|  |  |
| --- | --- |
| A picture of a winding road and trees  [Document title]  [Document subtitle] | Abstract  [Draw your reader in with an engaging abstract. It is typically a short summary of the document. When you’re ready to add your content, just click here and start typing.]  Robert Finch  [Course title] |

Table of Contents

[Overview 1](#_Toc84880495)

[History 1](#_Toc84880496)

[Design Objectives 1](#_Toc84880497)

[Motivation 2](#_Toc84880498)

[Case Comparison Hi-lites 2](#_Toc84880499)

[Case Comparison 6502 2](#_Toc84880500)

[Case Comparison ARM 3](#_Toc84880501)

[Case Comparison RISCV 3](#_Toc84880502)

[Case Comparison MMIX 5](#_Toc84880503)

[Case Comparison PowerPC 5](#_Toc84880504)

[Case Comparison x86 5](#_Toc84880505)

[Case Comparison SPARC 6](#_Toc84880506)

[Nomenclature 6](#_Toc84880507)

[Development Aspects 6](#_Toc84880508)

[Device Target 6](#_Toc84880509)

[Implementation Language 6](#_Toc84880510)

[Programming Model 7](#_Toc84880511)

[General Registers 7](#_Toc84880512)

[Stack and Frame Pointers 8](#_Toc84880513)

[Loop Count 8](#_Toc84880514)

[Code Address Registers 8](#_Toc84880515)

[Code Address Register Format 8](#_Toc84880516)

[Instruction Pointer 9](#_Toc84880517)

[Link Registers 9](#_Toc84880518)

[Selector Registers 9](#_Toc84880519)

[General Purpose Vector (v0 to v31) / Registers 10](#_Toc84880520)

[Mask Registers (m0 to m7) 10](#_Toc84880521)

[Summary of Special Purpose Registers 10](#_Toc84880522)

[C0,C1,…C7 (CREGS) – SPR #16 to 31 10](#_Toc84880523)

[ZS,DS,ES,FS,GS,HS,SS,CS (SREGS) – SPR #32 to 39 10](#_Toc84880524)

[LDT – SPR #40 10](#_Toc84880525)

[GDT – SPR #41 11](#_Toc84880526)

[KEYS – SPR #48,#49 11](#_Toc84880527)

[TICK - SPR #50 11](#_Toc84880528)

[VL – SPR #54 11](#_Toc84880529)

[TVEC – SPR #64 to #71 12](#_Toc84880530)

[M\_HARTID (0x3001) 12](#_Toc84880531)

[M\_BADADDR (CSR 0x3007) 12](#_Toc84880532)

[Operating Modes 13](#_Toc84880533)

[Exceptions 13](#_Toc84880534)

[External Interrupts 13](#_Toc84880535)

[Polling for Interrupts 13](#_Toc84880536)

[Effect on Machine Status 13](#_Toc84880537)

[Exception Stack 13](#_Toc84880538)

[Exception Vectoring 14](#_Toc84880539)

[Reset 14](#_Toc84880540)

[Precision 14](#_Toc84880541)

[Exception Cause Codes 14](#_Toc84880542)

[DBG 16](#_Toc84880543)

[IADR 16](#_Toc84880544)

[UNIMP 16](#_Toc84880545)

[OFL 16](#_Toc84880546)

[KEY 16](#_Toc84880547)

[FLT 16](#_Toc84880548)

[DRF, DWF, EXF 16](#_Toc84880549)

[CPF, DPF 16](#_Toc84880550)

[PRIV 16](#_Toc84880551)

[STK 17](#_Toc84880552)

[DBE 17](#_Toc84880553)

[PMA 17](#_Toc84880554)

[IBE 17](#_Toc84880555)

[NMI 17](#_Toc84880556)

[BT 17](#_Toc84880557)

[Segmentation 18](#_Toc84880558)

[Overview 18](#_Toc84880559)

[Privilege levels 18](#_Toc84880560)

[Usage 18](#_Toc84880561)

[Software Support 18](#_Toc84880562)

[Address Formation: 19](#_Toc84880563)

[Selecting a segment register 19](#_Toc84880564)

[Selectors 19](#_Toc84880565)

[Selector Format: 19](#_Toc84880566)

[Descriptor Cache 20](#_Toc84880567)

[Non-Segmented Code Area 20](#_Toc84880568)

[Changing the Code Segment 20](#_Toc84880569)

[The Descriptor Table 20](#_Toc84880570)

[System Segment Descriptors 22](#_Toc84880571)

[Segment Load Exception 22](#_Toc84880572)

[Segment Bounds Exception 23](#_Toc84880573)

[Segment Usage Conventions 23](#_Toc84880574)

[Power-up State 23](#_Toc84880575)

[Segment Registers 23](#_Toc84880576)

[Instruction Set Description 25](#_Toc84880577)

[Overview 25](#_Toc84880578)

[Root Opcode 25](#_Toc84880579)

[Vector Instruction Indicator 25](#_Toc84880580)

[Target Register Spec 25](#_Toc84880581)

[Register Formats 26](#_Toc84880582)

[R1 (one source register) 26](#_Toc84880583)

[R1L (one source register) 26](#_Toc84880584)

[R2 (two source register) 26](#_Toc84880585)

[R2L (two source register) 26](#_Toc84880586)

[R3 (three source register) 26](#_Toc84880587)

[R3L (three source register) 26](#_Toc84880588)

[Arithmetic / Logical / Shift 27](#_Toc84880589)

[ABS – Absolute Value 27](#_Toc84880590)

[ADD - Register-Register 28](#_Toc84880591)

[ADDI – Add Immediate 29](#_Toc84880592)

[ADDIL – Add Immediate Long 30](#_Toc84880593)

[ADDIQ – Add Immediate Quick 31](#_Toc84880594)

[ADDIS - Add Immediate Shifted 32](#_Toc84880595)

[AND – Bitwise And 32](#_Toc84880596)

[ANDI – Bitwise And Immediate 33](#_Toc84880597)

[ANDIL – Bitwise And Immediate Long 34](#_Toc84880598)

[ANDIS - And Immediate Shifted 35](#_Toc84880599)

[MUL – Multiply 36](#_Toc84880600)

[MUL[O] – Multiply 37](#_Toc84880601)

[MULI – Multiply Immediate 38](#_Toc84880602)

[MULF – Fast Unsigned Multiply 39](#_Toc84880603)

[MULFI – Fast Unsigned Multiply Immediate 40](#_Toc84880604)

[NOT 41](#_Toc84880605)

[OR – Bitwise Or 42](#_Toc84880606)

[ORI – Bitwise Or Immediate 43](#_Toc84880607)

[ORIL – Bitwise Or Immediate Long 44](#_Toc84880608)

[ORIS - Or Immediate Shifted 45](#_Toc84880609)

[SEQ – Set if Equal 46](#_Toc84880610)

[SEQI – Set if Equal Immediate 46](#_Toc84880611)

[SGT – Set if Greater Than 47](#_Toc84880612)

[SGTI – Set if Greater Than Immediate 47](#_Toc84880613)

