|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 39 24 | 23 21 | 20 15 | | 14 9 | | 8 | 7 0 |  |
|  |  | | 23 17 | | 16 13 | | 12 9 | 8 | 7 0 |  |
|  |  | | Constant7 | | Ra4 | | Rt4 | v | Opcode8 | LDBU |
|  |  | Constant19 | | Ra6 | | Rt6 | | v | Opcode8 |  |

## General Registers

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

Register #0 is always zero.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | |
| r0 | always zero |  |  | LC | Loop Counter | |
| r1 | return value |  |  |  |  | |
| r2 | return value |  |  | C0 | available | |
| r3 | temporary register caller save |  |  | C1 | return address | |
| r4 | temporary register |  |  | C2 | milli-code return address | |
| r5 | temporary register |  |  | C3 | available | |
| r6 | temporary register |  |  | C4 | available | |
| r7 | temporary register |  |  | C5 | available | |
| r8 | temporary register |  |  | C6 | exceptioned IP | |
| r9 | temporary register |  |  | C7 | Instruction pointer, read only | |
| r10 | temporary register |  |  |  |  | |
| r11 | register var callee save |  |  |  |  | |
| r12 | register var |  |  | ZS |  |  |
| r13 | register var |  |  | DS |  |  |
| r14 | register var |  |  | ES |  |  |
| r15 | register var |  |  | FS |  |  |
| r16 | register var |  |  | GS |  |  |
| r17 | register var |  |  | HS |  |  |
| r18 | register var |  |  | SS |  |  |
| r19 |  |  |  | CS |  |  |
| r20 |  |  |  |  |  | |
| r21 |  |  |  |  |  | |
| r22 |  |  |  |  |  | |
| r24 | Type number |  |  |  |  | |
| r25 | Class Pointer |  |  |  |  | |
| r26 | Base Pointer |  |  |  |  | |
| r27 | User Stack Pointer1 |  |  |  |  | |
| r28 | Interrupt Stack Pointer |  |  |  |  | |
| r29 | Exception Stack Pointer |  |  | DBAD0 | Debug Address #0 | |
| r30 | Debug Stack Pointer |  |  | DBAD1 | Debug address #1 | |
| r31 | Kernel task register |  |  | DBAD2 | Debug address #2 | |
| r32/F0 | Floating point |  |  | DBAD3 | Debug Address #3 | |
| … |  |  |  | DBCTRL | Debug Control | |
| r63/F31 |  |  |  | DBSTAT | Debug Status | |

1 this register is implied in the push and rts instructions, and updated by hardware

r27 is special in that it refers to one of r27, r28, r29, or r30 depending on the operating mode of the core. This allows the same code to be reused in different operating modes. For instance loading r27 while in debug mode will actually load r30 and all references to r27 will be rerouted to r30 in debug mode.

## Loop Count

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

## Code Address Registers

Thor2021 has eight code address registers C0 to C7. These are also referred to as branch registers in other architectures. C7 refers to the current instruction pointer.

### Code Address Register Format

A code address register is composed of a 64-bit offset field and a 32-bit selector field. It is necessary to store both the selector and offset in a linkage register so that a far return may be performed.

A branch instruction will set both the selector and offset values for the instruction pointer. The new selector value for the IP will come from one of the other code address registers as specified in the instruction.

|  |  |
| --- | --- |
| 95 64 | 63 0 |
| Selector32 | Offset64 |

The selector value of a code address register may be set using the MTSPR instruction and selecting the code address selector as the target.

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

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

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

### Instruction Pointer

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

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

|  |  |
| --- | --- |
| 63 24 | 23 0 |
| IP High | IP Low |

### Link Registers

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

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

|  |  |
| --- | --- |
|  | 63 0 |
| Lk1 | Return Address |
| Lk2 | Return Address |

## Selector Registers

Selector registers are a piece of the segmented memory management. They are a short form for segment descriptors which they represent. There are eight selector registers in the architecture.

## Summary of Special Purpose Registers

### C0,C1,…C7 (CREGS) – SPR #16 to 31

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

|  |  |
| --- | --- |
| Reg # |  |
| 16 | C0 low |
| 17 | C0 high |
| … |  |
| 30 | C7 low |
| 31 | C7 high |

### ZS,DS,ES,FS,GS,HS,SS,CS (SREGS) – SPR #32 to 39

These registers reflect the values of the segment selectors currently active in the core. Writing to a selector register triggers a load of the corresponding descriptor cache.

### LDT – SPR #40

The LDT register holds the selector for the local descriptor table.

|  |  |  |
| --- | --- | --- |
| 31 24 | 23 | 22 0 |
| PL8 | T | Index23 |

### GDT – SPR #41

The GDT register holds the location and size of the global descriptor table. The descriptor table must be 64-byte aligned. The low order six bits of the GDT register are used to indicate the table size.

|  |  |
| --- | --- |
| 63 6 | 5 0 |
| Table Address63..6 | Size |

Since the table is software managed the exact meaning of the size field is up to the software implementation. It is suggested that the size field at least be a multiple of the memory page size so that the memory protection capability can be applied to the table.

Note that the global descriptor table must be located in the low 64-bits of the physical address space.

### KEYS – SPR #48,#49

This pair of registers contains the collection of keys associated with the process for the memory lot system. Each key is twenty bits in size. The pair of registers contains six keys.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 59 40 | 39 20 | 19 0 |
| ~4 | key2 | key1 | key0 |

### TICK - SPR #50

The tick count register contains the number of clock cycles that have passed since the last reset.

### VL – SPR #54

The VL register stores the vector length for vector instructions. The length cannot exceed the length of vectors supported in hardware. Only the low order 16-bits of this register are implemented.

