# Preface

## Who This Book is For

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

## Motivation

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

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

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

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

## About the Author

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

# Nomenclature

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

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

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

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

## Little Endian vs big Endian

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

## Endian

Thor2024 is a little-endian machine. The difference between big endian and little endian is in the ordering of bytes in memory. Bits are also numbered from lowest to highest for little endian and from highest to lowest for big endian.

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

Little Endian:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Address | 3 | 2 | 1 | 0 |
| Byte | 3 | 2 | 1 | 0 |

Big Endian:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Address | 3 | 2 | 1 | 0 |
| Byte | 0 | 1 | 2 | 3 |

For Thor2024 the root opcode is in byte zero of the instruction and bytes are shown from right to left in increasing order. As the following table shows.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Address 3 | Address 2 | Address 1 | | Address 0 | |
| Byte 3 | Byte 2 | Byte 1 | | Byte 0 | |
|  |  |  |  | ▼ | |
| 31 24 | 23 16 | 15 8 | | 7 5 | 4 0 |
| Constant8 | Raspec8 | Rtspec8 | | Sz3 | Opcode5 |

# Programming Model

## Register File

### Rn – General Purpose Registers

The register file contains 32 128-bit general purpose registers.

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

The stack pointer, register 63, is banked with a separate stack pointer for each operation mode. Registers may be loaded or stored individually or in groups of four 128-bit values.

#### Register ABI

|  |  |  |  |
| --- | --- | --- | --- |
| Regno | ABI | Group Reg | ABI Usage |
| 0 | 0 | AG0 | Always zero |
| 1 | A0 | First argument / return value register |
| 2 | A1 | Second argument / return value register |
| 3 | A2 | Third argument register |
| 4 to 7 | T0 to T3 | TG0 | Temporary register, caller save |
| 8 to 11 | T4 to T7 | TG1 | Temporary register, caller save |
| 12 to 15 | S0 to S3 | SG0 | Saved register, register variables |
| 16 to 19 | S0 to S7 | SG1 | Saved register, register variables |
| 20 to 23 | A3 to A6 | AG1 | Argument register |
| 24 | S8 | G6 | Saved register, register variables |
| 25 | S9 | Saved register, register variables |
| 26 | S10 | Saved register, register variables |
| 27 | LC | Loop Counter |
| 28 | TP |  | Thread Pointer |
| 29 | GP |  | Global Pointer |
| 30 | FP |  | Frame Pointer |
| 31 | ASP |  | Application / User stack pointer |
| 31 | SSP |  | Supervisor Stack pointer |
| 31 | HSP |  | Hypervisor Stack pointer |
| 31 | MSP |  | Machine Stack pointer |

### Fn – Floating-Point Registers

The design includes a set of 32 128-bit floating point registers.

|  |  |  |  |
| --- | --- | --- | --- |
| Regno | ABI | Group Reg | ABI Usage |
| 0 | 0 | FG0 | Always zero |
| 1 | A0 | First argument / return value register |
| 2 | A1 | Second argument / return value register |
| 3 | A2 | Third argument register |
| 4 to 15 | T0 to T11 | FG1, FG2, FG3 | Temporary register, caller save |
| 16 to 27 | S0 to S11 | FG4, FG5, FG6 | Saved register, register variables |
| 28 to 31 | A3 to A6 | FG7 | Argument Register |

### Pn - Predicate Registers

There are 16 128-bit predicate registers.

Predicate registers are used to mask off vector operations so that a vector instruction doesn’t perform the operation on all elements of the vector.

### Code Address Registers

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

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

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

#### LRn – Link Registers

There are three registers in the Thor2024 architecture reserved for subroutine linkage. These registers are used to store the address after the calling instruction. They may be used to implement fast returns for several levels of subroutines or to used to call milli-code routines. The jump to subroutine, [JSR](#_JSR_–_Jump), and branch to subroutine, [BSR](#_BSR_–_Branch), instructions update a link register. The return from subroutine,. [RTS](#_RTS_–_Return), instruction is used to return to the next instruction.

#### PC – Program Counter

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

### LC - Loop Counter (reg 27)

The loop counter register is used in counted loops along the decrement and branch, [DBcc](#_DBcc_–_Decrement), and REP modifier instructions.

### SR - Status Register (CSR 0x?004)

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

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

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

T indicates that trace mode is active.

OM processor operating mode.

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

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

IPL is the interrupt mask level

RT specifies the return type for an [RTI](#_RTI_–_Return) instruction.

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

#### Decimal Mode

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

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

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

# Vector Programming Model

## Register File

### Vn – SIMD Registers

The SIMD register file contains 32 512-bit registers.

|  |  |  |
| --- | --- | --- |
| Regno | ABI | ABI Usage |
| 0 |  |  |
| 1 | VA0 | First argument / return value |
| 2 | VA1 | Second argument / return value |
| 3 | VA2 | Third argument |
| 4 to 15 | VT0 to VT11 |  |
| 16 to 27 | VS0 to VS11 |  |
| 28 to 31 | VA3 to VA6 |  |

### Vector Related CSRs

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

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

## Vector Global Mask Register (VGM)

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

## Vector Restart Mask Register (VRM)

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

## Vector Error Mask Register (VERR)

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

## Vector Register Size (VRGSZ)

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

## Vector Element Description Register (VED)

This register contains bits describing an element of a vector.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 127 6 | 5 3 | 2 0 |
|  | ~ | OT3 | Size3 |

|  |  |  |
| --- | --- | --- |
| Size3 | Bits | Bytes |
| 0 | 8 | 1 |
| 1 | 16 | 2 |
| 2 | 32 | 4 |
| 3 | 64 | 8 |
| 4 | 128 | 16 |
| 5 | 256 | 32 |
| 6 | 512 | 64 |
| 7 |  | reserved |

|  |  |
| --- | --- |
| OT3 | Operand Type |
| 0 | Integer |
| 1 | Float |
| 2 | Decimal |
| 3 | Posit |
| 4 | Char |
| 5 to 7 | reserved |

### Register-Register Format

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Fmt3 | Rb | Ra | Rt | Mask |
| 000 | scalar | scalar | scalar | No |
| 001 | scalar | scalar | scalar | Yes |
| 010 | scalar | vector | vector | No |
| 011 | scalar | vector | vector | Yes |
| 100 | vector | vector | vector | No |
| 101 | vector | vector | vector | Yes |

### Special Purpose Registers

#### SC - Stack Canary (GPR 53)

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

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

See status register.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 4 | 3 | 2 | 1 | 0 |
| ~ | mie | hie | sie | uie |

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

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

#### U\_REPBUF - (CSR – 0x008)

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

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

Pc: (64 bits) the address of the instruction following the REP

V: REP valid bit, 1 only if a REP instruction is active

ICnt: the current instruction count, distance from REP instruction.

Limit: a 32-bit amount to compare the loop counter against.

Ins: bits 9 to 15 of the REP instruction which contains the instruction count of instruction included in the repeat and condition under which the repeat occurs.

#### [U/S/H/M]\_SCRATCH – CSR 0x?041

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

#### S\_PTBR (CSR 0x1003)

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 95 14 | 13 12 | 11 8 | 7 6 | 5 4 | 3 | 2 1 | 0 |
| Page Table Address67..14 | ~2 | Levels | AL2 | ~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. For a normal page table gives the top entry level.

AL2: TLB entry replacement algorithm, 0=fixed,1=LRU,2=random,3=reserved

#### S\_ASID (CSR 0x101F)

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

#### S\_KEYS (CSR 0x1020 to 0x1027)

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

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | 23 0 |
| 1020 |  |  | key0 |
| 1021 |  |  | key1 |
| … |  |  | … |
| 1027 |  |  | key7 |

#### M\_CORENO (CSR 0x3001)

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

#### M\_TICK (CSR 0x3002)

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

#### M\_SEED (CSR 0x3003)

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

|  |  |  |
| --- | --- | --- |
| 63 60 | 59 16 | 15 0 |
| State4 | ~44 | seed16 |

|  |  |
| --- | --- |
| State4 Bit |  |
| 0 | dead |
| 1 | test |
| 2 | valid, the seed value is valid |
| 3 | Busy, the collector is busy collecting a new seed value |

#### M\_BADADDR (CSR 0x3007)

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

#### M\_BAD\_INSTR (CSR 0x300B)

This register contains a copy of the exceptioned instruction.

#### M\_SEMA (CSR 0x300C)

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

#### M\_TVEC – CSR 0x3030 to 0x3034

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

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

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

#### M\_SR\_STACK (CSR 0x303C to CSR 0x303D)

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

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

#### M\_IOS – IO Select Register (CSR 0x3100)

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

|  |  |
| --- | --- |
| 63 16 | 15 0 |
| Virtual Address67..20 | 016 |

#### M\_EPC (CSR 0x3108 to 0x310F)

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

|  |  |
| --- | --- |
| Reg # | Name |
| 0x3108 | EIP0 |
| … |  |
| 0x310F | EIP7 |

#### AV – Application Vector Table Address

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

|  |
| --- |
| 63 0 |
| App Vector Table Address67..4 |

#### VB – Vector Base Register

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

|  |  |  |
| --- | --- | --- |
| 63 3 | 2 | 1 0 |
| Vector Table Address63..3 | ~ | ~ |

# Operating Modes

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

A subset of instructions is limited to machine mode.

