud rate selection bit, used in combination with bits 0 to 3.
		10000	38400	
		10001	57600	
		10010	115200	
		10011	230600	
		10100	460800	
		10101	921600	
		10110	reserved	
		10111	reserved	
		11xxx	reserved	
28,29	reserved			
30	selDV			1 = use clock divider register, 0 = use baud table
31	accessDV			1 = access clock divider via TRB register, 0 = normal TRB operation

Selecting the clock divider register as the baud source allows any programmable baud rate.

ToolSet

Compiler

The compiler currently available for Thor is the cc64 compiler. cc64 is a derivative of 'C' with a number of language extensions. Please see the cc64 language document for more information.

Assembler

Overview

The author has developed a backend for the excellent assembler, vasm which was developed by Volker Barthelmann. Please see the documentation for the assembler for more information. Contained in this document are features relevant to Thor.

Vasm is a flexible and powerful macro assembler. The backend is flexible in its recognition of mnemonics. There are many alternate mnemonics to the standard set that are recognized allowing the programmer some freedom in specifying operations.

Recognized Mnemonics

Register Names

The assembler will recognize the general-purpose registers either by their ABI usage names or by the name 'rn' as in 'r0', 'r1', 'r2' and so on. General-purpose registers given specific usages in the ABI may also be referenced by that set of names. For instance, the first argument register may be referenced as 'a0' which is also 'r1'. The names of the registers recognized by the assembler are shown in the table below.

	Alternate Name
r0	
r1	a0
r2	a1
r3 to r12	t0 to t9
r13 to r22	s0 to s9
r23 to r30	a2 to a9
r31 to r58	
r59	gp2 (rodata)
r60	gp1 (bss)
r61	gp / gp0 (data)
r62	fp
r63	sp

LC	

CA0	
CA1	LK1
CA2	LK2
CA3	
CA4	
CA5	
CA6	
CA7	IP
CA8 to CA15	

ZS	
DS	
ES	
FS	

GS	
HS	
SS	
CS	
PTA	
LDT	
KYT	
TCB	

DBAD0	address #0	
DBAD1	address #1	
DBAD2	address #2	
DBAD3	address #3	
DBCTRL	Debug Control	
DBSTAT	Debug Status	

Linker

The linker in use is vlink ‘out of the box’.

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.

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.

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 Franciso, 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

IBM® is a registered trademark of International Business Machines Corporation. Intel® is a registered trademark of Intel Corporation. HP® is a registered trademark of Hewlett-Packard Development Company. "SPARC® is a registered trademark of SPARC International, Inc.

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:		