### TVEC – SPR #64 to #71

This set of register pairs contains the trap vectors for each operating mode. TVEC[3] is used by hardware to vector to the exception processing routine. TVEC[0] to TVEC[2] are used by the REX instruction to redirect exceptions. The low half of the register contains the offset of the exception processing routine. The high half of the register contains the selector value of the exception processing routine.

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

|  |  |
| --- | --- |
| Reg # |  |
| 64 | TVEC[0] low |
| 65 | TVEC[0] high |
| … |  |
| 70 | TVEC[3] low |
| 71 | TVEC[3] high |

# 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.

# Segmentation

## Overview

Segmentation is a low overhead means of memory protection and virtualization. Providing separate protected address spaces for different applications is the job of the operating system. Ideally segmentation hardware should not be visible to the application. The application should appear as though it has a flat memory model. The core contains eight segment registers. The segmentation system is managed via a combination of hardware and software. Up to 256 privilege levels are available.

## Privilege levels

Memory access is available according to privilege levels. The segmentation system allows up to 256 privilege levels.

## Usage

The segment register to use during address formation for data addresses is identified by a field in the instruction. This field is set to default values by the assembler. For code addresses segment register #7 (the CS) is always used.

* If segmentation is not desired then segmentation can effectively be ignored by setting all the segment registers to zero. The processor can also be built without segmentation by commenting out the ‘SEGMENTATION’ definition.

## Software Support

Segmentation is software supported. A software implementation allows a high degree of flexibility when implementing the segmentation model. Loading a value into a selector register causes a software segmentation exception to occur. The exception routine then loads the segment base, limit and access rights from a table in memory. It’s up to the system level software to determine if protection rules are violated.

Segment registers may only be transferred to or from one of the general-purpose registers. The [mtspr](#_MTSPR_–Register-Special_Register) and [mfspr](#_MFSPR_–_Special) instructions can be used to perform the move. A segment register may also be loaded using the [LDIS](#_LDIS_-_Load-Immediate) instruction. After loading a segment register the instruction stream should be synchronized with a memory barrier ([MEMSB](#_MEMSB_–_Memory)) to ensure the segment value can be ready for a following memory operation.

There are two vectors in the vector table reserved for implementing far subroutine call and return instructions.

## Address Formation:

Non-segmented address bits 0 to 11 pass through the segmentation module unchanged. Address bits 63 to 12 are added to the contents of the segment register to form the final segmented address. Note that there is no shift associated with the segment addition. Future implementations of the processor may include additional low order address bits in the segment register to allow a finer grain for memory page / paragraph size.

|  |  |
| --- | --- |
| Address[63:12] | Address[11:0] |
| + | + |
| Segment register value[63:12] | 00012 |
| = | |
| Segmented address[63:0] | |

## Selecting a segment register

A specific segment register for a memory operation may be selected using a segment prefix in assembler code. Segment prefixes apply to data addresses only. Code addresses always use segment register #7 – the code segment. The segment prefix indicator is encoded by a three-bit field in the instruction.

## Selectors

The core uses selectors as a more compact way to represent segment registers. Rather than pass the entire segment descriptor to routines (256 bits) and have each routine check for privilege violations, the core uses 32-bit selectors. Privilege violations are checked for at the time the segment register components (base, limit and access rights) are loaded into the descriptor cache. The selector includes a field identifying the privilege level, and a second field identifying which segment descriptor the selector is associated with. The selector format is shown below.

### Selector Format:

|  |  |  |
| --- | --- | --- |
| 31 24 | 23 | 22 0 |
| PL8 | T | Index23 |

PL8: the privilege level associated with the segment

Index23: the index into the descriptor table

T: 0 = global, 1 = local descriptor table

## Descriptor Cache

If every memory access had to incur another memory access to load descriptor information processing speed would be adversely affected. To avoid additional memory access descriptors selected by selectors are cached in a descriptor cache for the selector register. The descriptor cache is only loaded when the value in the selector register is updated. Moving a value to a selector register with the MTSEL instruction causes the descriptor cache to be loaded from memory.

## Non-Segmented Code Area

The address range defined as 64’hFxxxxxxxxxxxxxxx (the top nibble is ‘F’) is a non-segmented code area. This area allows the operating system to work without paying attention to the code segment. Interrupt and exception vectors should vector into the non-segmented code area. The only way to change the code segment is by transferring to the operating system via a sys call instruction.

## Changing the Code Segment

The only way to change the code segment is by transferring to the operating system via a sys call instruction. The operating system, while operating in the non-segmented code area, can alter the code segment without causing a transfer of control. The operating system establishes the code segment for a task while running in the non-segmented code area. To support far subroutine calls and returns there are vectors in the vector table that allow implementation of a far call or return.

## The Descriptor Table

The descriptor table is a software managed table that contains information on the location and size for segments in the form of memory descriptors. Each descriptor is 32 bytes in size. Memory descriptor entries in the table have the following format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 255 244 | 243 192 | 191 128 | 127 64 | 63 0 |
| w0 | ACR16 | ~48 | Limit64 | ~64 | Base64 |
| w1 | ACR16 | ~48 | Limit64 | ~64 | Base64 |
| … |  |  |  |  |  |

The descriptor table may contain other types of descriptors beyond basic memory descriptors, such as call gates.

The base address of, and the number of entries in the descriptor table is contained in the LDT or GDT special purpose registers. The descriptor table may be updated with regular load and store instructions when the processor is at privilege level zero.

32-bit selectors are used to index into the table in order to determine the characteristics of the segment.