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

# Memory Management

## Bank Swapping

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

## The Page Map

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

## Regions

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

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

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

## PMA - Physical Memory Attributes Checker

### Overview

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

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

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

Physical memory attributes checking is applied in all operating modes.

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

### Region Table Description

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

#### PMT Address

The PMT address specifies the location of the associated PMT.

#### CTA – Card Table Address

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

#### Attributes

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

## Page Management Table - PMT

### Overview

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

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

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

### Location

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

## PMTE Description

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

Page Management Table Entry

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | N | M | ~9 | | C | E | AL2 | ~16 |
| ACL16 | | | | | | | | Share Count16 |
| Access Count32 | | | | | | | | |
| PL8 | | | | Key24 | | | | |

### Access Control List

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

### Share Count

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

### Access Count

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

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

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

### Key

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

### Privilege Level

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

### N

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

### M

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

### E

indicates if the page is encrypted.

### AL

indicates the compression algorithm used.

### C

The C indicator bit indicates if the page is compressed.

## Page Tables

### Intro

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

### Hierarchical Page Tables

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

A picture of multi-level page tables


### Inverted Page Tables

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

### The Simple Inverted Page Table

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

Diagram of Inverted Page Table


### Hashed Page Tables

#### Hashed Table Access

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

#### Clustered Hash Tables

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

### Shared Memory

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

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

### Specifics: Thor2023 Page Tables

### Thor2023 Hash Page Table Setup

#### Hash Page Table Entries - HPTE

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | LVL/BC5 | | RGN3 | M | A | T | S | G | SW2 | | CACHE4 | | MRWX3 | HRWX3 | SRWX3 | URWX3 |
| PPN31..0 | | | | | | | | | | | | | | | | |
| PPN63..32 | | | | | | | | | | | | | | | | |
| VPN37.. 6 | | | | | | | | | | | | | | | | |
| VPN69.. 38 | | | | | | | | | | | | | | | | |
| ~4 | | ASID11..0 | | | | | | | | ~2 | | VPN83.. 70 | | | | |

Fields Description

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

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

#### Small Hash Page Table Entries - SHPTE

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | LVL/BC5 | RGN3 | M | A | T | S | G | SW2 | CACHE4 | MRWX3 | HRWX3 | SRWX3 | URWX3 |
| VPN5..0 | | PPN25..0 | | | | | | | | | | | |
| ASID11..0 | | | | | | VPN25.. 6 | | | | | | | |

#### Page Table Groups – PTG

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

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

|  |
| --- |
| 127 0 |
| PTE0 |
| PTE1 |
| PTE2 |
| PTE3 |
| PTE4 |
| PTE5 |
| PTE6 |
| PTE7 |

#### Size of Page Table

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

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

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

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

#### Hash Function

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

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

#### Collision Handling

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

#### Finding a Match

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

#### Locality of Reference

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

#### Access Rights

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

### Thor2023 Hierarchical Page Table Setup

#### Page Table Entries - PTE

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

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

#### Page Table Entry Format – PTE

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | LVL/BC5 | RGN3 | M | A | T | S | G | SW2 | CACHE4 | MRWX3 | HRWX3 | SRWX3 | URWX3 |
| PPN31..0 | | | | | | | | | | | | | |
| PPN63..32 | | | | | | | | | | | | | |

#### Small Page Table Entry Format – SPTE

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | LVL/BC5 | RGN3 | M | A | T | S | G | SW2 | CACHE4 | MRWX3 | HRWX3 | SRWX3 | URWX3 |
| PPN31..0 | | | | | | | | | | | | | |

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

#### Super Pages

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

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

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

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

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | 15 | RGN3 | M | A | 0 | S | G | SW2 | ~4 | MRWX3 | | HRWX3 | SRWX3 | URWX3 |
| PPN=0x3FFFE022 | | | | | | | | | | | NPG=0x03F10 | | | |
| PPN=0xFFFFFFFF 63..32 | | | | | | | | | | | | | | |

The PTE would be pointed to by a LVL=1 pointer resulting in a 16MB super-page size. 512MB is 32 pages before the end of memory, reflected in the value 0x3FFFE022 for the PPN above. There are 64 x 16kB pages in 1MB so the length field, NPG, is set to 0x03f10.

##### PTE Format for 16MB page

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | 15 | RGN3 | M | A | 0 | S | G | SW2 | ~4 | MRWX3 | | HRWX3 | SRWX3 | URWX3 |
| PPN31..10 | | | | | | | | | | | NPG10 | | | |
| PPN63..32 | | | | | | | | | | | | | | |

##### PTE Format for 16GB page

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | 25 | RGN3 | M | A | 0 | S | G | SW2 | ~4 | MRWX3 | HRWX3 | SRWX3 | URWX3 |
| PPN31..20 | | | | | NPG20 | | | | | | | | |
| PPN63..32 | | | | | | | | | | | | | |

## TLB – Translation Lookaside Buffer

### Overview

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

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

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

### Size / Organization

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

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

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

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

### TLB Entries - TLBE

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

|  |  |
| --- | --- |
| Count6 | LRU3 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | LVL/BC5 | RGN3 | M | A | T | S | G | SW2 | CACHE4 | MRWX3 | HRWX3 | SRWX3 | URWX3 |
| PPN31..0 | | | | | | | | | | | | | |
| PPN63..32 | | | | | | | | | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| VPN41.. 10 | | | |
| VPN73.. 42 | | | |
| ~4 | ASID11..0 | ~5 | VPN83.. 73 |

### Small TLB Entries - TLBE

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

|  |  |
| --- | --- |
| Count6 | LRU3 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | LVL/BC5 | RGN3 | M | A | T | S | G | SW2 | CACHE4 | MRWX3 | HRWX3 | SRWX3 | URWX3 |
| ~6 | | PPN25..0 | | | | | | | | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| VPN41.. 10 | | | | |
| ~4 | ASID11..0 | PS | ~ | VPN55.. 42 |

### What is Translated?

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

### Page Size

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

### Ways

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

### Management

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

### Accessing the TLB

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

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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 00 | TLBE (PTE63..0) | | | | | | | | | | |
| 08 |  | | | | | TLBE (PTE95..64) | | | | | |
| 10 | TLBE (VPN63..0) | | | | | | | | | | |
| 18 |  | | | | | TLBE (VPN95..64) | | | | | |
| 20 | TLB Miss Address63..0 | | | | | | | | | | |
| 28 | ~4 | Miss ASID12 | | ~16 | | TLB Miss Address95..64 | | | | | |
| 30 to 68 |  | | | | | | | | | | |
| 70 |  | | | | | | AL2 | 0 | Entry Num10 | ~ | Way4 |
| 78 | RWTRIG | | WTRIG | RTRIG | ~8 | ~32 | | | | | |

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

### ?RWX3

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

### CACHE4

The cache4 field is combined with the cache attributes specified in the region table. The region table takes precedence; however, if the cache4 field indicates non-cache-ability then the data will not be cached.

#### Example TLB Update Routine

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

### TLB Entry Replacement Policies

The TLB supports three algorithms for replacement of entries with new entries on a TLB miss. These are fixed replacement (0), least recently used replacement (1) and random replacement (2). The replacement method is stored in the AL2 bits of the page table base register.

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

### Flushing the TLB

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

### Reset

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

#### Global Bit

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

## Card Table

### Overview

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

### Organization

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

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

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

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

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

### Location

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

### Operation

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

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

### Sample Write Barrier

; Milli-code routine for garbage collect write barrier.

; This sequence is short enough to be used in-line.

; Three level card memory.

; a2 is a register pointing to the card table.

; STPTR will cause an update of the master card table, and hardware card table.

;

GCWriteBarrier:

STPTR a0,[a1] ; store the pointer value to memory at a1

LSR t0,a1,#8 ; compute card address

STB r0,[a2+t0] ; clear byte in card memory

## Arithmetic Operations

### Representations

#### long

|  |
| --- |
| 127 0 |
| 128 bits |

#### int

|  |
| --- |
| 63 0 |
| 64 bits |

#### short

|  |
| --- |
| 31 0 |
| 32 bits |

#### char

|  |
| --- |
| 15 0 |
| 16 bits |

#### byte

|  |
| --- |
| 7 0 |
| 8 bits |

#### decimal

|  |  |
| --- | --- |
| 127 120 | 119 0 |
|  | 120 bits |

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

### Arithmetic Operations

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

**ADD Rt,Ra,Imm15**

**Instruction Format:** RI

|  |  |  |  |
| --- | --- | --- | --- |
| 31 17 | 16 12 | 11 7 | 6 0 |
| Immediate14..0 | Ra5 | Rt5 | 47 |

Immediate instructions may have the constant overridden via the use of postfixed immediates. In fact, almost all instructions can work with postfix immediates.