[SLL –Shift Left Logical 48](#_Toc84880614)

[SLT – Set if Less Than 49](#_Toc84880615)

[SLTI – Set if Less Than Immediate 50](#_Toc84880616)

[SLEI – Set if Less Than or Equal Immediate 50](#_Toc84880617)

[SNE – Set if Not Equal 51](#_Toc84880618)

[SNEI – Set if Not Equal Immediate 51](#_Toc84880619)

[SUBF – Subtract From 52](#_Toc84880620)

[SUBFI – Subtract from Immediate 53](#_Toc84880621)

[XOR – Bitwise Exclusive Or 54](#_Toc84880622)

[XORI – Bitwise Exclusive Or Immediate 55](#_Toc84880623)

[XORIL – Bitwise Exclusive Or Immediate Long 56](#_Toc84880624)

[XORIS – Exclusive Or Immediate Shifted 56](#_Toc84880625)

[Load / Store Instructions 57](#_Toc84880626)

[Overview 57](#_Toc84880627)

[Addressing Modes 57](#_Toc84880628)

[Load Formats 57](#_Toc84880629)

[Store Formats 58](#_Toc84880630)

[LDB – Load Byte 59](#_Toc84880631)

[LDBL – Load Byte, Long Address 60](#_Toc84880632)

[LDBU – Load Byte, Unsigned 61](#_Toc84880633)

[LDBUL – Load Byte Unsigned, Long Address 62](#_Toc84880634)

[LDBUX – Load Byte Unsigned Indexed 63](#_Toc84880635)

[LDBX – Load Byte Indexed 64](#_Toc84880636)

[LDT – Load Tetra 65](#_Toc84880637)

[LDTL – Load Tetra, Long Address 66](#_Toc84880638)

[LDTU – Load Tetra Unsigned 67](#_Toc84880639)

[LDW – Load Wyde 68](#_Toc84880640)

[LDWL – Load Wyde, Long Address 69](#_Toc84880641)

[LDWU – Load Wyde Unsigned 70](#_Toc84880642)

[LDWUL – Load Wyde Unsigned, Long Address 71](#_Toc84880643)

[LDWX – Load Wyde Indexed 72](#_Toc84880644)

[LDWUX – Load Wyde Unsigned Indexed 73](#_Toc84880645)

[STB – Store Byte 74](#_Toc84880646)

[STBL – Store Byte, Long Addressing 75](#_Toc84880647)

[STBX – Store Byte Indexed 76](#_Toc84880648)

[STT – Store Tetra 77](#_Toc84880649)

[STTL – Store Tetra, Long Addressing 78](#_Toc84880650)

[STTX – Store Tetra Indexed 79](#_Toc84880651)

[STW – Store Wyde 80](#_Toc84880652)

[STWL – Store Wyde, Long Addressing 81](#_Toc84880653)

[STWX – Store Wyde Indexed 82](#_Toc84880654)

[Branch / Flow Control Instructions 83](#_Toc84880655)

[Overview 83](#_Toc84880656)

[Branch Format 83](#_Toc84880657)

[Branch Conditions 84](#_Toc84880658)

[Linkage 84](#_Toc84880659)

[Branch Target 85](#_Toc84880660)

[Near or Far Branching 86](#_Toc84880661)

[Branch to Register 86](#_Toc84880662)

[[D]BBC – Branch if Bit Clear 87](#_Toc84880663)

[[D]BBS – Branch if Bit Set 88](#_Toc84880664)

[[D]BEQ – Branch if Equal 89](#_Toc84880665)

[[D]BGE – Branch if Greater Than or Equal 90](#_Toc84880666)

[[D]BGEU – Branch if Greater Than or Equal Unsigned 91](#_Toc84880667)

[[D]BGT – Branch if Greater Than 92](#_Toc84880668)

[[D]BGTU – Branch if Greater Than Unsigned 93](#_Toc84880669)

[[D]BLE – Branch if Less Than or Equal 94](#_Toc84880670)

[[D]BLEU – Branch if Less Than or Equal Unsigned 95](#_Toc84880671)

[[D]BLT – Branch if Less Than 96](#_Toc84880672)

[[D]BLTU – Branch if Less Than Unsigned 97](#_Toc84880673)

[[D]BNE – Branch if Not Equal 98](#_Toc84880674)

[[D]BRA – Branch Always 99](#_Toc84880675)

[[D]BSR – Branch to Subroutine 99](#_Toc84880676)

[[D]JBC – Jump if Bit Clear 101](#_Toc84880677)

[[D]JBS – Jump if Bit Set 102](#_Toc84880678)

[[D]JEQ – Jump if Equal 103](#_Toc84880679)

[[D]JGE – Jump if Greater Than or Equal 104](#_Toc84880680)

[[D]JGEU – Jump if Greater Than or Equal Unsigned 105](#_Toc84880681)

[[D]JGT – Jump if Greater Than 106](#_Toc84880682)

[[D]JGTU – Jump if Greater Than Unsigned 107](#_Toc84880683)

[[D]JLE – Jump if Less Than or Equal 108](#_Toc84880684)

[[D]JLEU – Jump if Less Than or Equal Unsigned 109](#_Toc84880685)

[[D]JLT – Jump if Less Than 110](#_Toc84880686)

[[D]JLTU – Jump if Less Than Unsigned 111](#_Toc84880687)

[[D]JNE – Jump if Not Equal 112](#_Toc84880688)

[[D]JMP – Jump 113](#_Toc84880689)

[[D]JSR – Jump to Subroutine 114](#_Toc84880690)

[NOP – No Operation 115](#_Toc84880691)

[RTS – Return from Subroutine 116](#_Toc84880692)

[System Instructions 117](#_Toc84880693)

[BRK – Break 117](#_Toc84880694)

[CSRx – Control and Special / Status Access 118](#_Toc84880695)

[DI – Disable Interrupts 119](#_Toc84880696)

[MFSEL – Move from Selector Register 120](#_Toc84880697)

[MTSEL – Move to Selector Register 121](#_Toc84880698)

[PFI – Poll for Interrupt 122](#_Toc84880699)

[REX – Redirect Exception 123](#_Toc84880700)

[RTE – Return from Exception 124](#_Toc84880701)

[SYNC -Synchronize 124](#_Toc84880702)

[TLBRW – Read / Write TLB 125](#_Toc84880703)

[WFI – Wait for Interrupt 126](#_Toc84880704)

[Vector Specific Instructions 127](#_Toc84880705)

[V2BITS 127](#_Toc84880706)

[VBITS2V 128](#_Toc84880707)

[VMADD – Vector Mask Add 128](#_Toc84880708)

[VMAND – Vector Mask And 128](#_Toc84880709)

[VMFILL – Vector Mask Fill 129](#_Toc84880710)

[VMOR – Vector Mask Or 130](#_Toc84880711)

[VMSLL – Vector Mask Shift Left Logical 130](#_Toc84880712)

[VMSUB – Vector Mask Subtract 131](#_Toc84880713)

[VMXOR – Vector Mask Exclusive Or 131](#_Toc84880714)

[Opcode Maps 132](#_Toc84880715)

[Root Opcode 132](#_Toc84880716)

[{R1} Integer Monadic Register Ops – Func7 132](#_Toc84880717)

# Overview

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

## History

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

## Design Objectives

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

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

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

## Motivation

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

### Case Comparison Hi-lites

Some of the more striking points of a handful of architectures are compared to what is available in Thor2021.