#### Memory Descriptors

Memory descriptors describe the location and size of memory segments. They have the following format:

|  |  |  |
| --- | --- | --- |
| n+3 | ACR16 | ~48 |
| n+2 | Limit63..0 | |
| n+1 | ~64 | |
| n | Base63..0 | |

#### The Access Rights Field (ACR16) – Memory Descriptor

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 |  |  | 12 | 11 | 10 | 9 | 8 | 7 |  |  | 0 |
| P | Stk | C | 1/S | R | W | X | A | DPL8 | | | |

P: 1 = segment present, 0 = segment not present

S: 0 = system descriptor, 1 = memory descriptor

C: 1 = conforming code segment

R: 1 = readable

W: 1 = writeable

X: 1 = executable, 0 = data

A: 1= accessed

DPL8 = descriptor privilege level

#### Typical Values for ACR

9A00 – executable, readable code segment, privilege level zero

9C00 – read/writeable data segment, privilege level zero

CC00 – read / writeable stack segment, privilege level zero

#### Stack Segment Descriptors

Stack segment descriptors describe the location and limits of stack segments. They have the following format:

|  |  |  |
| --- | --- | --- |
| ~48 | | ACR16 |
| ~64 | | |
| Upper Limit34..3 | Lower Limit34..3 | |
| Base63..0 | | |

There is no difference between a stack segment descriptor and a memory segment descriptor except in the way that the segment limit field is used. (Bit 10 of the ACR for the data descriptor is set). For a stack segment, when the descriptor is loaded, the limit field is split in two in order to provide both an upper and lower bounds to the stack. If either bounds are exceeded a stack fault occurs rather than a bounds violation. This provides the capacity to expand the stack. One limitation of this mechanism is that the stack is limited to 35 address bits (32GB). Note that the stack is always word aligned so the upper and lower limits represent word boundaries.

### System Segment Descriptors

System descriptors are identified by having bit12 of the access rights character set to zero. There are potentially sixteen different system descriptor types.

#### The Access Rights Field (ACR16) – System Descriptor

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 |  |  | 12 | 11 | 10 | 9 | 8 | 7 |  |  | 0 |
| P | ~ | ~ | 0 | Type4 | | | | DPL8 | | | |

|  |  |  |
| --- | --- | --- |
| Type4 | Gate |  |
| 0 | unused |  |
| 2 | LDT descriptor |  |
| 4 | Call gate |  |
| 5 | Task Gate |  |
| 6 | Interrupt Gate |  |
| 7 | Trap gate |  |

#### LDT Descriptor

The LDT descriptor establishes the location and size of the local descriptor table in memory.

|  |  |  |  |
| --- | --- | --- | --- |
| n+3 | ~48 | | ACR16 |
| n+2 | ~41 | Size22..0 | |
| n+1 |  | | |
| n | Base63..0 | | |

#### Call Gate Descriptor

|  |  |  |  |
| --- | --- | --- | --- |
| ~48 | | | ACR16 |
| ~64 | | | |
| ~27 | N5 | Selector31..0 | |
| Base63..0 | | | |

## Segment Load Exception

Moving a value to a selector register (a move to SPR #32 to 38,40) triggers a segment load exception to allow the segment descriptor to be loaded from one of the descriptor tables. This exception is triggered for a LDIS or MTSPR instruction. There is a separate exception vector (vectors #256 to 264) to handle each segment register. The selector value being loaded into the segment register is reflected in the ARG1 special purpose register.

## Segment Bounds Exception

If an address is greater than or equal to the limit specified in the segment limit register then a segment limit exception occurs. This applies for all segments including code and data segments.

## Segment Usage Conventions

Segment register #7 is the code segment (CS) register. All program counter addresses are formed with the code segment register unless the upper nibble of the address is ‘F’ in which case the code segment is ignored.

Segment register #6 is the stack segment (SS) register by convention. Future versions of the core may use this register implicitly for stack accesses. The assembler automatically selects the stack segment when one of the stack pointer registers is specified in the instruction. Segment register #1 is the data segment (DS) by convention. The data segment is selected as the segment register for memory operations when the stack segment is not selected.

## Power-up State

On reset the value in the segment registers are undefined. Note that the processor begins executing instructions out of the non-segmented code area as the reset address is 64’hFFFFFFFFFFFC0000. One of the first tasks of the boot program would be to initialize the segment registers to known values. The segment register must be setup to perform data accesses properly.

### Segment Registers

|  |  |  |  |
| --- | --- | --- | --- |
| Num |  | Long name | Comment |
| 0 | ZS | zero (NULL) segment | by convention contains zero |
| 1 | DS | data segment | by convention – default for loads/stores |
| 2 | ES | extra segment | by convention |
| 3 | FS |  |  |
| 4 | GS |  |  |
| 5 | HS |  |  |
| 6 | SS | Stack segment | default for stack load/stores |
| 7 | CS | Code segment | always used for code addressing |

# Instruction Set Description

## Overview

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

## Root Opcode

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  | ▼ |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Constant11 | Ra6 | Rt6 | v | Opcode8 |

## Vector Instruction Indicator

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  | ▼ |  |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Constant11 | Ra6 | Rt6 | v | Opcode8 |

## Target Register Spec

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | ▼ |  |  |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Constant21 | Ra6 | Rt6 | V | Opcode8 |

## Register Formats

### R1 (one source register)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 24 22 | 21 | 20 15 | 14 9 | 8 | 7 0 |
| Func7 | m3 | z | Ra6 | Rt6 | v | Opcode8 |

### R1R (one source register)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 28 | 27 25 | 24 22 | 21 | 20 15 | 14 9 | 8 | 7 0 |
| Func7 | ~5 | Rm3 | m3 | z | Ra6 | Rt6 | v | Opcode8 |

### R2 (two source register)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Func7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | Opcode8 |

### R2R (two source register)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 36 | 35 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Func7 | ~5 | Rm3 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | Opcode8 |

### R3 (three source register)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 38 | 37 | 3635 | 34 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Func7 | m3 | z | Tc2 | Rc6 | Tb2 | Rb6 | Ra6 | Rt6 | v | Opcode8 |

### R3R (three source register)

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 49 | 48 44 | 43 41 | 40 38 | 37 | 3635 | 34 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Func7 | ~5 | Rm3 | m3 | z | Tc2 | Rc6 | Tb2 | Rb6 | Ra6 | Rt6 | v | Opcode8 |

## Branch Instructions

Branches branch to the target address only if the condition is true. The condition is determined by the comparison of two general-purpose registers. The Rb register field may also specify a quick immediate value.