**ADD Rt,Ra,Imm24**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 17 | 16 12 | 11 7 | | 6 0 |
| ~14 | Ra5 | Rt5 | | 47 |
| Immediate23..0 | | | 1 | 1277 |

There may seem to be significant wasted space in the instruction when an instruction postfix is used. However, the use of a postfix is the rare case which occurs when a fifteen-bit immediate value is not sufficient. Having the postfix begin with bit 0 to 23 encoded is to allow for instructions that do not have space for an immediate field in the instruction. The postfix usage is kept consistent between all instructions to make decoding easier to handle and smaller resource wise.

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

**MULSU Rt, Ra, Rb**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 | 32 | 31 | 30 | 29 | 28 23 | 22 | 21 16 | 15 | 14 9 | 8 | 7 5 | 4 0 |
| 66 | 1 | Vc | 1 | Vb | Sb | Rb6 | Sa | Ra6 | St | Rt6 | V | Sz3 | 25 |
| Immediate31..0 | | | | | | | | | | | | 03 | 315 |

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

### ABS – Absolute Value

**Description:**

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 35 | Ra5 | Rt5 | 1h7 |

**Operation:**

If Ra < 0

Rt = -Ra

else

Rt = Ra

**Execution Units:** Integer ALU #0

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### ADD - Register-Register

**Description:**

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 43 | Rb5 | Ra5 | Rt5 | 2h7 |

**Operation: R2**

Rt = Ra + Rb

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### ADDI - Add Immediate

**Description:**

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

**Instruction Format:** RIQ

|  |  |  |
| --- | --- | --- |
| 15 12 | 11 7 | 6 0 |
| Imm3..0 | Rt5 | 127 |

**Instruction Format:** RI

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Immediate8..0 | Ra5 | Rt5 | 47 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:**

**Notes:**

### ADDIPC - Add Immediate to Program Counter

**Description:**

Add the program counter and immediate value and place the sum in the target register. The immediate is sign extended to the machine width. This instruction is used to generate relative addresses.

**Instruction Format:** ADDIPC

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Immediate13..0 | Rt5 | 967 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:**

**Notes:**

### AND – Bitwise And

**Description:**

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 03 | Rb5 | Ra5 | Rt5 | 2h7 |

**Operation: R2**

Rt = Ra & Rb

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### ANDI - Add Immediate

**Description:**

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

**Instruction Format:** RI

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Immediate8..0 | Ra5 | Rt5 | 87 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:**

**Notes:**

### BMAP – Byte Map

**Description:**

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

**Instruction Format:** R2

**BMAP Rt, Ra, Rb**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 33 | Rb5 | Ra5 | Rt5 | 3h7 |

**BMAP Rt, Ra, Imm64**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 33 | 05 | Ra5 | Rt5 | 3h7 |

|  |  |  |  |
| --- | --- | --- | --- |
| 79 8 | | 7 | 6 0 |
| ~8 | Immediate64 | 1 | 1267 |

**Operation:**

**Vector Operation**

**Execution Units:** First Integer ALU

**Exceptions:** none

**Notes:**

### BMM – Bit Matrix Multiply

BMM Rt, Ra, Rb

**Description**:

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 23 | Rb5 | Ra5 | Rt5 | 3h7 |

**Operation**:

for I = 0 to 7

for j = 0 to 7

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

**Clock Cycles:** 1

**Execution Units: First Integer** ALU

**Exceptions**: none

**Notes**:

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

### CHARNDX – Character Index

**Description:**

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

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

**Instruction Format: R2 (byte)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 53 | Rb5 | Ra5 | Rt5 | 3h7 |

**Instruction Format: R2 (wyde)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 63 | Rb5 | Ra5 | Rt5 | 3h7 |

**Instruction Format: R2 (tetra)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 73 | Rb5 | Ra5 | Rt5 | 3h7 |

**Operation:**

Rt = Index of (Rb in Ra)

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### CLMUL – Carry-less Multiply

**Description**:

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

**Instruction Formats:**

**Instruction Format:** R2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 63 | Rb5 | Ra5 | Rt5 | 197 |

**Exceptions**: none

**Execution Units: First** Integer ALU

Operations

Rt = Ra \* Rb

**Vector Operation**

for x = 0 to VL - 1

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

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

**Exceptions**: none

### CMP - Comparison

**Description:**

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

**Operation:**

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

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** R2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 33 | Rb5 | Ra5 | Rt5 | 2h7 |

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

### CMPI – Compare Immediate

**Description:**

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

**Operation:**

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

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** RI

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Immediate8..0 | Ra5 | Rt5 | 117 |

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

### CNTLZ – Count Leading Zeros

**Description:**

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 05 | Ra5 | Rt5 | 1h7 |

**Operation:**

**Execution Units:** Integer ALU #0

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### CNTLO – Count Leading Ones

**Description:**

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 15 | Ra5 | Rt5 | 1h7 |

**Operation:**

**Execution Units:** Integer ALU #0

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### CNTPOP – Count Population

**Description:**

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 25 | Ra5 | Rt5 | 1h7 |

**Operation:**

**Execution Units:** Integer ALU #0

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### EOR – Bitwise Exclusive Or

**Description:**

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 23 | Rb5 | Ra5 | Rt5 | 2h7 |

**Operation: R2**

Rt = Ra ^ Rb

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### EORI – Exclusive Or Immediate

**Description:**

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

**Instruction Format:** RI

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Immediate8..0 | Ra5 | Rt5 | 107 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:**

**Notes:**

### MAX – Maximum Value

**Description:**

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

**Instruction Format: R2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 13 | Rb5 | Ra5 | Rt5 | 3h7 |

**Execution Units:** ALU #0 only

**Operation:**

IF (Ra > Rb)

Rt = Ra

else

Rt = Rb

### MIN – Minimum Value

**Description:**

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

**Instruction Format: R2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 03 | Rb5 | Ra5 | Rt5 | 3h7 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

**Operation:**

IF (Ra < Rb)

Rt = Ra

else

Rt = Rb

### MFLK – Move from Link Register

**Description**:

This instruction moves a link register to a general-purpose register.

**Instruction Format:** MFLK

|  |  |  |
| --- | --- | --- |
| 15 12 | 11 7 | 6 0 |
| L4 | Rt5 | 947 |

### MTLK – Move to Link Register

**Description**:

This instruction moves a general-purpose register to a link register.

**Instruction Format:** MFLK

|  |  |  |
| --- | --- | --- |
| 15 12 | 11 7 | 6 0 |
| L4 | Rs5 | 957 |

### MUX – Multiplex

**Description:**

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

**Instruction Format:** BITFLD

**MUX Rt, Ra, Rb, Rc**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 38 | 37 | 36 | 35 | 33 29 | 28 24 | 2322 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 03 | 0 | 0 | Ot2 | ~4 | Rc5 | ~2 | Rb5 | Ra5 | Rt5 | 1007 |

**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

### OR – Bitwise Or

**Description:**

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 13 | Rb5 | Ra5 | Rt5 | 2h7 |

**Operation: R2**

Rt = Ra | Rb

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### ORI - Or Immediate

**Description:**

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

**Instruction Format:** RI

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Immediate8..0 | Ra5 | Rt5 | 97 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:**

**Notes:**

### PFX – Constant Postfix

**Description:**

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

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

**Instruction Format: PFX0**

This format provides an eight-bit constant, and sign extends the value to the width of the constant prefix buffer.

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 | 6 0 |
| Immediate8 | 0 | 1247 |

**Instruction Format: LPFX0**

This format provides a twenty-four-bit constant, and sign extends the value to the width of the constant prefix buffer.

|  |  |  |
| --- | --- | --- |
| 31 8 | 7 | 6 0 |
| Immediate24 | 1 | 1247 |

**Instruction Format: PFX1**

This format provides a forty-bit constant, and sign extends the value to the width of the constant prefix buffer.

|  |  |  |
| --- | --- | --- |
| 47 8 | 7 | 6 0 |
| Immediate40 | 0 | 1257 |

**Instruction Format: LPFX1**

This format provides a fifty-six-bit constant, and sign extends the value to the width of the constant prefix buffer.

|  |  |  |
| --- | --- | --- |
| 63 8 | 7 | 6 0 |
| Immediate56 | 1 | 1257 |

**Instruction Format: PFX2**

This format provides a seventy-two-bit constant, and sign extends the value to the width of the constant prefix buffer.

|  |  |  |
| --- | --- | --- |
| 79 8 | 7 | 6 0 |
| Immediate72 | 1 | 1267 |

**Instruction Format: LPFX2**

This format provides an eighty-eight-bit constant, and sign extends the value to the width of the constant prefix buffer.

|  |  |  |
| --- | --- | --- |
| 95 8 | 7 | 6 0 |
| Immediate88 | 1 | 1267 |

**Instruction Format: PFX3**

This format provides a 104-bit constant, and sign extends the value to the width of the constant prefix buffer.

