Table of Contents

[Overview 2](#_Toc36143173)

[Design Objectives 2](#_Toc36143174)

[Thor2020 Differences from Thor Original 3](#_Toc36143175)

[Programming Model 4](#_Toc36143176)

[General Registers 4](#_Toc36143177)

[Code Address Registers 5](#_Toc36143178)

[Predicates 6](#_Toc36143179)

[Predicate Conditions 6](#_Toc36143180)

[Compiler Usage 7](#_Toc36143181)

[Operating Levels 8](#_Toc36143182)

[Instructions 8](#_Toc36143183)

[Instruction Bundles 8](#_Toc36143184)

[Instruction Pointer 8](#_Toc36143185)

[Instruction Set Summary 8](#_Toc36143186)

[Branch Instructions 8](#_Toc36143187)

[Loops 8](#_Toc36143188)

[Detailed Instruction Set Description 11](#_Toc36143189)

[Control Flow Instructions 11](#_Toc36143190)

[BRK –Break 11](#_Toc36143191)

[BSR - Branch to Subroutine 12](#_Toc36143192)

[JML - Jump to Long Address 13](#_Toc36143193)

[JMP - Jump to Address 14](#_Toc36143194)

[JSR - Jump to Subroutine 15](#_Toc36143195)

[LOOP – Loop Branch 16](#_Toc36143196)

[RTS – Return from Subroutine 17](#_Toc36143197)

[Arithmetic / Logical 19](#_Toc36143198)

[ADD - Addition 19](#_Toc36143199)

[AND – Bitwise ‘And’ 20](#_Toc36143200)

[ANDCM – And with Compliment 20](#_Toc36143201)

[BIT – Test bits 21](#_Toc36143202)

[CMP - Compare 22](#_Toc36143203)

[DIV - Divide 23](#_Toc36143204)

[DIVU – Unsigned Divide 24](#_Toc36143205)

[ENOR – Exclusive Nor 25](#_Toc36143206)

[EOR – Bitwise Exclusive Or 26](#_Toc36143207)

[MUL - Multiply 27](#_Toc36143208)

[MULU – Unsigned Multiply 27](#_Toc36143209)

[SUB - Subtract 28](#_Toc36143210)

[SUBF – Subtract from Immediate 28](#_Toc36143211)

[Memory Operations 29](#_Toc36143212)

[LDB – Load Byte (8 bits) 29](#_Toc36143213)

[LDD – Load Double Word (64 bits) 30](#_Toc36143214)

[STD – Store Double Word (64-bits) 31](#_Toc36143215)

[MOVI – Move Immediate 32](#_Toc36143216)

# Overview

Thor2020 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. Thor2020 uses fixed length instructions three packed into a 128-bit bundle and handles 8, 16, 32, and 64-bit data within a 64-bit address space.

## 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. |
| * 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.

## Thor2020 Differences from Thor Original

Instructions are fixed 41-bit length and packed into instruction bundles. Thor’s instruction set was variable length and byte aligned. The fixed instruction size of Thor2020 simplifies fetch logic removing instruction alignment requirements and the need to determine the instruction length.

Opcodes are seven bits instead of eight bits. About ½ of the opcode space of Thor is unassigned and instruction bits are valuable. Thor’s instruction set could be compressed to fit into seven bits.

Predicate registers are only two bits wide and record a +1,0, or -1 compare result. There are only eight predicate conditions testable. Reducing the number of tests reduces the number of encoding bits required to three from four. Reducing the predicate register size reduces hardware requirements and simplifies the design.

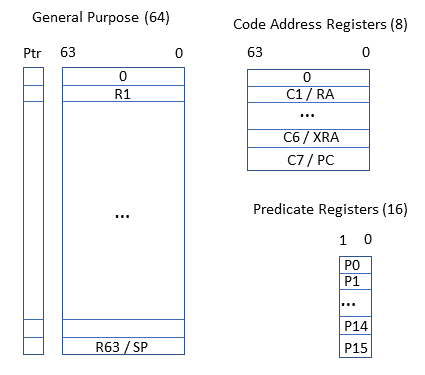
There is an auto-record bit in most instructions allowing the result status (plus, zero, or minus) to be placed into predicate register #1 for integer or predicate register #2 for floating-point operations. This allows some code compression by removing the need for a compare operation in some circumstances.

## Programming Model

### General Registers

There are 64 general-purpose registers. General purpose registers are 64 bits wide. The general registers may hold integer or floating-point values.

Register #0 is always zero or +0.0. Predicate register #0 is always 0.