*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.*

### Branch Format

There are two branch formats, short and long, differing only in the number of bits used for the target address constant. The remainder of the branch instruction functions identically.

In many cases the branch target is located a short distance away from the branch. It would be wasteful to use a long format for all cases. However, calling subroutines typically requires more address bits than is available in the short format. Hence, a longer format branch instruction is also available.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2xh8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3xh8 |

### Branch Conditions

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | ▼ | |  | |  | | ▼ | |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | | 8 | | 7 0 | |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | | 0 | | 3xh8 | |

|  |  |
| --- | --- |
| 2x/3x | Comparison Test |
| 28h | signed less than |
| 29h | signed greater or equal |
| 2Ah | signed less than or equal |
| 2Bh | signed greater than |
| 2Ch | unsigned less than |
| 2Dh | unsigned greater or equal |
| 2Eh | unsigned less than or equal |
| 2Fh | unsigned greater than |
| 26h | equal |
| 27h | not equal |
| 24h | bit clear |
| 25h | bit set |

|  |  |
| --- | --- |
| Cn2 |  |
| 0 | ignore loop counter, branch if condition is true |
| 1 | decrement loop counter, branch if loop counter is non-zero and condition is true |
| 2 | reserved |
| 3 | reserved |

### Linkage

Branches may specify a linkage register which is updated with the address of the next instruction. This allows subroutines to be called. There are two link registers in the architecture.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  | ▼ |  |  |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2xh8 |

|  |  |
| --- | --- |
| Lk2 | Meaning |
| 0 | do not store return address |
| 1 | use Lk1 |
| 2 | use Lk2 |
| 3 | reserved (C3) |

### Branch Target

For the short format branches, the target address is formed as the sum of a code address register and a 10-bit constant specified in the instruction. Branches may be IP relative with a range of ±512B.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ▼ | ▼ |  |  |  | ▼ |  |  |  |  |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2xh8 |

For the long format branches, the target address is formed as the sum of a code address register and a 26-bit constant specified in the instruction. Branches may be IP relative with a range of ±32MB. To perform a far branch operation, specify a code address register containing the target selector value as part of the target address.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ▼ | ▼ |  |  |  | ▼ |  |  |  |  |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3xh8 |

### Near or Far Branching

In a segmented system a question that comes to mind is how to perform a far jump or call versus performing a near one. For Thor2021 all branches are effectively far branches as the target selector is always loaded into the IP register. The return address selector is always stored in a link register. However, if addresses are formed using IP relative addressing, Ca3 = 7, then the selector value will remain unchanged making the branch effectively a near branch. If the Ca3 field is specified as a zero, then the IP selector value will remain unchanged. A non-zero Ca3 register is needed to perform a far branch.

### 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.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ▼ | ▼ |  |  |  | ▼ |  |  |  |  |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2xh8 |

## Load / Store Instructions

### Addressing Modes

There are three address modes associated with load and store instructions. Register indirect with displacement, register indirect with long displacement and scaled indexed addressing. Load and store instructions specify a segment register to use with the instruction. The assembler will provide a default value for this field that is suitable in most cases. The default can be overridden using a segment prefix indicator.

#### 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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3129 | 28 21 | 20 15 | 14 9 | 8 | 7 0 |
| Seg3 | Displacement7..0 | Rt6 | Ra6 | v | Opcode8 |

#### Register Indirect with Long Displacement Format

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4745 | 44 21 | 20 15 | 14 9 | 8 | 7 0 |
| Seg3 | Displacement23..0 | Rt6 | Ra6 | v | Opcode8 |

#### Scaled Indexed Format

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Func7 | Seg3 | Sc2 | Tb2 | Rb6 | Rt6 | Ra6 | v | Opcode8 |

## Arithmetic / Logical / Shift

### ADD - Register-Register

**Description:**

Add two registers and place the sum in the target register. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The mask register is ignored for scalar instructions.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 04h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + Rb

**Exceptions:**

**Notes:**

### ADDI - Register-Immediate

**Description:**

Add a register and a sign extended immediate value and place the sum in the target register. For the vector instruction both Ra and Rt are vector registers. If the Ra register field is zero then the value zero is used for Ra.

**Instruction Format: RIL**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Ra6 | Rt6 | v | 48h8 |

**Instruction Format: RI**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 04h8 |

|  |  |  |  |
| --- | --- | --- | --- |
| 23 15 | 14 9 | 8 | 7 0 |
| Immediate9..0 | Rt6 | v | 48h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:**

**Notes:**

*Quite often when incrementing indexes for arrays in loops the source and target registers are the same register. The constant increment is often a small number. This led to the 10-bit immediate format instruction. An even shorter instruction with a two-bit immediate was considered and discarded.*

*For the large immediate instruction, a size of 35 bits for the immediate was chosen as this allows a 64-bit constant to be built in a register using only two instructions.*

### ADDIS - Add Immediate Shifted

**Description:**

Add a register and immediate value and place the sum in the target register. The immediate value is shifted to the left by 32 bits before the addition takes place. This instruction combined with a 35-bit immediate ADDI instruction may be used to build a 64-bit constant.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Rt6 | Ra6 | v | 48h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + (immediate << 32)

**Exceptions:**

**Notes:**

### AND – Bitwise And

**Description**:

Perform a bitwise ‘and’ operation between operands.