|  |  |  |
| --- | --- | --- |
| 111 8 | 7 | 6 0 |
| Immediate104 | 0 | 1277 |

**Instruction Format: LPFX3**

This format provides a 128-bit constant, and sign extends the value to the width of the constant prefix buffer.

|  |  |  |  |
| --- | --- | --- | --- |
| 143 8 | | 7 | 6 0 |
| ~8 | Immediate128 | 1 | 1277 |

### PTRDIF – Difference Between Pointers

**Description**:

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

**Instruction Format:** BITFLD

**PTRDIF Rt, Ra, Rb, Rc**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 38 | 37 | 36 | 35 | 3331 | 30 24 | 2322 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 13 | Ci | 0 | 02 | ~2 | Rc7 | ~2 | Rb5 | Ra5 | Rt5 | 1007 |

**Operation**:

Rt = Abs(Ra – Rb) >> Rc[3:0]

**Clock Cycles**: 1

**Execution Units: Integer**

**Exceptions**:

None

### REVBIT – Reverse Bit Order

**Description:**

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 55 | Ra5 | Rt5 | 1h7 |

**Operation:**

**Execution Units:** I

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### SEQ – Set if Equal

**Description:**

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

**Operation:**

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

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** R2

SEQ Rt, Ra, Rb

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 03 | Rb5 | Ra5 | Rt5 | 207 |

SEQ Rt, Ra, Imm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | | 6 0 |
| Fmt3 | Pr4 | 03 | 05 | Ra5 | Rt5 | | 207 |
| Immediate24 | | | | | | 2528 | |

### SLE – Set if Less Than or Equal

**Description:**

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

**Operation:**

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

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** R2

SLE Rt, Ra, Rb

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 33 | Rb5 | Ra5 | Rt5 | 207 |

### SLEU – Set if Unsigned Less Than or Equal

**Description:**

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

**Operation:**

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

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** R2

SLEU Rt, Ra, Rb

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 53 | Rb5 | Ra5 | Rt5 | 207 |

### SLT – Set if Less Than

**Description:**

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

**Operation:**

Rt = Ra < Rb or Rt = Ra < Imm

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** R2

SLT Rt, Ra, Rb

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 23 | Rb5 | Ra5 | Rt5 | 207 |

### SLTU – Set if Unsigned Less Than

**Description:**

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

**Operation:**

Rt = Ra < Rb or Rt = Ra < Imm

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** R2

SLTU Rt, Ra, Rb

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 43 | Rb5 | Ra5 | Rt5 | 207 |

### SNE – Set if Not Equal

**Description:**

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

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

**Operation:**

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

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** R2

SNE Rt, Ra, Rb

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 13 | Rb5 | Ra5 | Rt5 | 207 |

### SUB – Subtract Register-Register

**Description:**

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 53 | Rb5 | Ra5 | Rt5 | 2h7 |

**Operation: R2**

Rt = Ra - Rb

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### WYDENDX – Wyde Index

**Description:**

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

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

**Instruction Format: R2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 63 | Rb5 | Ra5 | Rt5 | 3h7 |

**Operation:**

Rt = Index of (Rb in Ra)

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

## Shift and Rotate Operations

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

Thor2024 has a full complement of shift instructions including rotates.

### ASL –Arithmetic Shift Left

**Description**:

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 | 23 | 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 0 | ~ | B | Rb5 | Ra5 | Rt5 | 887 |

**Operation:**

Rt = Ra << Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### ASLI –Arithmetic Shift Left

**Description**:

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

**Instruction Format:** RIS

The RIS format shifts a target register by a small immediate constant.

|  |  |  |
| --- | --- | --- |
| 15 12 | 11 7 | 6 0 |
| Imm3..0 | Rt5 | 127 |

**Instruction Format:** RI7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 | 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | B | 1 | Immediate6..0 | Ra5 | Rt5 | 887 |

**Operation:**

Rt = Ra << Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### ASR –Arithmetic Shift Right

**Description**:

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 | 23 | 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 0 | ~ | 0 | Rb5 | Ra5 | Rt5 | 907 |

**Operation:**

Rt = Ra << Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### ASRI –Arithmetic Shift Right

**Description**:

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

**Instruction Format:** RI7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 | 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | 0 | 1 | Immediate6..0 | Ra5 | Rt5 | 887 |

**Operation:**

Rt = Ra << Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### LSR –Logic Shift Right

**Description**:

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 | 23 | 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 0 | ~ | B | Rb5 | Ra5 | Rt5 | 897 |

**Operation:**

Rt = Ra >> Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### LSRI –Logical Shift Right

**Description**:

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

**Instruction Format:** RI7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 | 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | B | 1 | Immediate6..0 | Ra5 | Rt5 | 897 |

**Operation:**

Rt = Ra >> Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### ROL –Rotate Left

**Description**:

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 | 23 | 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 0 | ~ | 0 | Rb5 | Ra5 | Rt5 | 917 |

**Operation:**

Rt = Ra << Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### ROLI –Rotate Left by Immediate

**Description**:

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

**Instruction Format:** RI7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 | 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | 0 | 1 | Immediate6..0 | Ra5 | Rt5 | 917 |

**Operation:**

Rt = Ra << Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### ROR –Rotate Right

**Description**:

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 | 23 | 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 0 | ~ | 0 | Rb5 | Ra5 | Rt5 | 927 |

**Operation:**

Rt = Ra >> Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### RORI –Rotate Right by Immediate

**Description**:

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

**Instruction Format:** RI7

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 | 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | 0 | 1 | Immediate6..0 | Ra5 | Rt5 | 927 |

**Operation:**

Rt = Ra >> Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

## Bit Manipulation Operations

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

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

### General Format of Bitfield Instructions

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

**CLR Rt, Ra, Rb, Rc**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 38 | 37 | 36 | 3534 | 3332 | 30 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | Fn3 | Ci | Bi | Ot2 | ~2 | Me7 | Mb7 | Ra5 | Rt5 | 937 |

Mb7 may be either a register spec or a seven-bit immediate constant specifying the start position of the bitfield.

Me7 may be either a register spec or a seven-bit immediate constant specifying the end position of the bitfield.

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

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

The Ot2 field determines the operand type for operands Ra and Rt.

|  |  |
| --- | --- |
| Ot2 | Ra, Rt |
| 0 | Integer |
| 1 | Float |
| 2 | predicate |
| 3 | reserved |

### CLR – Clear Bit Field

**Description:**

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

**Operation:**

**Instruction Format:** BITFLD

**CLR Rt, Ra, Rb, Rc**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 38 | 37 | 36 | 35 | 33 32 | 30 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 03 | Ci | Bi | Ot2 | ~2 | Me7 | Mb7 | Ra5 | Rt5 | 1047 |

Mb7 may be either a register spec or a seven-bit immediate constant specifying the start position of the bitfield.

Me7 may be either a register spec or a seven-bit immediate constant specifying the end position of the bitfield.

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

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

Ot2 indicates to use predicate registers for the Ra and Rt operands.

**Clock Cycles:**

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### COM – Complement Bit Field

**Description:**

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

**Operation:**

**Instruction Format:** BITFLD

**COM Rt, Ra, Rb, Rc**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 38 | 37 | 36 | 35 | 33 32 | 30 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 23 | Ci | Bi | Ot2 | ~2 | Me7 | Mb7 | Ra5 | Rt5 | 1047 |

Mb7 may be either a register spec or a seven-bit immediate constant specifying the start position of the bitfield.

Me7 may be either a register spec or a seven-bit immediate constant specifying the end position of the bitfield.

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

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

Ot2 indicates to use predicate registers for the Ra and Rt operands.

**Clock Cycles:**

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### EXT – Extract Bit Field

**Description:**

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

**Operation:**

**Instruction Format:** BITFLD

**COM Rt, Ra, Rb, Rc**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 38 | 37 | 36 | 35 | 33 32 | 30 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 53 | Ci | Bi | Ot2 | ~2 | Me7 | Mb7 | Ra5 | Rt5 | 1047 |

Mb7 may be either a register spec or a seven-bit immediate constant specifying the start position of the bitfield.

Me7 may be either a register spec or a seven-bit immediate constant specifying the end position of the bitfield.

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

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

Ot2 indicates to use predicate registers for the Ra and Rt operands.

**Clock Cycles:**

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### EXTU – Extract Unsigned Bit Field

**Description:**

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

**Operation:**

**Instruction Format:** BITFLD

**COM Rt, Ra, Rb, Rc**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 38 | 37 | 36 | 35 | 33 32 | 30 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 43 | Ci | Bi | Ot2 | ~2 | Me7 | Mb7 | Ra5 | Rt5 | 1047 |

Mb7 may be either a register spec or a seven-bit immediate constant specifying the start position of the bitfield.

Me7 may be either a register spec or a seven-bit immediate constant specifying the end position of the bitfield.

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

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

Ot2 indicates to use predicate registers for the Ra and Rt operands.