### Case Comparison 6502

6502 vs Thor2021

#### Overview

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

#### Instruction Format

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

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

#### Registers

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

#### Instructions

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

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

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

### Case Comparison ARM

#### Overview

The ARM architecture has become extremely popular.

#### Instruction Format

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

#### Registers

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

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

### Case Comparison RISCV

RISCV vs Thor2021

#### Instruction Format

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

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

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

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

#### Instructions

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

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

#### Register File

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

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

#### Return Address Registers

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

#### Compare Results Registers

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

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

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

#### Operating modes.

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

#### Memory Management

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

### Case Comparison MMIX

#### Instruction Format

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

#### Register File

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

#### Instructions

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

### Case Comparison PowerPC

#### Instruction Format

The PowerPC uses a fixed 32-bit instruction format.

#### Instructions

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

#### Registers

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

### Case Comparison x86

#### Registers

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

### Case Comparison SPARC

#### Registers

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

# Nomenclature

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

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

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

# Development Aspects

## Device Target

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

## Implementation Language

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

# Programming Model

## General Registers

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

Register #0 is always zero.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | |
| r0 | always zero |  |  | LC | Loop Counter | |
| r1 | return value / arg0 |  |  |  |  | |
| r2 | return value / arg1 |  |  | 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.

## Stack and Frame Pointers

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

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

## Loop Count

The loop count register 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.

## General Purpose Vector (v0 to v31) / Registers

v0 always has the value zero.

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

## Mask Registers (m0 to m7)

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

|  |  |
| --- | --- |
| Register | Usage |
| m0 | contains all ones by convention |
| m1 to m7 |  |

## 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 |

### M\_HARTID (0x3001)

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

### M\_BADADDR (CSR 0x3007)

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

# 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.

# Exceptions

## External Interrupts

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

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

## Polling for Interrupts

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

## Effect on Machine Status

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

## Exception Stack

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

## Exception Vectoring

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

|  |  |  |
| --- | --- | --- |
| Operating Level | Address (If TVEC[3] contains $F…FC0000) |  |
| 0 | $F…FC0000 | Handler for operating level zero |
| 1 | $F…FC0020 |  |
| 2 | $F…FC0040 |  |
| 3 | $F…FC0060 |  |

## Reset

The core begins executing instructions at address $F…FC0100. All registers are in an undefined state. Register set #0 is selected.

## Precision

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

## Exception Cause Codes

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cause Code |  | HW | Description |  |
| 0 |  |  | no exception |  |
| 1 | IBE | x | instruction bus error |  |
| 2 | EXF | x | Executable fault |  |
| 4 | TLB | x | tlb miss |  |
|  |  |  | FMTK Scheduler |  |
| 128 |  | e |  |  |
| 129 | KRST | e | Keyboard reset interrupt |  |
| 130 | MSI | e | Millisecond Interrupt |  |
| 131 | TICK | e |  |  |
| 156 | KBD | e | Keyboard interrupt |  |
| 157 | GCS | e | Garbage collect stop |  |
| 158 | GC | e | Garbage collect |  |
| 159 | TSI | e | FMTK Time Slice Interrupt |  |
| 3 |  |  | Control-C pressed |  |
| 20 |  |  | Control-T pressed |  |
| 26 |  |  | Control-Z pressed |  |
|  |  |  |  |  |
| 32 | SSM | x | single step |  |
| 33 | DBG | x | debug exception |  |
| 34 | TGT | x | call target exception |  |
| 35 | MEM | x | memory fault |  |
| 36 | IADR | x | bad instruction address |  |
| 37 | UNIMP | x | unimplemented instruction |  |
| 38 | FLT | x | floating point exception |  |
| 39 | CHK | x | bounds check exception |  |
| 40 | DBZ | x | divide by zero |  |
| 41 | OFL | x | overflow |  |
|  |  |  |  |  |
| 47 |  |  |  |  |
| 48 | ALN | x | data alignment |  |
| 49 | KEY | x | memory key fault |  |
| 50 | DWF | x | Data write fault |  |
| 51 | DRF | x | data read fault |  |
| 52 | SGB | x | segment bounds violation |  |
| 53 | PRIV | x | privilege level violation |  |
| 54 | CMT | x | commit timeout |  |
| 55 | BT | x | branch target |  |
| 56 | STK | x | stack fault |  |
| 57 | CPF | x | code page fault |  |
| 58 | DPF | x | data page fault |  |
| 60 | DBE | x | data bus error |  |
| 61 | PMA | x | physical memory attributes check fail |  |
| 62 | NMI | x | Non-maskable interrupt |  |
|  |  |  |  |  |
| 225 | FPX\_IOP | x | Floating point invalid operation |  |
| 226 | FPX\_DBZ | x | Floating point divide by zero |  |
| 227 | FPX\_OVER | x | floating point overflow |  |
| 228 | FPX\_UNDER | x | floating point underflow |  |
| 229 | FPX\_INEXACT | x | floating point inexact |  |
| 231 | FPX\_SWT | x | floating point software triggered |  |
| 239 |  |  | Software exception handling |  |
| 240 | SYS |  | Call operating system (FMTK) |  |
| 241 |  |  | FMTK Schedule interrupt |  |
| 242 | TMR | x | system timer interrupt |  |
| 243 | GCI | x | garbage collect interrupt |  |
| 253 | RST | x | reset |  |
| 254 | NMI | x | non-maskable interrupt |  |
| 255 | PFI |  | reserved for poll-for-interrupt instruction |  |

### DBG

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

### IADR

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

### UNIMP

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

### OFL

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

### KEY

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

### FLT

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

### DRF, DWF, EXF

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

### CPF, DPF

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

### PRIV

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

### STK

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

### DBE

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

### PMA

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

### IBE

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

### NMI

Non-maskable interrupt.

### BT

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

# 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 |

### R1L (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 |

### R2L (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 |

### R3L (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 |

## Arithmetic / Logical / Shift

### ABS – Absolute Value

**Description:**

This instruction takes the absolute value of a register and places the result in a target register.