**Integer Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 08h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

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

**Exceptions**: none

### OR – Bitwise Or

**Description**:

Perform a bitwise ‘or’ operation between operands.

**Integer Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 09h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

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

**Exceptions**: none

### ORI – Bitwise Or Immediate

**Description**:

Perform a bitwise or operation between operands. The immediate constant is zero extended before use.

**Instruction Format: RIL**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Ra6 | Rt6 | v | 48h8 |

**Instruction Format: RI**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 09h8 |

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

**Integer Instruction Format: RM**

Generates a mask consisting of ‘1’s from the mask begin, Mb6, for a width determined by Mw6, inclusive. Bitwise ‘or’ the mask with the value in register Ra and store the result in register Rt. The result may be optionally sign extended from bit me6 to the width of the machine. This instruction is useful for inserting or setting bit fields.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 9h4 | S | ~2 | Mw6 | Mb6 | Ra6 | Rt6 | v | AAh7 |

**Operation**

**Rt = Ra | Immediate**

**Vector Operation**

for x = 0 to VL-1

if (Vm0[x]) Vt[x] = Va[x] | Immediate

else Vt[x] = Vt[x]

**Exceptions**: none

### ORIS - Or Immediate Shifted

**Description:**

Bitwise ‘or’ a register and immediate value and place the sum in the target register. The immediate value is shifted to the left by 32 bits before the addition takes place. This instruction combined with a 35-bit immediate ORI instruction may be used to build a 64-bit constant.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Rt6 | Ra6 | v | 48h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra | (immediate << 32)

**Exceptions:** integer overflow

**Notes:**

### SUBF – Subtract From

**Description:**

Subtract Ra from Rb and place the difference in the target register Rt. If the instruction is a vector addition then Ra and Rt are vector registers. Rb may be either a vector or a scalar register. The mask register is ignored for scalar instructions.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 05h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + Rb

**Exceptions:**

**Notes:**

### SUBFI – Subtract from Immediate

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 05h8 |

**Clock Cycles:** 1

**Execution Units:** AllALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:**

**Notes:**

*Unlike the ADDI instruction there is only a single form for this instruction. SUBFI is used far less often than ADDI.*

## Load / Store Instructions

### LDB – Load Byte

**Description:**

An eight-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn’t in the cache provided the current memory page is cacheable.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3129 | 28 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement7..0 | Rt6 | Ra6 | v | 80h8 |

**Clock Cycles:** 10 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+offset])

**Exceptions:** DBE, DBG, LMT, TLB

### LDBL – Load Byte, Long Address

**Description:**

An eight-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn’t in the cache provided the current memory page is cacheable.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4745 | 44 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement23..0 | Rt6 | Ra6 | v | D0h8 |

**Clock Cycles:** 10 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+offset])

**Exceptions:** DBE, DBG, LMT, TLB

### LDBU – Load Byte, Unsigned

**Description:**

An eight-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn’t in the cache provided the current memory page is cacheable.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3129 | 28 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement7..0 | Rt6 | Ra6 | v | 81h8 |

**Clock Cycles:** 10 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+offset])

**Exceptions:** DBE, DBG, LMT, TLB

### LDBUL – Load Byte Unsigned, Long Address

**Description:**

An eight-bit value is loaded from memory and zero extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn’t in the cache provided the current memory page is cacheable.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4745 | 44 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement23..0 | Rt6 | Ra6 | v | D1h8 |

**Clock Cycles:** 10 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+offset])

**Exceptions:** DBE, DBG, LMT, TLB

### LDBX – Load Byte Indexed

**Description:**

An eight-bit value is loaded from memory sign extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 00h7 | Seg3 | Sc2 | Tb2 | Rb6 | Ra6 | Rt6 | v | B0h8 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+Rb])

**Exceptions:** DBE, DBG, LMT, TLB

### LDBUX – Load Byte Unsigned Indexed

**Description:**

An eight-bit value is loaded from memory zero extended and placed in the target register. The memory address is the sum of register Ra and scaled register Rb.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 01h7 | Seg3 | Sc2 | Tb2 | Rb6 | Ra6 | Rt6 | v | B0h8 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (mem[Ra+Rb])

**Exceptions:** DBE, DBG, LMT, TLB

### STB – Store Byte

**Description:**

An eight-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3129 | 28 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement7..0 | Rt6 | Ra6 | v | 90h8 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+offset] = Rb[7..0]

**Exceptions**: DBE, DBG, TLB, LMT

### STBL – Store Byte, Long Addressing

**Description:**

An eight-bit value is stored to memory from the source register Rb. The memory address is the sum of the sign extended displacement and register Ra.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 47 45 | 44 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement23..0 | Rt6 | Ra6 | v | E0h8 |

**Clock Cycles:** 3 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

memory[Ra+offset] = Rb[7..0]

**Exceptions**: DBE, DBG, TLB, LMT

## Branch / Flow Control Instructions

### BEQ – Branch if Equal

**Description**:

This instruction branches to the target address if the contents of the Ra equals the contents of Rb, otherwise program execution continues with the next instruction. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: BR,BRL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 26h8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 36h8 |

**Operation:**

Lk = next IP

If (Ra==Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

For a floating-point comparison positive and negative zero are considered equal.