**Clock Cycles:**

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

### SET – Set Bit Field

**Description:**

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

**Operation:**

**Instruction Format:** BITFLD

**CLR Rt, Ra, Rb, Rc**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 38 | 37 | 36 | 35 | 33 32 | 30 24 | 23 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 13 | Ci | Bi | Ot2 | ~2 | Me7 | Mb7 | Ra5 | Rt5 | 1047 |

Mb7 may be either a register spec or a seven-bit immediate constant specifying the start position of the bitfield.

Me7 may be either a register spec or a seven-bit immediate constant specifying the end position of the bitfield.

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

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

Ot2 indicates to use predicate registers for the Ra and Rt operands.

**Clock Cycles:**

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

## Cryptographic Accelerator Instructions

### AES64DS – Final Round Decryption

**Description**:

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 185 | Ra5 | Rt5 | 1h7 |

**Operation:**

**Exceptions:** none

### AES64DSM – Middle Round Decryption

**Description**:

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 195 | Ra5 | Rt5 | 1h7 |

**Operation:**

**Exceptions:** none

### AES64ES – Final Round Encryption

**Description**:

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 205 | Ra5 | Rt5 | 1h7 |

**Operation:**

**Exceptions:** none

### AES64ESM – Middle Round Encryption

**Description**:

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 215 | Ra5 | Rt5 | 1h7 |

**Operation:**

**Exceptions:** none

### SHA256SIG0

**Description:**

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

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 245 | Ra5 | Rt5 | 1h7 |

**Operation:**

Rt = sign extend(ror32(Ra,7) ^ ror32(Ra,18) ^ (Ra32 >> 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 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 255 | Ra5 | Rt5 | 1h7 |

**Clock Cycles: 1**

**Operation:**

Rt = sign extend(ror32(Ra,17) ^ ror32(Ra,19) ^ (Ra32 >> 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 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 265 | Ra5 | Rt5 | 1h7 |

**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 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 275 | Ra5 | Rt5 | 1h7 |

**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 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 285 | Ra5 | Rt5 | 1h7 |

**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 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 295 | Ra5 | Rt5 | 1h7 |

**Operation:**

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

**Execution Units:** ALU #0

**Exceptions:** none

### SHA512SUM0

Description:

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 305 | Ra5 | Rt5 | 1h7 |

### SHA512SUM1

Description:

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 315 | Ra5 | Rt5 | 1h7 |

### SM3P0

Description:

P0 transform of SM3 hash function.

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 145 | Ra5 | Rt5 | 1h7 |

**Operation**

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

### SM3P1

Description:

P1 transform of SM3 hash function.

**Instruction Format:** R1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 155 | Ra5 | Rt5 | 1h7 |

**Operation**

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

## Floating-Point Operations

### Precision

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

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

### Representations

#### Binary Floats

Triple Precision, Float:128

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

Quad Precision, long double

|  |  |  |
| --- | --- | --- |
| 127 | 126 112 | 111 0 |
| S | Exponent15 | Significand112 |

Double Precision, double

|  |  |  |
| --- | --- | --- |
| 63 | 62 52 | 51 0 |
| S | Exponent11 | Significand52 |

Single Precision, float

|  |  |  |
| --- | --- | --- |
| 31 | 30 23 | 22 0 |
| S | Exponent8 | Significand23 |

Half Precision, short float

|  |  |  |
| --- | --- | --- |
| 15 | 14 10 | 9 0 |
| S | Exponent5 | Significand10 |

#### Decimal Floats

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

|  |  |  |  |
| --- | --- | --- | --- |
| 127 | 126 122 | 121 110 | 109 0 |
| S | Combo5 | Exponent12 | Significand110 |

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

The exponent is a power of ten as a binary number with an offset of 1535. Range is 10-1535 to 101536

64-bit double precision decimal floating point:

|  |  |  |  |
| --- | --- | --- | --- |
| 63 | 62 58 | 57 50 | 49 0 |
| S | Combo5 | Exponent8 | Significand50 |

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

32-bit single precision decimal floating point:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 | 30 26 | 25 20 | 19 0 |
| S | Combo5 | Exponent6 | Significand20 |

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

### Rounding Modes

#### Binary Float Rounding Modes

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

#### Decimal Float Rounding Modes

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

### FABS – Absolute Value

**Description:**

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

**Integer Instruction Format: R1**

**FABS Rt, Ra**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | ~3 | 05 | Ra5 | Rt5 | 977 |

**Clock Cycles: 1**

**Operation:**

FPt = Abs(FPa)

**Execution Units:** FPU #0

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### FADD –Float Addition

**Description:**

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

**Supported Operand Sizes:**

**Operation:**

Rt = Ra + Rb or Rt = Ra + Imm

**Clock Cycles:** 8

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** FLT2

FADD Rt, Ra, Rb

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 36 | 35 33 | 32 27 | 26 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 45 | Rm3 | ~6 | ~5 | Rb5 | Ra5 | Rt5 | 987 |

### FADDI –Float Add Immediate

**Description:**

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

**Supported Operand Sizes:**

**Operation:**

Rt = Ra + Rb or Rt = Ra + Imm

**Clock Cycles:** 8

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** R2

FADDI Rt, Ra, Imm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 36 | 35 33 | 32 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 205 | Rm3 | Imm16 | Ra5 | Rt5 | 987 |

### FCMP - Comparison

**Description:**

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as floating-point values. This instruction may compare against lower precision immediate values to conserve code space.

**Supported Operand Sizes:**

**Operation:**

Rt = Ra ? Rb or Rt = Ra ? Imm

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** FLT2

FCMP Rt, Ra, Rb

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 36 | 35 33 | 32 27 | 26 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 135 | ~3 | ~6 | ~5 | Rb5 | Ra5 | Rt5 | 987 |

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

### FCMPI – Float Compare Immediate

**Description:**

Compare two source operands and place the result in the target register. The result is a vector identifying the relationship between the two source operands as floating-point values. This instruction may compare against lower precision immediate values to conserve code space.

**Supported Operand Sizes:**

**Operation:**

Rt = Ra ? Rb or Rt = Ra ? Imm

**Clock Cycles:** 1

**Execution Units:** All Integer ALU’s

**Exceptions:** none

**Notes:**

**Instruction Format:** FLT2

FCMPI Rt, Ra, Imm

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 36 | 35 33 | 32 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 295 | Rm3 | Imm16 | Ra5 | Rt5 | 987 |

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

### FSEQ – Float Set if Equal

**Description:**

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

**Operation:**

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

**Clock Cycles:** 1

**Execution Units:** All FPU’s

**Exceptions:** none

**Notes:**

**Instruction Formats:**

FSEQ Rt, Ra, Rb

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 36 | 35 | 34 27 | 26 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 85 | Tt1 | ~8 | ~5 | Rb5 | Ra5 | Rt5 | 987 |

Tt1: Target register 0 = general-purpose register, 1 = predicate register

### FSNE – Float Set if Not Equal

**Description:**

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

**Operation:**

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

**Clock Cycles:** 1

**Execution Units:** All FPU’s

**Exceptions:** none

**Notes:**

**Instruction Formats:**

FSNE Rt, Ra, Rb

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 36 | 35 | 34 27 | 26 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 85 | Tt1 | ~8 | ~5 | Rb5 | Ra5 | Rt5 | 987 |

Tt1: Target register 0 = general-purpose register, 1 = predicate register

## Load / Store Instructions

### Overview

### Addressing Modes

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

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

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

### Load Formats

#### Register Indirect with Displacement Format

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

**Instruction Format:** d[Rn]

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 19 | 1817 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Disp6..0 | Ot2 | Ra5 | Rt5 | 647 |

**Instruction Format:** d[Ra+Rb\*Sc]

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 36 | 3534 | 33 25 | 24 22 | 22 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Fn5 | Ot2 | Disp8..0 | Sc3 | Rb5 | Ra5 | Rt5 | 797 |

### 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 |
| Displacement28..0 | Ra5 | Rs5 | 0 | Opcode8 |

#### Indexed Format

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 | 27 | 2625 | 24 | 23 19 | 18 14 | 13 9 | 8 | 7 0 |
| m3 | z | C | ~2 | Tb | Rb5 | Ra5 | Rs5 | v | Opcode8 |

### CACHE <cmd>,<ea>

**Description:**

Issue command to cache controller.