**Integer Instruction Format: R1**

Both the source and target registers are treated as integer values.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 24 22 | 21 | 20 15 | 14 9 | 8 | 7 0 |
| 06h7 | m3 | z | Ra6 | Rt6 | v | 01h8 |

v: 0 = scalar, 1 = vector op

**Operation:**

If Ra < 0

Rt = -Ra

else

Rt = Ra

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Rt[x] = Ra[x] < 0 ? -Ra[x] : Ra[x]

else if (z) Rt[x] = 0

else Rt[x] = Rt[x]

**Execution Units:** I

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### ADD - Register-Register

**Description:**

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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:** All ALU’s

**Operation:**

Rt = Ra + Rb

**Exceptions:**

**Notes:**

### ADDI – Add 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: RI

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

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra + immediate

**Exceptions:**

**Notes:**

### ADDIL – Add Immediate Long

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 | 44h8 |

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra + immediate

**Exceptions:**

**Notes:**

*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.*

### ADDIQ – Add Immediate Quick

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: RIQ

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

Clock Cycles: 1

Execution Units: All ALU’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.*

### 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: RIS

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

Clock Cycles: 1

Execution Units: All ALU’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)

**Operation:**

Rt = Ra & Rb

**Exceptions**: none

### ANDC – Bitwise And with Complement

**Description**:

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

**Integer Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 0Bh7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

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

**Operation:**

Rt = Ra & ~Rb

**Exceptions**: none

### ANDI – Bitwise And Immediate

**Description**:

Perform a bitwise and operation between operands. The immediate constant is one extended before use.

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 ‘and’ 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 extracting or clearing bit fields.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 8h4 | 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

### ANDIL – Bitwise And Immediate Long

**Description**:

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

Instruction Format: RIL

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

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

Operation

**Rt = Ra & Immediate**

**Vector Operation**

for x = 0 to VL-1

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

else Vt[x] = Vt[x]

**Exceptions**: none

### ANDIS - And Immediate Shifted

**Description:**

Bitwise ‘and’ 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 | 58h8 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra & (immediate << 32)

**Exceptions:**

**Notes:**

### BCDADD – BCD Add

**Description:**

Adds two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is an eight-bit BCD number.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 00h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | F5h8 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra + Rb

**Exceptions:** none

### BCDMUL – BCD Multiply

**Description:**

Multiply two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is an sixteen-bit BCD number.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 02h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | F5h8 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra \* Rb

**Exceptions:** none

### BCDSUB – BCD Subtract

**Description:**

Subtract two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is an eight-bit BCD number.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 01h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | F5h8 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra - Rb

**Exceptions:** none

### BFCHG – Bitfield Change

**Description**:

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

**Integer Instruction Format: RM**

Generates a mask consisting of ‘1’s between the mask begin, Mb6, and mask end, Me6, inclusive. Bitwise exclusive ‘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 flipping bit fields.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Ah4 | S | ~2 | Me6 | Mb6 | Ra6 | Rt6 | v | AAh7 |

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

**Operation**

Rt = Ra ^ Immediate

**Vector Operation**

for x = 0 to VL-1

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

else Vt[x] = Vt[x]

**Exceptions**: none

### BFCLR – Bitfield Clear

**Description**:

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

**Integer Instruction Format: RM**

Generates a mask consisting of ‘1’s between the mask begin, Mb6, and mask end, Me6, inclusive. Bitwise and with the complement of 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 flipping bit fields.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Bh4 | S | ~2 | Me6 | Mb6 | Ra6 | Rt6 | v | AAh7 |

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

**Operation**

Rt = Ra & ~Immediate

**Vector Operation**

for x = 0 to VL-1

if (Vm0[x]) Vt[x] = Va[x] & ~Immediate

else Vt[x] = Vt[x]

**Exceptions**: none

### BFEXT – Bitfield Extract

**Description**:

Extract the bits of a bitfield specified between mask begin and mask end and right align the result in the target register.

**Integer Instruction Format: RM**

Generates a mask consisting of ‘1’s between the mask begin, Mb6, and mask end, Me6, inclusive. Bitwise exclusive ‘and’ the mask with the value in register Ra rotate the result to the right by Mb6 and store the result in register Rt. The result is sign extended from bit Me6 to the width of the machine.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 5h4 | 1 | ~2 | Me6 | Mb6 | Ra6 | Rt6 | v | AAh7 |

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

**Operation**

Rt = Ra ^ Immediate

**Vector Operation**

for x = 0 to VL-1

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

else Vt[x] = Vt[x]

**Exceptions**: none

### BFEXTU – Bitfield Extract Unsigned

**Description**:

Extract the bits of a bitfield specified between mask begin and mask end and right align the result in the target register.

**Integer Instruction Format: RM**

Generates a mask consisting of ‘1’s between the mask begin, Mb6, and mask end, Me6, inclusive. Bitwise exclusive ‘and’ the mask with the value in register Ra rotate the result to the right by Mb6 and store the result in register Rt.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 5h4 | 0 | ~2 | Me6 | Mb6 | Ra6 | Rt6 | v | AAh7 |

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

**Operation**

Rt = (Ra & Immediate) ror by Mb

**Vector Operation**

for x = 0 to VL-1

if (Vm0[x]) Vt[x] = (Va[x] & Immediate) ror by Mb

else Vt[x] = Vt[x]

**Exceptions**: none

### BFINS – Bit-field Insert

**Description:**

Inserts a bit-field into the target register located between the mask begin (mb) and mask end (me) bits from the low order bits of Ra.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 0h4 | S | ~2 | Me6 | Mb6 | Ra6 | Rt6 | v | AAh7 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### BFINSI – Bit-field Insert Immediate

**Description:**

Inserts a bit-field into the target register located between the mask begin (mb) and mask end (me) bits from an immediate constant in the instruction.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 6h4 | S | Im2 | Me6 | Mb6 | Imm6 | Rt6 | v | AAh7 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### BFSET – Bitfield Set

**Description**:

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

**Integer Instruction Format: RM**

Generates a mask consisting of ‘1’s between the mask begin, Mb6, and mask end, Me6, 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.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 9h4 | S | ~2 | Me6 | Mb6 | Ra6 | Rt6 | v | AAh7 |

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

**Operation**

Rt = Ra | Immediate

**Vector Operation**

for x = 0 to VL-1

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

else Vt[x] = Vt[x]

**Exceptions**: none

### BYTNDX – Byte Index

**Description:**

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

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

**Instruction Format:** RI

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 55h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rt = Index of (Rb in Ra)

**Exceptions:** none

### BYTNDXI – Byte Index

**Description:**

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

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

**Instruction Format:** RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 55h8 |

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rt = Index of (Imm8 in Ra)

**Exceptions:** none

### CMP – Compare

**Description**

Compare two registers and return the relationship between them.

**Integer Instruction Format: R2**

Both values are treated as signed numbers.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 2Ah7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

**1 clock cycle**

**Operation:**

Rt = Ra < Rb ? –1 : Ra = Rb ? 0 : 1

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] < Vb[x] ? –1 : Va[x]=Vb[x] ? 0 : 1

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

### CMPI – Compare Immediate

**Description**

Compare a register and an immediate value and return the relationship between them.

Both values are treated as signed numbers.

**Instruction Format:** RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 50h8 |

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

**Operation:**

Rt = Ra < Imm ? –1 : Ra = Imm ? 0 : 1

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] < Imm ? –1 : Va[x]=Imm ? 0 : 1

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

### CMPIL – Compare Immediate Long

**Description**

Compare a register and an immediate value and return the relationship between them.

Both values are treated as signed numbers.