### BGE – Branch if Greater Than or Equal

**Description**:

This instruction branches to the target address if the contents of the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: BR,BRL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 29h8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 39h8 |

**Operation:**

Lk = next IP

If (Ra>=Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

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

**Description**:

This instruction branches to the target address if the contents of the Ra is greater than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: BR,BRL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2Dh8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Dh8 |

**Operation:**

Lk = next IP

If (Ra>=Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### BGT – Branch if Greater Than

**Description**:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: BR,BRL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2Bh8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Bh8 |

**Operation:**

Lk = next IP

If (Ra > Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### BGTU – Branch if Greater Than Unsigned

**Description**:

This instruction branches to the target address if the contents of the Ra is greater than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: BR,BRL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2Fh8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Fh8 |

**Operation:**

Lk = next IP

If (Ra > Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### BLT – Branch if Less Than

**Description**:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: BR,BRL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 28h8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 38h8 |

**Operation:**

Lk = next IP

If (Ra > Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### BLTU – Branch if Less Than Unsigned

**Description**:

This instruction branches to the target address if the contents of the Ra is less than that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: BR,BRL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2Ch8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Ch8 |

**Operation:**

Lk = next IP

If (Ra > Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### RET – 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.

**Formats Supported**: RET

|  |  |  |  |
| --- | --- | --- | --- |
| 15 11 | 10 9 | 8 | 7 0 |
| Const5 | Lk2 | 0 | F2h8 |

**Flags Affected**: none

**Operation:**

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

A branch instruction may also be used to return from a subroutine

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

## System Instructions

### BRK – Break

**Description**:

This instruction initiates the processor debug routine. The processor enters debug mode. The cause code register is set to indicate execution of a BRK instruction. Interrupts are disabled. The instruction pointer is reset to the contents of tvec[3] and instructions begin executing. There should be a jump instruction placed at the break vector location. The address of the BRK instruction is stored in the EIP register, C6.

**Instruction Format**: BRK

|  |  |  |
| --- | --- | --- |
| 15 9 | 8 | 7 0 |
| ~7 | 0 | 00h8 |

**Operation:**

PMSTACK = (PMSTACK << 4) | 6

CAUSE = FLT\_BRK

C[6] = IP

IP = tvec[3]

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

### CSRx – Control and Special / Status Access

**Description**:

The CSR instruction group provides access to control and special or status registers in the core. For the read operation the current value of the CSR is placed in the target register Rt.

**Instruction Format**: CSR

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 24 | 2321 | 20 15 | 14 9 | 8 | 7 0 |
| Regno16 | O3 | Ra6 | Rt6 | 0 | 0Fh8 |

|  |  |  |
| --- | --- | --- |
| O3 |  | Operation |
| 0 | CSRRD | Only read the CSR, no update takes place, Ra should be x0. |
| 1 | CSRRW | Read/Write to CSR |
| 2 | CSRRS | Read/Set CSR bits |
| 3 | CSRRC | Read/Clear CSR bits |
| 4 to 7 |  | Reserved |

CSRRS and CSRRC operations are only valid on registers that support the capability.

The Regno[15..12] field is reserved to specify the operating mode. Note that registers cannot be accessed by a lower operating mode.

**Execution Units:** Integer, the instruction may be available on only a single execution unit (not supported on all available integer units).

**Clock Cycles**: 1

**Exceptions**: privilege violation attempting to access registers outside of those allowed for the operating mode.

### DI – Disable Interrupts

**Description:**

This instruction disables interrupts for a short period of time. The Rb field specifies the number of following instructions for which interrupts are disabled. Interrupts may be disabled for a maximum of seven instructions.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 3229 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 16h7 | ~4 | Tb2 | Rb6 | 06 | ~6 | 0 | 07h8 |

**Operation:**

**Execution Units**: ALU

**Clock Cycles**:

**Exceptions**: none

**Notes**:

### MFSEL – Move from Selector Register

**Description:**

This instruction moves a selector register indirectly specified by Rb or directly if Rb is a constant, to register Rt.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 3229 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 28h7 | ~4 | Tb2 | Rb5 | 06 | Rt6 | 0 | 07h8 |

**Operation:**

Rt = B[Rb]

**Examples:**

LDI $t1,#7

MFSEL $t0,[$t1] ; get CS

STO $t0,CSSave

**Execution Units**: ALU

**Clock Cycles**:

**Exceptions**: none

### MTSEL – Move to Selector Register

**Description:**

This instruction moves register Ra to the selector register identified indirectly by the contents of Rb or directly if Rb is a constant. This instruction will load the associated descriptor cache from either the GDT or LDT tables.

It is not possible to move a value directly to the CS register as that would cause an unexpected change of program flow. Instead, the CS register must be loaded via a branch instruction which uses one of ES, FS, or GS as the selector.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 3229 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 29h7 | ~4 | Tb2 | Rb5 | Ra6 | ~6 | 0 | 07h8 |

**Operation:**

B[Rb] = Ra

Examples:

MTSEL DS,$a0

LDI $a1,#3

MTSEL [$a1],$a0 ; move indirect $a0 to b[$a1]

**Execution Units**: ALU

**Clock Cycles**:

**Exceptions**: none

**Notes**:

### PFI – Poll for Interrupt

**Description**:

The poll for interrupt instruction polls the interrupt status lines and performs an interrupt service if an interrupt is present. Otherwise, the PFI instruction is treated as a NOP operation. Polling for interrupts is performed by managed code. PFI provides a means to process interrupts at specific points in running software. Rt is loaded with the cause code in the low order eight bits, and the interrupt level in bits eight to eleven of the register.

**Instruction Format: SYS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 39 33 | 32 15 | 14 9 | 8 | 7 0 |
| 11h7 | ~18 | Rt6 | 0 | 07h8 |

**Clock Cycles**: 1 (if no exception present)

**Operation:**

if (irq <> 0)

Rt[7:0] = cause code

Rt[11:8] = irq level

PMSTACK = (PMSTACK << 4) | 6

CAUSE = Const8

C6 = IP

IP = tvec[3]

**Execution Units: Branch**

### REX – Redirect Exception

**Description**:

This instruction redirects an exception from an operating mode to a lower operating mode. This instruction if successful jumps to the target exception handler and does not return. If this instruction fails execution will continue with the next instruction.