### Code Address Registers

The processor contains eight code address registers (C0-C7). Several of the registers are reserved for predefined purposes. A code address register is used in the formation and storage of code addresses.

|  |  |  |
| --- | --- | --- |
| Reg # |  | Usage |
| 0 | Always Zero | Absolute address formation |
| 1 |  | Subroutine return address |
| 2 |  | This register is available for general use. |
| 3 |  | This register is available for general use. |
| 4 |  | This register is available for general use. |
| 5 | Catch Link Register | Used by the compiler to link to try/catch handlers. |
| 6 | Exceptioned PC | This register is automatically set during a hardware interrupt or exception. |
| 7 | Instruction Pointer | Relative address formation. This register is read-only. |

Code address registers may be used to point to a block of code from which the JMP instruction can index into with its 24-bit offset. For instance, a register may contain a pointer to a class method jump list; the JMP instruction can then index into this list in order to invoke a method.

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

|  |  |  |
| --- | --- | --- |
| 63 32 | 31 0 | |
| Instruction Bank | Instruction Pointer | 04 |

The instruction pointer is special in that it is always incrementing by the size of an instruction bundle fetched as a program runs. Program code is 128-bit aligned. To improve performance only the low order 32 bits of the program counter increment. The entire instruction pointer may be loaded with a jump instruction. If the upper eight bits of the instruction pointer / bank are all ones, then segmentation with the code segment is ignored.

## Predicates

The processor features predicated execution of all instructions. Whether or not an instruction is executed depends on the contents of a predicate register and the predicate condition specified in the predicate byte. There are 32, 2-bit predicate registers. The predicate registers hold the value +1,0, or -1. The value is set as the result of a compare operation. If the first operand is greater than the second then the value +1 is stored in the predicate register. If the first operand is equal to the second then the value 0 is stored in the predicate register. Otherwise if the first operand is less than the second then the value -1 is stored in the predicate register.

All instructions are executed conditionally determined by the value of a predicate register.

### Predicate Conditions

There are eight predicate conditions that can be checked, summarized in the table below.

|  |  |  |  |
| --- | --- | --- | --- |
| Cond. |  | Test |  |
| 0 | PUN | p = -2 | unordered (floating point) |
| 1 |  |  | reserved |
| 2 | PZ | p = 0 | Instruction executes if the predicate register zero |
| 3 | PNZ | p = 1 or p = -1 | Instruction executes if the predicate register is non-zero |
| 4 | PLE | p = 0 or p = -1 | predicate less than or equal to zero |
| 5 | PGT | p = 1 | predicate greater than zero |
| 6 | PGE | p = 0 or p = 1 | predicate greater or equal to zero |
| 7 | PLT | p = -1 | predicate less than zero |

### Compiler Usage

The compiler uses predicate register #15 to conditionally move TRUE / FALSE values to a register when evaluating a logical operation.

Predicate registers beginning with P1 and incrementing are applied for use as the control flow nesting level increases. The compiler does not support control flow nesting more than 13 levels in a single subroutine. Predicate registers beginning with P14 and decrementing are used in the evaluation of the hook operator. Care must be taken such that the number of predicate registers in use does not exceed the number available.

|  |  |
| --- | --- |
| Pred. | Usage |
| P0 | always 1 – allows unconditional instruction execution. This predicate is automatically implied by the assembler if no other predicate is specified. |
| P1 | control flow nesting level 1 |
| P2 | control flow nesting level 2 |
| … |  |
| Pn | control flow nesting level n (n not to exceed 29) |
| … |  |
| P12 | third hook operator in an expression |
| P13 | second hook operator in an expression |
| P14 | first hook operator in an expression |
| P15 | conditionally moves TRUE/FALSE for logical expressions |

# Operating Levels

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

A subset of instructions is limited to the machine level.

# Instructions

## Instruction Bundles

Thor2020 uses instruction bundles. Three 41-bit instructions are placed in a 128-bit bundle. The remaining bits of the bundle are for control. Instructions are effectively executed in the order slot0, slot1, then slot2. Multiple instructions may be executed in the same clock cycle. Stop bits in the control part of the bundle indicate when multiple cycles are required to resolve dependencies.

|  |  |  |  |
| --- | --- | --- | --- |
| 127 123 | 122 82 | 81 41 | 40 0 |
| Control | Slot2 | Slot1 | Slot0 |

|  |  |
| --- | --- |
| Control | Description |
| 123 | stop after instruction zero |
| 124 | stop after instruction one |
| 125 | stop after instruction two |
| 126 | reserved |
| 127 | reserved |

## Instruction Pointer

The instruction pointer is always bundle (128-bit) aligned. Jumps and branches target a 128-bit aligned address. The first slot of a bundle is always targeted. Since branch targets must be bundle aligned it may be necessary for the assembler / compiler to output NOP instructions.