**Instruction Format:** d[Rn]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Immediate8..0 | Ra5 | Cmd5 | 757 |

**Instruction Format:** d[Ra+Rb\*]

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 2524 | 22 | 22 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | 32 | Sc | Rb5 | Ra5 | Cmd5 | 787 |

Notes:

|  |  |  |
| --- | --- | --- |
| Cmd5 | Cache |  |
| ???00 | Ins. | Invalidate cache |
| ???01 | Ins. | Invalidate line |
| ???10 | TLB | Invalidate TLB |
| ???11 | TLB | Invalidate TLB entry |
| 000?? | Data | Invalidate cache |
| 001?? | Data | Invalidate line |
| 010?? | Data | Turn cache off |
| 011?? | Data | Turn cache on |

## Block Instructions

### BFND – Block Find

**Description:**

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

**Instruction Format:** d[Ra+Rb\*Sc]

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 41 | 40 36 | 3534 | 33 30 | 2928 | 27 25 | 24 22 | 22 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | 285 | Ot2 | Pt4 | Bi2 | Sz3 | Sc3 | Rb5 | Ra5 | Rs5 | 797 |

|  |  |
| --- | --- |
| **Bi2** | **Rb** |
| **0** | **No Change** |
| **1** | **Increment by 64** |
| **2** | **Decrement by 64** |
| **3** | **reserved** |

**Execution Units:** Memory

**Operation:**

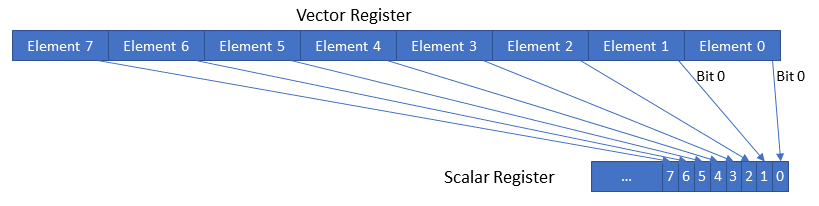
## Vector Specific Instructions

### V2BITS

**Description**

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

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



**Instruction Format: R2**

**Instruction Format:** R2

**V2BITS Rt, Ra, Rb**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 7 | 6 0 |
| Fmt3 | Pr4 | 03 | Rb5 | Ra5 | Rt5 | 187 |

**V2BITS Prt, Ra, Rb**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 22 | 21 17 | 16 12 | 11 | 10 7 | 6 0 |
| Fmt3 | Pr4 | 63 | Rb5 | Ra5 | ~ | Prt4 | 187 |

**Operation**

For x = 0 to VL-1

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

**Exceptions:** none

**Example:**

|  |
| --- |
| cmp v1,v2,v3 ; compare vectors v2 and v3  v2bits pr1,v1,#8 ; move NE status to bits in m1  vadd v4,v5,v6,pr1 ; perform some masked vector operations  vmuls v7,v8,v9,pr1  vadd v7,v7,v4,pr1 |

### 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 |
| 13h5 | ~9 | ~3 | Rt6 | 0 | 52h8 |

**Operation**

Rt = VL

**Execution Units:** ALUs

## Predicate Operations

### MFPR – Move from Predicate

**Description**

Move a predicate register to a general-purpose register.

**Instruction Format: PR2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 2019 | 18 15 | 14 12 | 11 7 | 6 0 |
| ~3 | Pr4 | 114 | ~2 | Prb4 | ~3 | Rt5 | 48h8 |

**Operation**

Rt = Pr

**Execution Units:** ALUs

### MTPR – Move to Predicate

**Description**

Move a general-purpose register to a predicate register.

**Instruction Format: PR2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 20 17 | 16 12 | 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 124 | ~4 | Ra5 | ~ | Prt4 | 48h8 |

**Operation**

Prt = Ra

**Execution Units:** ALUs

### PRADD – Predicate Add

**Description:**

Add the contents of two predicate registers and place the result in a predicate register.

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 2019 | 18 15 | 14 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 44 | ~2 | Prb4 | Pra4 | Prt4 | 48h8 |

1 clock cycle

**Exceptions:** none

### PRAND – Predicate And

**Description:**

Bitwise ‘and’ the contents of two predicate registers and place the result in a predicate register.

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 2019 | 18 15 | 14 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 84 | ~2 | Prb4 | Pra4 | Prt4 | 48h8 |

**Exceptions:** none

### PRASL – Predicate Arithmetic Shift Left

**Description:**

Shift a predicate register to the left up to 63 bits.

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 20 15 | 14 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 04 | Amt6 | Pra4 | Prt4 | 48h8 |

1 clock cycle

**Exceptions:** none

### PRCNTPOP – Predicate Population Count

**Description**

Count the set bits in a predicate register and store result in a general-purpose register.

**Instruction Format: PR2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 2019 | 18 15 | 14 12 | 11 7 | 6 0 |
| ~3 | Pr4 | 134 | ~2 | Prb4 | ~3 | Rt5 | 48h8 |

**Operation**

Rt = Pr

**Execution Units:** ALUs

### PREOR – Predicate Exclusive Or

**Description:**

Bitwise exclusive ‘or’ the contents of two predicate registers and place the result in a predicate register.

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 2019 | 18 15 | 14 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 104 | ~2 | Prb4 | Pra4 | Prt4 | 48h8 |

**Exceptions:** none

### PRFILL – Predicate Fill Mask

**Description:**

Generate a mask between Mb, mask begin, and Me, Mask End and store in predicate register.

**Instruction Format: VMR2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 18 | 17 11 | 10 7 | 6 0 |
| ~3 | Pr4 | Me7 | Mb7 | Prt4 | 49h7 |

1 clock cycle

**Exceptions:** none

### PRFIRST – 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 -1. The search begins at the least significant bit and proceeds to the most significant bit.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 2019 | 18 15 | 14 12 | 11 7 | 6 0 |
| ~3 | Pr4 | 144 | ~2 | Prb4 | ~3 | Rt5 | 48h8 |

**Operation**

Rt = first set bit number of (Prb)

**Exceptions:** none

**Execution Units:** ALUs

### PRLAST – Find Last Set Bit

**Description**

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

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 2019 | 18 15 | 14 12 | 11 7 | 6 0 |
| ~3 | Pr4 | 154 | ~2 | Prb4 | ~3 | Rt5 | 48h8 |

**Operation**

Rt = last set bit number of (Prb)

**Exceptions:** none

**Execution Units:** ALUs

### PRLSR – Predicate Logical Shift Right

**Description:**

Shift a predicate register to the left up to 31 bits.

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 20 15 | 14 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 24 | Amt6 | Pra4 | Prt4 | 48h8 |

1 clock cycle

**Exceptions:** none

### PROR – Predicate Or

**Description:**

Bitwise ‘or’ the contents of two predicate registers and place the result in a predicate register.

**Instruction Format: PR2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 2019 | 18 15 | 14 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 94 | ~2 | Prb4 | Pra4 | Prt4 | 48h8 |

**Exceptions:** none

### PRROL – Predicate Rotate Left

**Description:**

Rotate a predicate register to the left up to 63 bits.

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 20 15 | 14 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 14 | Amt6 | Pra4 | Prt4 | 48h8 |

1 clock cycle

**Exceptions:** none

### PRROR – Predicate Rotate Right

**Description:**

Rotate a predicate register to the Right up to 63 bits.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 20 15 | 14 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 34 | Amt6 | Pra4 | Prt4 | 48h8 |

1 clock cycle

**Exceptions:** none

### PRSUB – Predicate Subtract

**Description:**

Add the contents of two predicate registers and place the result in a predicate register.

**Instruction Format: PR2**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 29 | 28 25 | 24 21 | 2019 | 18 15 | 14 11 | 10 7 | 6 0 |
| ~3 | Pr4 | 54 | ~2 | Prb4 | Pra4 | Prt4 | 48h8 |

1 clock cycle

**Exceptions:** none

## Branch / Flow Control Instructions

### Overview

#### Mnemonics

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

#### 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 or by the comparison of a general purpose register and a postfixed immediate constant.

*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 five comparison types that may be performed as outlined in the table below.

|  |  |
| --- | --- |
| Cm3 | 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 32 or 48-bit opcodes.

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25 | Rb5 | Ra5 | Cm3 | Lk2 | 1 | 2xh8 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target9 | Rb5 | Ra5 | Cm3 | Lk2 | 0 | 2xh8 |

### Branch Conditions

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | ▼ |  |  | ▼ |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25 | Rb5 | Ra5 | Cm3 | Lk2 | 1 | 2xh7 |

|  |  |  |  |
| --- | --- | --- | --- |
| 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 three link registers in the architecture.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | ▼ |  |  |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25 | Rb5 | Ra5 | Cm3 | Lk2 | 0 | 2xh7 |

|  |  |
| --- | --- |
| Lk2 | Meaning |
| 0 | do not store return address |
| 1 | use Lk1 / Ca1 |
| 2 | use Lk2 / Ca2 |
| 3 | Use Lk3 / Ca3 |

### Branch Target

For conditional branches, the target address is formed as the sum of the instruction pointer and a constant specified in the instruction. Long branches are IP relative with a range of ±32MB. Short branches are IP relative with a range of ±512B. The displacement field is shifted left once before use.