This instruction may fail if exceptions are not enabled at the target level.

The location of the target exception handler is found in the trap vector register for that operating mode (tvec[xx]).

The cause (cause) and bad address (badaddr) registers of the originating mode are copied to the corresponding registers in the target mode.

**Instruction Format**: REX

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 31 | 30 21 | 20 15 | 14 9 | 8 | 7 0 |
| 10h7 | Tm2 | ~10 | Ra5 | 06 | 0 | 07h8 |

|  |  |
| --- | --- |
| Tm2 |  |
| 0 | redirect to user mode |
| 1 | redirect to supervisor mode |
| 2 | redirect to hypervisor mode |
| 3 | reserved |

**Clock Cycles**: 4

**Execution Units: Branch**

Example:

|  |
| --- |
| REX 1 ; redirect to supervisor handler  ; If the redirection failed, exceptions were likely disabled at the target level.  ; Continue processing so the target level may complete its operation.  RTE ; redirection failed (exceptions disabled ?) |

**Notes**:

Since all exceptions are initially handled in machine mode the machine handler must check for disabled lower mode exceptions.

### RTE – Return from Exception

**Description**:

Restore the previous interrupt enable setting and operating level and transfer program execution back to the address in the exception address register (C6). One of sixty-four semaphore registers specified by the Rb field of the instruction may also be cleared. Semaphore register zero is always cleared by this instruction.

This instruction may be encoded to return a short distance past the exception address point. This may be useful to return to the next instruction or return to a point past inline parameters. The constant12 field specifies a return offset in terms of bytes.

There is really only a single instruction to return from any mode for an exception. Although there are several additional mnemonics.

**Instruction Format: SYS**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 3229 | 2827 | 26 21 | 20 9 | 8 | 7 0 |
| 13h7 | ~4 | Tb2 | Rb6 | Constant12 | 0 | 07h8 |

**Flags Affected**: none

**Operation:**

PMSTACK = PMSTACK >> 4

Semaphore[0] = 0

Semaphore[Rb] = 0

IP = C6 + Constant

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

### 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:**

|  |
| --- |
| 7 0 |
| F7h8 |

### TLBRW – Read / Write TLB

**Description**:

This instruction both reads and writes the TLB. Which translation entry to update comes from the value in Ra. The update value comes from the value in Rb. Rb contains the virtual page number, ASID, and physical page number. The current value of the entry selected by Ra is copied to Rt. The TLB will be written only if bit 63 of Ra is set. A random way may be selected for write by setting the ‘r’ bit in the Ra value.

The entry number for Ra comes from virtual address bits 14 to 23.

Page numbers are in terms of a 4kB page size.

**Instruction Format: SYS**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 3229 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 1eh7 | ~4 | Tb2 | Rb6 | Ra6 | Rt6 | 0 | 07h8 |

**Clock Cycles**: 5

**Execution Units: Memory**

Ra Value Format

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 63 | 62 16 | 15 | 14 12 | 11 10 | 9 0 |
| w | ~ | r | ~ | way | entry no |

Rb/Rt Value Format

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 63 56 | 55 | 54 | 53 | 52 48 | 47 26 | 25 22 | 21 0 |
| ASID | G | D | A | UCRWX | VPN | ~4 | PPN |

|  |  |  |  |
| --- | --- | --- | --- |
| Bits |  | Meaning | |
| 0 to 21 | PPN | Physical page number (bits 22 to 43) | |
| 22 to 25 | ~ | reserved (expansion of physical page number) | |
| 26 to 49 | VPN | Virtual page number high address order bits 22 to 43 | |
| 48 | X | 1 = page is executable | These three combined indicate page present (P) 0 = not present |
| 49 | W | 1 = page is writeable |
| 50 | R | 1 = page is readable |
| 51 | C | 1 = page is cachable | |
| 52 | U | reserved for system usage | |
| 53 | A | Accessed, set if translation was used | |
| 54 | D | Dirty, set if a write occurred to the page | |
| 55 | G | Global, global translation indicator | |
| 56 to 63 | ASID | ASID address space identifier | |

**Exceptions:** none

### WFI – Wait for Interrupt

**Description**:

The WFI instruction waits for an external interrupt to occur before proceeding. While waiting for the interrupt, the processor clock is stopped placing the processor in a lower power mode.

**Instruction Format: SYS**

|  |  |  |
| --- | --- | --- |
| 15 9 | 8 | 7 0 |
| ~7 | 0 | FAh8 |

**Clock Cycles**: 1 (if no exception present)

**Execution Units: Branch**

## Vector Specific Instructions

### V2BITS

**Description**

Convert Boolean vector to bits. The least significant bit of each vector element is copied to the corresponding bit in the target register. The target register is a scalar register.

**Instruction Format: R1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 2422 | 21 | 20 15 | 14 9 | 8 | 7 0 |
| 18h7 | m3 | z | Ra5 | Rt6 | 1 | 01h8 |

**Operation**

For x = 0 to VL-1

if (Vm[x])

Rt[x] = Ra[x].LSB

else if (z)

Rt[x] = 0

else

Rt[x] = Rt[x]

**Exceptions:** none

### VBITS2V

**Description**

Convert bits to Boolean vector. Bits from a general register are copied to the corresponding vector target register.