**Instruction Format:** RIL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Ra6 | Rt6 | v | 60h8 |

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

**Operation:**

Rt = Ra < Imm ? –1 : Ra = Imm ? 0 : 1

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] < Imm ? –1 : Va[x]=Imm ? 0 : 1

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

### CMPIS – Compare Immediate Shifted

**Description**

Compare a register and an immediate value and return the relationship between them.

Both values are treated as signed numbers.

**Instruction Format:** RIL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Ra6 | Rt6 | v | 70h8 |

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

**Operation:**

Rt = Ra < Imm ? –1 : Ra = Imm ? 0 : 1

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] < Imm ? –1 : Va[x]=Imm ? 0 : 1

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

### CMPU – Compare Unsigned

**Description**

Compare two registers and return the relationship between them.

**Integer Instruction Format: R2**

Both values are treated as unsigned numbers.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 2Bh7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

**1 clock cycle**

**Operation:**

Rt = Ra < Rb ? –1 : Ra = Rb ? 0 : 1

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] < Vb[x] ? –1 : Va[x]=Vb[x] ? 0 : 1

else if (z) Vt[x] = 0

else Vt[x] = Vt[x]

### EOR – Bitwise Exclusive 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 |
| 0Ah7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

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

**Exceptions**: none

### EORI – Bitwise Exclusive Or Immediate

**Description**:

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

Instruction Format: RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 0Ah8 |

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

**Integer Instruction Format: RM**

Generates a mask consisting of ‘1’s between the mask begin, Mb6, and mask end, Me6, inclusive. Bitwise exclusive ‘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 flipping bit fields.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Ah4 | S | ~2 | Me6 | 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

### EORIL – Bitwise Exclusive Or Immediate Long

**Description**:

Perform a bitwise exclusive ‘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 | 4Ah8 |

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

Operation

Rt = Ra ^ Immediate

**Vector Operation**

for x = 0 to VL-1

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

else Vt[x] = Vt[x]

**Exceptions**: none

### EORIS – Exclusive Or Immediate Shifted

Description:

Bitwise exclusive ‘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.

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Rt6 | Ra6 | v | 5Ah8 |

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra ^ (immediate << 32)

**Exceptions:**

**Notes:**

### MUL – Multiply

**Description**:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as signed values, the result is a signed result. The register form of the instruction may cause an overflow exception if the overflow enable bit in the instruction is set.

**Integer Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 06h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

4 clock cycles

**Exceptions**: none

**Execution Units**: 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

### MUL[O] – Multiply

**Description**:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as signed values, the result is a signed result. This form of the instruction may cause an overflow exception if the overflow enable bit in the instruction is set.

**Integer Instruction Format: R2L**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 41 | 40 | 39 36 | 35 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 06h7 | O | ~4 | ~3 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 12h8 |

4 clock cycles

**Exceptions**: none

**Execution Units**: 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**: overflow, if enabled

### MULI – Multiply Immediate

**Description**:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as signed values, the result is a signed result. The register form of the instruction may cause an overflow exception if the overflow enable bit in the instruction is set.

**Integer Instruction Format: RI**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 06h8 |

4 clock cycles

**Execution Units**: ALU

**Operation:**

Rt = Ra \* Immediate

**Vector Operation**

for x = 0 to VL - 1

if (Vm0[x]) Vt[x] = Va[x] \* Vb[x]

else Vt[x] = Vt[x]

**Exceptions**: none

### MULF – Fast Unsigned Multiply

**Description**:

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

**Integer Instruction Format: R2**

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

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

**Execution Units**: ALU

**Clock Cycles:** 1

**Exceptions**: none

### MULFI – Fast Unsigned Multiply Immediate

**Description**:

Multiply two values. The first operand is in register Ra. The second operand is an immediate value specified in the instruction. Both the operands are treated as unsigned values. The result is an unsigned result. The fast multiply multiplies only the low order 24 bits of the first operand times the low order 16 bits of the second. The result is a 40-bit unsigned product.

**Integer Instruction Format: RI**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 39 37 | 36 21 | 20 15 | 14 9 | 8 | 7 0 |
| ~3 | Constant16 | Ra6 | Rt6 | v | 15h8 |

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

**Execution Units**: ALU

**Clock Cycles:** 1

**Exceptions**: none

### NOT

Description:

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

**Integer Instruction Format: R1**

Both the source and target registers are treated as integer values.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 25 | 24 22 | 21 | 20 15 | 14 9 | 8 | 7 0 |
| 04h7 | m3 | z | Ra6 | Rt6 | v | 01h8 |

v: 0 = scalar, 1 = vector op

Operation:

Rt = !Ra

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Rt[x] = !Ra[x]

else if (z) Rt[x] = 0

else Rt[x] = Rt[x]

Execution Units: I

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### 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

### ORC – Bitwise Or with Complement

**Description**:

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

**Integer Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 03h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

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

**Operation:**

Rt = Ra | ~Rb

**Exceptions**: none

### ORI – Bitwise Or Immediate

**Description**:

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

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

### ORIL – Bitwise Or Immediate Long

**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 |

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

Operation

**Rt = Ra | Immediate**

**Vector Operation**

for x = 0 to VL-1

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

else Vt[x] = Vt[x]

**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 | 59h8 |

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra | (immediate << 32)

**Exceptions:**

**Notes:**

### SEQ – Set if Equal

**Description:**

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

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 16h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

### SEQI – Set if Equal Immediate

**Description:**

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

Instruction Format: RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 16h8 |

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

### SGT – Set if Greater Than

**Description:**

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

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 1Bh7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

### SGTI – Set if Greater Than Immediate

**Description:**

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

Instruction Format: RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 1Bh8 |

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

### SGTIL – Set if Greater Than Immediate Long

**Description:**

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

**Instruction Format:** RIL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Ra6 | Rt6 | v | 1Ah8 |

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

### SLL –Shift Left Logical

**Description**:

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

**Instruction Formats**: R2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 40h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### SLT – Set if Less Than

**Description:**

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

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 18h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

### SLTI – Set if Less Than Immediate

**Description:**

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

**Instruction Format:** RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 18h8 |

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

### SLTIL – Set if Less Than Immediate Long

**Description:**

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

**Instruction Format:** RIL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Ra6 | Rt6 | v | 19h8 |

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

### SLEI – Set if Less Than or Equal Immediate

**Description:**

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

**Instruction Format:** RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 18h8 |

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

### SNE – Set if Not Equal

**Description:**

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

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 17h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

### SNEI – Set if Not Equal Immediate

**Description:**

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

Instruction Format: RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 17h8 |

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

### SRL –Shift Right Logical

**Description**:

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

**Instruction Formats**: R2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 41h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### 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: All ALU’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: All ALU’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.*

### WYDENDX – WYDE Index

**Description:**

This instruction searches Ra, which is treated as an array of four wydes, for a wyde value specified by Rb and places the index of the wyde into the target register Rt. If the wyde is not found -1 is placed in the target register. A common use would be to search for a null wyde. The index result may vary from -1 to +3. The index of the first found wyde is returned (closest to zero).