*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 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25 | Rb5 | Ra5 | Cm3 | Lk2 | 1 | 2xh7 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ▼ |  |  |  |  | ▼ |  |
| 31 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target9 | Rb5 | Ra5 | Cm3 | Lk2 | 0 | 2xh7 |

### 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 |
| Target21 | Ca3 | Rb5 | Ra5 | Cm3 | Lk2 | 0 | 2xh8 |

### 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](#_Branch_Instructions).

**Formats Supported**: B

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Rb5 | Ra5 | 03 | Lk2 | 1 | 25h6 |

**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](#_Branch_Instructions).

**Formats Supported**: B

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Imm5 | Ra5 | 03 | Lk2 | 1 | 22h7 |

**Operation:**

Lk = next IP

If (Ra.bit[Imm7] == 1)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### BCC –Branch if Carry Clear

BCC Rm, Rn, label

**Description:**

Branch if the carry would be set when comparing the first source operand to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

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

**Instruction Format: B, BL**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Rb5 | Ra5 | 03 | Lk2 | 1 | 457 |

**Clock Cycles: 4**

### BCS –Branch if Carry Set

BCS Rm, Rn, label

**Description:**

This is an alternate mnemonic for the [BLO](#_BLO_–Branch_if) instruction. Branch if the carry would be set because of the comparison of the first operand to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

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

**Instruction Format: B, LB**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Rb5 | Ra5 | 03 | Lk2 | 1 | 447 |

**Clock Cycles: 4**

### BGE –Branch if Greater Than or Equal

BGE Rm, Rn, label

**Description:**

Branch if the first source operand is greater than or equal to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as signed integer values. The displacement is relative to the address of the branch instruction.

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

**Instruction Format: B**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Rb5 | Ra5 | 03 | Lk2 | 1 | 417 |

**Clock Cycles: 4**

### BGEU –Branch if Unsigned Greater Than or Equal

BGEU Rm, Rn, label

**Description:**

Branch if the first source operand is greater than or equal to the second. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as unsigned integer values. The displacement is relative to the address of the branch instruction.

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

**Instruction Format: B, LB**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Rb5 | Ra5 | 03 | Lk2 | 1 | 457 |

**Clock Cycles: 4**

### BEQ –Branch if Equal

BEQ Ra, Rb, label

**Description:**

Branch if two source operands are equal. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as integer values. The displacement is relative to the address of the branch instruction.

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

**Formats Supported**: B, LB

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Rb5 | Ra5 | 03 | Lk2 | 1 | 387 |

**Clock Cycles: 4**

### BNE –Branch if Not Equal

BNE Rm, Rn, label

**Description:**

Branch if two source operands are not equal. The first operand is in a register, the second in a register or an immediate value. Both operands are treated as integer values. The displacement is relative to the address of the branch instruction.

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

**Instruction Format: B, LB**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 47 23 | 22 18 | 17 13 | 12 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Rb5 | Ra5 | 03 | Lk2 | 1 | 397 |

**Clock Cycles: 4**

### BRA – Branch Always

**Description**:

This instruction always branches to the target address. The target address range is ±256GB.

**Formats Supported**: BSR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 23 | 22 10 | 9 8 | 7 | 6 0 |
| Target9..1 | Target22..10 | 02 | 0 | 20h7 |

**Formats Supported**: LBSR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 47 23 | 22 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Target38..26 | 02 | 1 | 20h7 |

**Operation:**

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### BSR – Branch to Subroutine

**Description**:

This instruction always jumps to the target address. The address of the next instruction is stored in a link register. The target address range is ±256GB.

**Formats Supported**: BSR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 23 | 22 10 | 9 8 | 7 | 6 0 |
| Target9..1 | Target22..10 | Lk2 | 0 | 20h7 |

**Formats Supported**: LBSR

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 47 23 | 22 10 | 9 8 | 7 | 6 0 |
| Target25..1 | Target38..26 | Lk2 | 1 | 20h7 |

**Operation:**

Lk = next IP

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### JMP – Jump to Target

**Description**:

This instruction always jumps to the target address. The target address is the sum of a register and an immediate constant shift left four times.

**Instruction Format:** JSR

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 17 | 16 12 | 11 9 | 8 7 | 6 0 |
| Fmt2 | Pr4 | Immediate15..7 | Ra5 | Im6..4 | 02 | 367 |

**Operation:**

Lk = next IP

IP = IP = sign extend (Constant)

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### JSR – Jump to Subroutine

**Description**:

This instruction always jumps to the target address. The target address is the sum of a register and an immediate constant shift left four times. The address of the next instruction is stored in a link register.

**Instruction Format:** JSR

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 17 | 16 12 | 11 9 | 8 7 | 6 0 |
| Fmt2 | Pr4 | Immediate15..7 | Ra5 | Im6..4 | Lk2 | 367 |

**Operation:**

Lk = next IP

IP = IP = sign extend (Constant)

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### NOP – No Operation

NOP

**Description:**

This instruction does not perform any operation.

**Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 8 | 7 | 6 0 |
| 0xFF8 | 0 | 1277 |

|  |  |  |
| --- | --- | --- |
| 31 8 | 7 | 6 0 |
| 0xFFFFFF8 | 1 | 1277 |

### RTD – Return from Subroutine and Deallocate

**Description**:

This instruction returns from a subroutine by transferring program execution to the address stored in a link register. Additionally, the stack pointer is incremented by the amount specified. The const field is shifted left four times before use.

**Formats Supported**: RTS

|  |  |  |  |
| --- | --- | --- | --- |
| 15 11 | 10 9 | 8 7 | 6 0 |
| Const5 | 22 | Lk2 | 357 |

**Operation:**

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

Return address prediction hardware may make use of the RTS instruction.

### RTE – Return from Exception

**Description**:

This instruction returns from an exception routine by transferring program execution to the address stored in an internal stack. The const field is shifted left once before use. This instruction may perform a two-up level return.

**Formats Supported**: RTS

|  |  |  |  |
| --- | --- | --- | --- |
| 15 11 | 10 9 | 8 7 | 6 0 |
| Const5 | 12 | 02 | 357 |

**Formats Supported**: RTS – Two up level return.

|  |  |  |  |
| --- | --- | --- | --- |
| 15 11 | 10 9 | 8 7 | 6 0 |
| Const5 | 12 | 12 | 357 |

**Operation:**

Optionally pop the status register and program counter from the internal stack. Add Const wydes to the program counter. If returning from an application trap the status register is not popped from the stack.

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

### 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 | 6 0 |
| Const5 | 02 | Lk2 | 357 |

**Operation:**

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

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.

**Instruction Format**: BRK

|  |  |
| --- | --- |
| 15 7 | 6 0 |
| 009 | 00h7 |

**Operation:**

PUSH SR

PUSH IP

IP = tvec[3]

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

### IRQ – Generate Interrupt

**Description:**

Generate interrupt. This instruction invokes the system exception handler. The return address is stored in the EIP register (code address register #8 to 15).

The return address stored is the address of the interrupt instruction, not the address of the next instruction. To call system routines use the [SYS](#_SYS_–Call_system) 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 19 | 18 7 | 6 0 |
| Lvl3 | Pr4 | ~6 | Cause12 | 1127 |

**Operation:**

PUSH SR

PUSH IP

CAUSE = Cause12

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 8 | 6 0 |
| 19 | 1147 |

**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 8 | 6 0 |
| 29 | 1147 |

**Clock Cycles:** 1

**Execution Units:** Memory

### 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 8 | 6 0 |
| 09 | 1147 |

### SYS – System Call

**Description**:

Perform a system call. Interrupts are disabled. The instruction pointer is reset to the contents of tvec[3] and instructions begin executing. There should be a jump instruction placed at the break vector location. The address of the SYS instruction is stored in the EIP register.

**Instruction Format**: BRK

|  |  |
| --- | --- |
| 15 7 | 6 0 |
| Callno9 | 07 |

**Operation:**

PUSH SR onto internal stack

PUSH IP onto internal stack

IP = tvec[3]

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## Modifiers

### ATOM

**Description:**

Treat the following sequence of instructions as an “atom”. The instruction sequence is executed with interrupts disabled. Interrupts are disabled for up to eight instructions.

Disable interrupts for the following Cnt3 instructions.

**Instruction Format:**

**Instruction Format**: ATOM

|  |  |  |
| --- | --- | --- |
| 15 10 | 9 7 | 6 0 |
| ~6 | Cnt3 | 1227 |

**Assembler Syntax:**

**Example:**

|  |
| --- |
| ATOM 6  LOAD a0,[a3]  CMP t0,a0,a1  PEQ t0,”TTF”  STORE a2,[a3]  LDI a0,1  LDI a0,0 |

|  |
| --- |
| ATOM 4  LOAD a1,[a3]  ADD t0,a0,a1  MOV a0,a1  STORE t0,[a3] |

### PRED

**Description:**

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

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

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

**Instruction Format:** REP

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 12 | 11 | 10 7 | 6 0 |
| Fmt2 | Pr4 | Immediate13..0 | ~ | Pr4 | 1217 |

**Assembler Syntax:**

After the instruction mnemonic the register containing the predicate flags is specified. Next a character string containing ‘T’ for True, ‘F’ for false, or ‘I’ for ignore for the next seven instructions is present.