**Instruction Format: R1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 24 22 | 21 | 20 15 | 14 9 | 8 | 7 0 |
| 19h7 | m3 | z | Ra6 | Vt6 | 1 | 01h8 |

**Operation**

For x = 0 to VL-1

if (Vm[x]) Vt[x] = Ra.bit[x]

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

**Exceptions:** none

### VMADD – Vector Mask Add

**Description:**

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

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23 20 | 1918 | 17 15 | 14 12 | 11 9 | 8 | 7 0 |
| 04h4 | ~2 | Vmb3 | Vma3 | Vmt3 | 0 | 52h8 |

1 clock cycle

**Exceptions:** none

### VMAND – Vector Mask And

**Description:**

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

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23 20 | 1918 | 17 15 | 14 12 | 11 9 | 8 | 7 0 |
| 08h4 | ~2 | Vmb3 | Vma3 | Vmt3 | 0 | 52h8 |

1 clock cycle

**Exceptions:** none

### VMFILL – Vector Mask Fill

**Description:**

Fill the contents of a vector mask register with a mask of ones beginning at Mb6 for width Mw6. Fill the remainder of the register with zeros.

**Instruction Format: VMR2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 23 18 | 17 12 | 11 9 | 8 | 7 0 |
| Me6 | Mh6 | Vmt3 | 0 | 44h8 |

1 clock cycle

**Exceptions:** none

### VMOR – Vector Mask Or

**Description:**

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

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23 20 | 1918 | 17 15 | 14 12 | 11 9 | 8 | 7 0 |
| 09h4 | ~2 | Vmb3 | Vma3 | Vmt3 | 0 | 52h8 |

1 clock cycle

**Exceptions:** none

### VMSUB – Vector Mask Subtract

**Description:**

Subtract the contents of two vector mask registers and place the result in a vector mask register.

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23 20 | 1918 | 17 15 | 14 12 | 11 9 | 8 | 7 0 |
| 05h4 | ~2 | Vmb3 | Vma3 | Vmt3 | 0 | 52h8 |

1 clock cycle

**Exceptions:** none

### VMXOR – Vector Mask Exclusive Or

**Description:**

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

**Instruction Format: VMR2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23 20 | 1918 | 17 15 | 14 12 | 11 9 | 8 | 7 0 |
| 0Ah4 | ~2 | Vmb3 | Vma3 | Vmt3 | 0 | 52h8 |

1 clock cycle

**Exceptions:** none

# Opcode Maps

## Root Opcode

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | BRK | {R1} | {R2} | {R3} | ADDI  11 | SUBFI  11 |  | {SYS} | ANDI  11 | ORI  11 | EORI  11 |  |  |  |  | {CSR} |
| 1x |  | {R1R} | {R2R} | {R3R} |  |  | SEQ11 | SNE11 | SLT11 |  |  | SGT11 | SLTU  11 |  |  | SGTU  11 |
| 2x |  |  |  |  | BBC | BBS | BEQ | BNE | BLT | BGE | BLE | BGT | BLTU | BGEU | BLEU | BGTU |
| 3x |  |  |  |  | BBC | BBS | BEQ | BNE | BLT | BGE | BLE | BGT | BLTU | BGEU | BLEU | BGTU |
| 4x |  |  | {P} |  | VMFILL | CHKI | BITI |  |  |  | MULI | DIVI |  |  | MULUI | DIVUI |
| 5x | {logic} | MLO | {VM} | ANDI  27 | ORI  27 | EORI  27 | {VR} | {VAdd } | {shift} |  | VMAC | MODI | VCMPS | CHKXI | {VMUL} | MODUI |
| 6x | ADDI  27 |  | ADDI  10 |  | SUBFI 27 |  |  |  |  |  | LLA | \_2ADDUI | \_4ADDUI | \_8ADDUI | \_16ADDUI | LDI |
| 7x | JGR |  | MUX |  |  |  | {FMAC} | {double r} | {float rr} | {single r} |  |  |  |  |  |  |
| 8x | LDB | LDBU | LDW | LDWU | LDT | LDTU | LDO | LFS | LFD |  |  | LVWAR | SWCR | JSRI | LWS | LCL |
| 9x | STB | STW | STT | STO |  |  | STI | CAS | STSET | STMOV | STCMP | STFND |  | LDIS | SWS | CACHE |
| Ax | JSR | JSR | JSR | RTS |  | SYS | INT | {R} | MFSPR | MTSPR | {bitfld} | MOVS |  |  |  |  |
| Bx | {LDxX} |  |  |  |  |  |  | JSRIX | LLAX |  |  |  |  | LV | LVWS | LVX |
| Cx | {STxX} |  |  |  |  |  | STIX | INC | PUSH | PEA | POP | LINK | UNLINK | SV | SVWS | SVX |
| Dx | LDBL | LDBUL | LDWL | LDWUL | LDTL | LDTUL | LDOL |  |  |  |  |  |  |  |  |  |
| Ex | STBL | STWL | STTL | STOL |  |  |  |  |  |  |  |  |  |  |  |  |
| Fx | {TLB} | NOP | RET | RTE | RTI | {BCD} | STP | SYNC | MEMSB | MEMDB | WFI | SEI |  |  |  | IMM |

## {R1} Integer Monadic Register Ops – Func7

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| x000 | CNTLZ | CNTLO | CNTPOP |  | NOT |  | ABS | NABS |
| x001 | SQRT |  |  | TST |  |  |  |  |
| x010 | PTRINC | TRANSFORM |  |  |  |  |  |  |
| x011 | V2BITS | BITS2V |  |  | VCMPRSS | VCIDX | VSCAN |  |
| x100 | CLIP |  |  |  |  |  |  |  |
| x101 |  |  |  |  |  |  |  |  |
| x110 |  |  |  |  |  |  |  |  |
| x111 |  |  |  |  |  |  |  |  |