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

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 33 | 32 30 | 29 | 28 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 56h7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rt = Index of (Rb in Ra)

**Exceptions:** none

### WYDENDXI – Wyde Index

**Description:**

This instruction searches Ra, which is treated as an array of four wydes, for a value specified by an immediate value and places the index of the value into the target register Rt. If the wyde is not found -1 is placed in the target register. A common use would be to search for a null wyde. The index result may vary from -1 to +3. The index of the first found wyde is returned (closest to zero).

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

**Instruction Format:** RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 39 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate18..0 | Ra6 | Rt6 | v | 56h8 |

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rt = Index of (Imm8 in Ra)

**Exceptions:** none

### XOR – Bitwise Exclusive 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 |
| 0Ah7 | m3 | z | Tb2 | Rb6 | Ra6 | Rt6 | v | 02h8 |

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

**Exceptions**: none

### XORI – Bitwise Exclusive Or Immediate

**Description**:

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

Instruction Format: RI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate10..0 | Ra6 | Rt6 | v | 0Ah8 |

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

**Integer Instruction Format: RM**

Generates a mask consisting of ‘1’s between the mask begin, Mb6, and mask end, Me6, inclusive. Bitwise exclusive ‘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 flipping bit fields.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 36 | 35 | 3433 | 32 27 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Ah4 | S | ~2 | Me6 | 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

### XORIL – Bitwise Exclusive Or Immediate Long

**Description**:

Perform a bitwise exclusive ‘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 | 4Ah8 |

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

Operation

**Rt = Ra ^ Immediate**

**Vector Operation**

for x = 0 to VL-1

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

else Vt[x] = Vt[x]

**Exceptions**: none

### XORIS – Exclusive Or Immediate Shifted

Description:

Bitwise exclusive ‘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.

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 55 21 | 20 15 | 14 9 | 8 | 7 0 |
| Immediate34..0 | Rt6 | Ra6 | v | 5Ah8 |

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

Rt = Ra ^ (immediate << 32)

**Exceptions:**

**Notes:**

## Load / Store Instructions

### Overview

### Addressing Modes

Load and store instructions have two addressing modes, register indirect with displacement and scaled indexed addressing. There are two formats for register indirect with displacement addressing, differing in the number of bits used to represent a displacement. Store operations have two fewer bits reserved for encoding displacements in instructions.

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.

### 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.

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

#### Register Indirect with Long Displacement Format

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

### Store Formats

Stores have two fewer displacement bits than loads. The scaled indexed format is wider than that of the load.

#### 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 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Tb2 | Rb6 | Ra6 | Disp6 | v | Opcode8 |

#### Register Indirect with Long Displacement Format

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 29 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement21..6 | Tb2 | Rb6 | Ra6 | Disp6 | v | Opcode8 |

#### Scaled Indexed Format

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 42 | 41 39 | 3837 | 3635 | 34 29 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 00h6 | Seg3 | Sc2 | Tc2 | Rc6 | Tb2 | Rb6 | Ra6 | ~6 | v | C0h8 |

### CACHE – Cache Command

**Description:**

This instruction commands the cache controller to perform an operation. Commands are summarized in the command table below. Commands may be issued to both the instruction and data cache at the same time. The address of the cache line to be invalidated is passed in Ra if needed.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3129 | 28 21 | 20 15 | 1412 | 119 | 8 | 7 0 |
| Sg3 | Displacement7..0 | Ra6 | DC3 | IC3 | v | 9Fh8 |

**Commands:**

|  |  |  |
| --- | --- | --- |
| IC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 3 | invline | invalidate line associated with given address |
| 4 | invall | invalidate the entire cache (address is ignored) |
| 5 to 7 |  | reserved |

|  |  |  |
| --- | --- | --- |
| DC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 1 | enable | enable cache (instruction cache is always enabled) |
| 2 | disable | not valid for the instruction cache |
| 3 | invline | invalidate line associated with given address |
| 4 | invall | invalidate the entire cache (address is ignored) |
| 5 to 7 |  | reserved |

Clock Cycles: 3

Execution Units: All ALU’s / Memory

Operation:

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

### CACHEL – Cache Command

CACHE Cmd, d[Rn]

**Description:**

This instruction commands the cache controller to perform an operation. Commands are summarized in the command table below. Commands may be issued to both the instruction and data cache at the same time. The address of the cache line to be invalidated is passed in Ra if needed.

**Instruction Formats**: CACHE

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 4745 | 44 21 | 20 15 | 14 12 | 11 9 | 8 | 7 0 |
| Sg3 | Displacement23..0 | Ra6 | DC3 | IC3 | v | DFh8 |

**Commands:**

|  |  |  |
| --- | --- | --- |
| IC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 3 | invline | invalidate line associated with given address |
| 4 | invall | invalidate the entire cache (address is ignored) |
| 5 to 7 |  | reserved |

|  |  |  |
| --- | --- | --- |
| DC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 1 | enable | enable cache (instruction cache is always enabled) |
| 2 | disable | not valid for the instruction cache |
| 3 | invline | invalidate line associated with given address |
| 4 | invall | invalidate the entire cache (address is ignored) |
| 5 to 7 |  | reserved |

Notes:

### LDB – Load Byte

Description:

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

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

Instruction Format:

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

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory8[Ra+displacement])

**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. For vector instructions Ra is a scalar register.

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

Instruction Format:

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

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory8[Ra+displacement])

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

### LDBU – Load Byte, Unsigned

Description:

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

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

Instruction Format:

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

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory8[Ra+displacement])

**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 | Ra6 | Rt6 | v | D1h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory8[Ra+offset])

**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: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory8[Ra+Rb])

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

### LDBX – Load Byte Indexed

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory8[Ra+Rb])

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

### LDT – Load Tetra

Description:

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

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

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3129 | 28 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement7..0 | Ra6 | Rt6 | v | 84h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory32[Ra+displacement])

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

### LDTL – Load Tetra, Long Address

Description:

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

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

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4745 | 44 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement23..0 | Ra6 | Rt6 | v | D4h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory32[Ra+displacement])

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

### LDTU – Load Tetra Unsigned

Description:

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

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

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3129 | 28 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement7..0 | Ra6 | Rt6 | v | 84h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory32[Ra+displacement])

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

### LDTUL – Load Tetra Unsigned, Long Address

Description:

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

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

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4745 | 44 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement23..0 | Ra6 | Rt6 | v | D5h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory32[Ra+displacement])

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

### LDW – Load Wyde

Description:

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

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

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3129 | 28 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement7..0 | Ra6 | Rt6 | v | 82h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory16[Ra+displacement])

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

### LDWL – Load Wyde, Long Address

Description:

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

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

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4745 | 44 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement23..0 | Ra6 | Rt6 | v | D2h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory16[Ra+displacement])

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

### LDWU – Load Wyde Unsigned

Description:

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

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

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3129 | 28 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement7..0 | Ra6 | Rt6 | v | 83h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory16[Ra+offset])

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

### LDWUL – Load Wyde Unsigned, Long Address

Description:

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

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

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 4745 | 44 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement23..0 | Ra6 | Rt6 | v | D3h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory16[Ra+displacement])

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

### LDWX – Load Wyde Indexed

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 02h7 | Seg3 | Sc2 | Tb2 | Rb6 | Ra6 | Rt6 | v | B0h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = sign extend (memory16[Ra+Rb])

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

### LDWUX – Load Wyde Unsigned Indexed

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 34 | 33 31 | 3029 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 03h7 | Seg3 | Sc2 | Tb2 | Rb6 | Ra6 | Rt6 | v | B0h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

Rt = zero extend (memory16[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 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Tb2 | Rb6 | Ra6 | Disp6 | v | 90h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory8[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 29 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement21..6 | Tb2 | Rb6 | Ra6 | Disp6 | v | E0h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory8[Ra+offset] = Rb[7..0]

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

### STBX – Store 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:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 42 | 41 39 | 3837 | 3635 | 34 29 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 00h6 | Seg3 | Sc2 | Tc2 | Rc6 | Tb2 | Rb6 | Ra6 | ~6 | v | C0h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory8[Ra+Rc\*Sc] = Rb[7:0]

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

### STT – Store Tetra

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3129 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Tb2 | Rb6 | Ra6 | Disp6 | v | 92h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory32[Ra+displacement] = Rb[31..0]

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

### STTL – Store Tetra, Long Addressing

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 29 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement21..6 | Tb2 | Rb6 | Ra6 | Disp6 | v | E0h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory32[Ra+offset] = Rb[31..0]

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

### STTX – Store Tetra Indexed

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 42 | 41 39 | 3837 | 3635 | 34 29 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 02h6 | Seg3 | Sc2 | Tc2 | Rc6 | Tb2 | Rb6 | Ra6 | ~6 | v | C0h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory32[Ra+Rc\*Sc] = Rb[31:0]

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

### STW – Store Wyde

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3129 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Tb2 | Rb6 | Ra6 | Disp6 | v | 91h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory16[Ra+displacement] = Rb[15..0]

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

### STWL – Store Wyde, Long Addressing

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 45 | 44 29 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| Sg3 | Displacement21..6 | Tb2 | Rb6 | Ra6 | Disp6 | v | E0h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory16[Ra+offset] = Rb[15..0]

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

### STWX – Store Wyde Indexed

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 47 42 | 41 39 | 3837 | 3635 | 34 29 | 2827 | 26 21 | 20 15 | 14 9 | 8 | 7 0 |
| 01h6 | Seg3 | Sc2 | Tc2 | Rc6 | Tb2 | Rb6 | Ra6 | ~6 | v | C0h8 |

Clock Cycles: 10 (one memory access)

Execution Units: All ALU’s / Memory

Operation:

memory16[Ra+Rc\*Sc] = Rb[15:0]

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

## Branch / Flow Control Instructions

### Overview

#### Mnemonics

There are two sets of mnemonics for branch instructions. Branch instructions that are IP relative, specify Ca = 7 in the branch instruction and are referred to with a ‘B’ as in BEQ. Using mnemonics that begin with ‘B’ imply the code address register is the instruction pointer, C7. Branch instructions that are relative to other code address registers are referred to as jump instructions and begin with a ‘J’ in the mnemonic.

Instructions that decrement the loop counter begin with a ‘D’ in the mnemonic as in DBEQ standing for decrement and branch if equal.

#### Conditions

Conditional branches branch to the target address only if the condition is true. The condition is determined by the comparison of two general-purpose registers. The 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 conditional 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. A short displacement of 10 bits is suitable for about 90% of branches. However, calling subroutines typically requires more address bits than is available in the short format. For branching to more distant targets the long form of the branch instruction supports a 26-bit displacement constant. Unconditional branches support an even larger displacement target, either 24 or 40 bits. Unconditional branches are used for subroutine calls and typically require a larger range than conditional branches. 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 |

### [D]BBC – Branch if Bit Clear

**Description**:

This instruction branches to the target address if the Rb bit of Ra is clear, 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 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 24h8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 34h8 |

**Operation:**

Lk = next IP

If (Ra.bit[Rb] == 0)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### [D]BBS – Branch if Bit Set

**Description**:

This instruction branches to the target address if the Rb bit of Ra is set, 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 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 25h8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 35h8 |

**Operation:**

Lk = next IP

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

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### [D]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 | 73 | 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 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 36h8 |

**Operation:**

Lk = next IP

If (Ra==Rb)

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

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

### [D]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 | 73 | 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 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 39h8 |

**Operation:**

Lk = next IP

If (Ra>=Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]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 | 73 | 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 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Dh8 |

**Operation:**

Lk = next IP

If (Ra>=Rb)

IP = IP + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]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 | 73 | 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 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Bh8 |

**Operation:**

Lk = next IP

If (Ra > Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]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 | 73 | 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 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Fh8 |

**Operation:**

Lk = next IP

If (Ra > Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]BLE – Branch if Less Than or Equal

**Description**:

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

**Formats Supported**: BR,BRL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2Ah8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Ah8 |

**Operation:**

Lk = next IP

If (Ra <= Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]BLEU – Branch if Less Than or Equal Unsigned

**Description**:

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

**Formats Supported**: BR,BRL

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 2Eh8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 2827 | 26 21 | 20 15 | 1413 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Eh8 |

**Operation:**

Lk = next IP

If (Ra <= Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]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 | 73 | 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 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 38h8 |

**Operation:**

Lk = next IP

If (Ra > Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]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 | 73 | 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 | 73 | Tb2 | Rb6 | Ra6 | Tgt2 | Cn2 | Lk2 | 0 | 3Ch8 |

**Operation:**

Lk = next IP

If (Ra > Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]BNE – Branch if Not Equal

**Description**:

This instruction branches to the target address if the contents of the Ra is not equal to the contents of Rb, otherwise program execution continues with the next instruction. For a further description see [Branch Instructions](#_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 | 27h8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 37h8 |

**Operation:**

Lk = next IP

If (Ra != Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### [D]BRA – Branch Always

**Description**:

This instruction always branches to the target address.