**Example:**

|  |
| --- |
| PRED r2,”TTTFFFI” ; next three execute if true, three after execute if false, one after always execute  MUL r3,r4,r5 ; executes if True  ADD r6,r3,r7 ; executes if True  ADD r6,r6,#1234 ; executes if True  DIV r3,r4,r5 ; executes if FALSE  ADD r6,r2,r1 ; executes if FALSE  ADD r6,r6,#456 ; executes if FALSE  MUL r8,r9,r10 ; always executes |

### REP

**Description:**

This modifier indicates a short series of instructions to repeat while the loop counter condition is met. The repeat modifier includes instructions according to a count specified in the Icnt3 field. The number of included instructions is one greater than Icnt3. Up to eight instructions may be part of the repeat operation. The loop counter may be incremented or decremented for each repeat. Loop counter tests perform signed comparisons. The 12-bit immediate may be overridden with a constant postfix instruction. The constant postfix does not count as an instruction in the loop.

REP is limited to a 32-bit immediate value.

Context for the REP instruction is stored in a context buffer which must be saved and restored when the context changes or during interrupt processing.

**Instruction Format:** REP

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 14 | 13 | 12 10 | 9 7 | 6 0 |
| Fmt2 | Pr4 | Immediate11..0 | D | Icnt3 | Cnd3 | 1207 |

|  |  |
| --- | --- |
| D | Meaning |
| 0 | Decrement loop counter |
| 1 | Increment loop counter |

|  |  |  |
| --- | --- | --- |
| Cnd3 | Loop Counter Test |  |
| 0h | Equal | LC == Imm |
| 1h | Not equal | LC != Imm |
| 2h | Signed less than | LC < Imm |
| 3h | Signed less than or equal | LC <= Imm |
| 4h | Signed greater than or equal | LC >= Imm |
| 5h | Signed greater than | LC > Imm |
| 6h | Bit clear | LC[imm] = 0 |
| 7h | Bit set | LC[imm] == 1 |

**Assembler Syntax:**

# 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 | UN  LINK | 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 |  |  |  |

## Thor2024 Root Opcode

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0x** | 0  BRK / SYS | 1  {R1} | 2  {R2A} | 3  {R2M} | 4  ADDI | 5  SUBFI | 6  MULI | 7 |
|  | 8  ANDI | 9  ORI | 10  EORI | 11  CMPI | 12 | 13  DIVI | 14  MULUI | 15  CSR |
| **1x** | 16  ADDIQ | 17  ASLIQ | 18  {R2V} | 19  {R2C} | 20  {R2S} | 21  DIVUI | 22  {R2P} | 23 |
|  | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| **2x** | 32  BSR / BRA | 33  DBRA | 34  BBCI / BBSI | 35  RTx | 36  JSR | 37 | 38  BEQ | 39  BNE |
|  | 40  BLT | 41  BGE | 42  BLE | 43  BGT | 44  BLTU | 45  BGEU | 46  BLEU | 47  BGTU |
| **3x** | 48  {PR} | 49  PRFILL | 50  CHKI | 51 | 52 | 53  BLEND | 54  PUSH | 55  BFND |
|  | 56  FLDH | 57  FLDS | 58  FLDD | 59  FLDQ | 60  FSTH | 61  FSTS | 62  FSTD | 63  FSTQ |
| **4x** | 64  LDB | 65  LDBU | 66  LDW | 67  LDWU | 68  LDT | 69  LDTU | 70  LDO | 71  LDOU |
|  | 72  LDH | 73  LDG | 74 | 75  CACHE | 76  PLDS | 77  PLDD | 78  DFLD | 79  {LDX} |
| **5x** | 80  STB | 81  STW | 82  STT | 83  STO | 84  STH | 85  STG | 86  STPTR | 87  {STX} |
|  | 88  ASL ASLI | 89  LSR LSRI | 90  ASR ASRI | 91  ROL ROLI | 92  ROR RORI | 93  DFST | 94  PSTS | 95  PSTD |
| **6x** | 96  ADDIPC | 97  {FLT1} | 98  {FLT2} | 99  {FLT3} | 100 MUX  PTRDIF | 101  {DFLT1} | 102  {DFLT2} | 103  {DFLT3} |
|  | 104  {BITFLD} | 105  {PST1} | 106  {PST2} | 107  {PST3} | 108 | 109  BCMP | 110  ENTER | 111  LEAVE |
| **7x** | 112  IRQ | 113  STOP | 114  SYNC | 1115  WFI / PFI | 116  POP | 117  CPUID | 118  MFLK | 119  MTLK |
|  | 120  REP | 121  PRED | 122  ATOM | 123  PFX0 | 124  PFX1 | 125  PFX3 | 126  PFX3 | 127  NOP |

### {R1} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0  CNTLZ | 1  CNTLO | 2  CNTPOP | 3  ABS | 4  SQRT | 5  REVBIT | 6 | 7 |
|  | 8 | 9 | 10 | 11 | 12 | 13 | 14  SM3P0 | 15  SM3P1 |
| 1x | 16 | 17 | 18  AES64DS | 19  AES64DSM | 20  AES64ES | 21  AES64ESM | 22  AES64IM | 23 |
|  | 24  SHA256  SIG0 | 25  SHA256  SIG1 | 26  SHA256  SUM0 | 27  SHA256  SUM1 | 28  SHA512  SIG0 | 29  SHA512  SIG1 | 30  SHA512  SUM0 | 31  SHA512  SUM1 |

### {R2A} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 0  AND | 1  OR | 2  EOR | 3  CMP | 4  ADD | 5  SUB | 6  MUL | 7  DIV |

### {R2P} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 22 | 0  MULU | 1  DIVU | 2  MODU | 3  MULSU | 4  DIVSU | 5  MODSU | 6  MULH | 7  MULUH |

### {R2M} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 3 | 0  MIN | 1  MAX | 2  BMM | 3  BMAP | 4  DIF | 5  CHARNDX | 6  CHARNDX | 7  CHARNDX |

### {R2V} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 18 | 0  V2BITS | 1  BITS2V | 2  VEX | 3  VEINS | 4  VGNDX | 5 | 6  V2BITS | 7 |

### {R2C} Crypto Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 19 | 0  AES64K1I | 1  AES64KS2 | 2  SM4ED | 3  SM4KS | 4 | 5 | 6  CLMUL | 7 |

### {R2S} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 20 | 0  SEQ | 1  SNE | 2  SLT | 3  SLE | 4  SLTU | 5  SLEU | 6  SGT | 7  SGE |

### {FLT2} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0  FSCALEB | 1 | 2  FMIN | 3  FMAX | 4  FADD | 5  FSUB | 6  FMUL | 7  FDIV |
|  | 8  FSEQ | 9  FSNE | 10  FSLT | 11  FSLE | 12 | 13  FCMP | 14  FNXT | 15  FREM |
| 1x | 16 | 17 | 18  FMINI | 19  FMAXI | 20  FADDI | 21  FSUBI | 22  FMULI | 23  FDIVI |
|  | 24  FSEQI | 25  FNSEI | 26  FSLTI | 27  FSLEI | 28 | 29  FCMPI | 30 | 31  FREMI |

### {FLT1} Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0  FABS | 1  FNEG | 2  FOTI | 3  ITOF | 4 | 5 | 6  FSIGN | 7  FSIG |
|  | 8  FSQRT | 9  FS2D | 10  FS2Q | 11  FD2Q | 12 | 13 | 14  ISNAN | 15  FINITE |
| 1x | 16 | 17 | 18 | 19 | 20 | 21  FTRUNC | 22 | 23  FRES |
|  | 24 | 25  FD2S | 26  FQ2S | 27  FQ2D | 28 | 29 | 30  FCLASS | 31 |

**Fn5 – Indexed Loads**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0x** | 0  LDBX | 1  LDBUX | 2  LDWX | 3  LDWUX | 4  LDTX | 5  LDTUX | 6  LDOX | 7  LDOUX |
|  | 8  LDHX | 9  LDGX | 10 | 11  CACHEX | 12  PLDSX | 13  PLDDX | 14 | 15 |
| **1x** | 16 | 17 | 18  FLDHX | 19 | 20  FLDSX | 21 | 22  FLDDX | 23 |
|  | 24  FLDQX | 25  DFLDX | 26 | 27 | 28  BFNDX | 29 | 30 | 31 |

**Fn5 – Indexed Stores**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0x** | 0  STBX | 1  STWX | 2  STTX | 3  STOX | 4  STHX | 5  STGX | 6  STPTRX | 7  PUSH |
|  | 8 | 9  FSTHX | 10  FSTSX | 11  FSTDX | 12  FSTQX | 13 | 14  PSTSX | 15  PSTDX |
| **1x** | 16 | 17 | 18 | 19 | 20  DFSTX | 21 | 22  PUSH | 23 |
|  | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

## Thor2024 Predicate Operations

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0x** | 0  PRASL | 1  PRROL | 2  PRLSR | 3  PRROR | 4  ADD | 5  SUB | 6 | 7 |
|  | 8  PRAND | 9  PROR | 10  PREOR | 11  MFPR | 12  MTPR | 13  PRCNTPOP | 14  PRFIRST | 15  PRLAST |
| **1x** | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
|  | 24 | 25 | 26 | 27 | 28  PSASL | 29  PRROL | 30  PRLSR | 31  PRROR |