**Formats Supported**: BRA, BRAL

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 28 13 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | 73 | Target16 | Cn2 | Lk2 | 0 | 20h8 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 28 13 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | 73 | Target16 | Cn2 | Lk2 | 0 | 30h8 |

**Operation:**

Lk = next IP

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### [D]BSR – Branch to Subroutine

**Description**:

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

**Formats Supported**: BRA, BRAL

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 28 13 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | 73 | Target16 | Cn2 | Lk2 | 0 | 20h8 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 28 13 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | 73 | Target16 | Cn2 | Lk2 | 0 | 30h8 |

**Operation:**

Lk = next IP

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### [D]JBC – Jump if Bit Clear

**Description**:

This instruction branches to the target address if the Rb bit of Ra is clear, 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 | 24h8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 34h8 |

**Operation:**

Lk = next IP

If (Ra.bit[Rb] == 0)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### [D]JBS – Jump if Bit Set

**Description**:

This instruction branches to the target address if the Rb bit of Ra is set, 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 | 25h8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 35h8 |

**Operation:**

Lk = next IP

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

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### [D]JEQ – Jump if Equal

**Description**:

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

### [D]JGE – Jump if Greater Than or Equal

**Description**:

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

### [D]JGEU – Jump if Greater Than or Equal Unsigned

**Description**:

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

### [D]JGT – Jump if Greater Than

**Description**:

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

### [D]JGTU – Jump if Greater Than Unsigned

**Description**:

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

### [D]JLE – Jump if Less Than or Equal

**Description**:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see [Branch Instructions](#_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 | 2Ah8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 3Ah8 |

**Operation:**

Lk = next IP

If (Ra <= Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]JLEU – Jump if Less Than or Equal Unsigned

**Description**:

This instruction branches to the target address if the contents of the Ra is less than or equal to that of Rb, otherwise program execution continues with the next instruction. The values are treated as unsigned integers. For a further description see [Branch Instructions](#_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 | 2Eh8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 3Eh8 |

**Operation:**

Lk = next IP

If (Ra <= Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions:** none

### [D]JLT – Jump if Less Than

**Description**:

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

### [D]JLTU – Jump if Less Than Unsigned

**Description**:

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

### [D]JNE – Jump if Not Equal

**Description**:

This instruction branches to the target address if the contents of the Ra is not equal to the contents of Rb, otherwise program execution continues with the next instruction. For a further description see [Branch Instructions](#_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 | 27h8 |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 37h8 |

**Operation:**

Lk = next IP

If (Ra != Rb)

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### [D]JMP – Jump

**Description**:

This instruction always jumps to the target address.

**Formats Supported**: BRA, BRAL

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 28 13 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Target16 | Cn2 | 02 | 0 | 20h8 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 28 13 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Target16 | Cn2 | 02 | 0 | 30h8 |

**Operation:**

Lk = next IP

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### [D]JSR – Jump to Subroutine

**Description**:

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

**Formats Supported**: BRA, BRAL

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 32 | 31 29 | 28 13 | 1211 | 10 9 | 8 | 7 0 |
| Target8 | Ca3 | Target16 | Cn2 | Lk2 | 0 | 20h8 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 55 32 | 31 29 | 28 13 | 1211 | 10 9 | 8 | 7 0 |
| Target24 | Ca3 | Target16 | Cn2 | Lk2 | 0 | 30h8 |

**Operation:**

Lk = next IP

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### NOP – No Operation

Description:

This instruction does not do anything.

**Integer Instruction Format:**

|  |  |  |
| --- | --- | --- |
| 15 9 | 8 | 7 0 |
| ~7 | v | F1h8 |

Operation:

**Vector Operation**

Execution Units: I

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### RTS – Return from Subroutine

**Description**:

This instruction returns from a subroutine by transferring program execution to the address calculated as the sum of a link register and a constant.

**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:**

|  |  |  |
| --- | --- | --- |
| 15 9 | 8 | 7 0 |
| ~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 |
| Mw6 | Mb6 | Vmt3 | 0 | 53h8 |

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

### VMSLL – Vector Mask Shift Left Logical

**Description:**

Shift a vector mask register to the left up to 31 bits.

**Instruction Format: VMR2**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 23 20 | 19 15 | 14 12 | 11 9 | 8 | 7 0 |
| 0Eh4 | Amount5 | 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 | SUBFI | MULI | {SYS} | ANDI | ORI | EORI |  | ADCI | SBCFI | MULUI | {CSR} |
| 1x |  | {R1R} | {R2R} | {R3R} | ADDIQ | MULFI | SEQI | SNEI | SLTI | SLTIL | SGTIL | SGTI | SLTUI | SLTUIL | SGTUIL | SGTUI |
| 2x | BRA |  |  |  | BBC | BBS | BEQ | BNE | BLT | BGE | BLE | BGT | BLTU | BGEU | BLEU | BGTU |
| 3x | BRA |  |  |  | BBC | BBS | BEQ | BNE | BLT | BGE | BLE | BGT | BLTU | BGEU | BLEU | BGTU |
| 4x | DIVI | DIVUI | DIVSUI |  | ADDIL | CHKI | MULIL |  | ANDIL | ORIL | EORIL |  |  |  | MULUI |  |
| 5x | CMPI | MLO | {VM} | VMFILL |  | BYTNDX | WYDENDX | UTF21NDX | ANDIS | ORIS | EORIS |  |  | CHKXI |  |  |
| 6x | CMPIL | {FLT1} | {FLT2} | {FLT3} |  | {DFLT1} | {DFLT2} | {DFLT3} |  | {PST1} | {PST2} | {PST3} |  |  |  | MUX |
| 7x | CMPIS | {FLT1R} | {FLT2R} | {FLT3R} |  | {DFLT1} | {DFLT2} | {DFLT3} |  | {PST1R} | {PST2R} | {PST3R} |  |  |  |  |
| 8x | LDB | LDBU | LDW | LDWU | LDT | LDTU | LDO |  |  | LLA |  | LVOAR | SWCR | JSRI | LWS | LCL |
| 9x | STB | STW | STT | STO |  |  | STI | CAS | STSET | STMOV | STCMP | STFND |  |  | SWS | CACHE |
| Ax |  |  |  |  | SYS | SYS | INT |  |  |  | {bitfld} | MOVS |  |  |  |  |
| Bx | {LDxX} |  |  |  |  |  |  | JSRIX | LLAX |  |  |  |  | LV | LVWS |  |
| Cx | {STxX} |  |  |  |  |  | STIX |  | PUSH | PEA | POP | LINK | UN  LINK | SV | SVWS |  |
| Dx | LDBL | LDBUL | LDWL | LDWUL | LDTL | LDTUL | LDOL |  |  |  |  |  |  |  |  | CACHEL |
| Ex | STBL | STWL | STTL | STOL |  |  |  |  |  |  |  |  |  |  |  |  |
| Fx |  | NOP | RTS | 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 |  |  |  |  |  |  |  |  |

## {R2} Integer Dyadic Register Ops – Func7

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0000 | NAND | NOR | XNOR | ORC | ADD | SUB | MUL |  |
| 0001 | AND | OR | XOR | ANDC | ADC | SBC | MULU | MULH |
| 0010 | DIV | DIVU | DIVSU |  |  | MULF | MULSU | PERM |
| 0011 | DIF |  |  | WYDNDX |  | MULSUH | MULUH | MYST |
| 0100 |  |  |  | U21NDX |  |  |  |  |
| 0101 | MIN | MAX | CMP | CMPU | BIT |  |  |  |
| 0110 | BMM.or | BMM.xor | BMM | BMM | PLOT |  |  |  |
| 0111 | VSLLV | VSLRV | VEX | VEINS |  |  | RW\_COEFF |  |
| 1000 | SLL | SRL | SRA | ROL | ROR |  | SLLP | SRLP |
| 1001 | SLLI | SRLI | SRAI | ROLI | RORI |  |  |  |
| 1010 |  |  |  |  |  |  |  |  |