# Instruction Set Summary

## Branch Instructions

The core uses predicated a jump instruction to perform branching. Branches may be relative to the address of the current instruction by specifying code address register seven (the instruction pointer) as a reference. A single jump instruction may be used to implement branching on multiple complex conditions when combined with a predicate. The jump instruction supports a 24-bit displacement field.

## Loops

There is a loop instruction and corresponding loop count register to support counted loops. The loop instruction is predicted as always taken and does not consume room in the branch history table. Like a branch instruction a loop instruction takes place at the fetch stage of the core.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | |  | | |  |  |  | | | | | Opcode7 | | | Pred5 | Pcn3 | Rc1 |  |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 47 | | | Pr4 | Pc3 | Rc | ADD # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 57 | | | Pr4 | Pc3 | Rc | SUBF # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 87 | | | Pr4 | Pc3 | Rc | AND # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 97 | | | Pr4 | Pc3 | Rc | OR # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 107 | | | Pr4 | Pc3 | Rc | EOR # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 247 | | | Pr4 | Pc3 | Rc | MUL # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 257 | | | Pr4 | Pc3 | Rc | MULU # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 267 | | | Pr4 | Pc3 | Rc | DIV # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 277 | | | Pr4 | Pc3 | Rc | DIVU # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 287 | | | Pr4 | Pc3 | Rc | DIVR # |
| Immediate14 | | | | | | | Ra6 | Rt6 | | | | | 297 | | | Pr4 | Pc3 | Rc | DIVRU # |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 53 | Func4 | | Pr4 | Pc3 | Rc | LDx |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 807 | | | Pr4 | Pc3 | Rc | LDB |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 817 | | | Pr4 | Pc3 | Rc | LDH |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 827 | | | Pr4 | Pc3 | Rc | LDW |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 837 | | | Pr4 | Pc3 | Rc | LDD |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 847 | | | Pr4 | Pc3 | Rc | LDBU |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 857 | | | Pr4 | Pc3 | Rc | LDHU |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 867 | | | Pr4 | Pc3 | Rc | LDWU |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 897 | | | Pr4 | Pc3 | Rc | LDFS |
| Disp13..0 | | | | | | | Ra6 | Rt6 | | | | | 907 | | | Pr4 | Pc3 | Rc | LDFD |
| Disp13..0 | | | | | | | Ra6 | Rs6 | | | | | 63 | Func4 | | Pr4 | Pc3 | Rc | STx |
| Disp13..0 | | | | | | | Ra6 | Rs6 | | | | | 967 | | | Pr4 | Pc3 | ~ | STB |
| Disp13..0 | | | | | | | Ra6 | Rs6 | | | | | 977 | | | Pr4 | Pc3 | ~ | STH |
| Disp13..0 | | | | | | | Ra6 | Rs6 | | | | | 987 | | | Pr4 | Pc3 | ~ | STW |
| Disp13..0 | | | | | | | Ra6 | Rs6 | | | | | 997 | | | Pr4 | Pc3 | ~ | STD |
| Disp13..0 | | | | | | | Ra6 | Rs6 | | | | | 1017 | | | Pr4 | Pc3 | ~ | STFS |
| Immediate14 | | | | | | | Ra6 | 0 | | Pt5 | | | 67 | | | Pr4 | Pc3 | ~ | CMPU # |
| Immediate14 | | | | | | | Ra6 | 1 | | Pt5 | | | 67 | | | Pr4 | Pc3 | ~ | CMP # |
| Immediate14 | | | | | | | Ra6 | ~ | | Pt5 | | | 77 | | | Pr4 | Pc3 | ~ | BIT # |
|  | | | | | | |  |  | | | | |  | | |  |  |  |  |
|  | | |  | | |  |  |  | | | | |  | | |  |  |  |  |
| 47 | | | | Oe | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | ADD |
| 57 | | | | Oe | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | SUB |
| 87 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | AND |
| 97 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | OR |
| 107 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | EOR |
| 117 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | ANDCM |
| 127 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | NAND |
| 137 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | NOR |
| 147 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | ENOR |
| 157 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | ORCM |
| 167 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | SHL |
| 177 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | SHR |
| 187 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | ASR |
| 207 | | | | ~ | | imm6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | SHL # |
| 217 | | | | ~ | | imm6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | SHR # |
| 227 | | | | ~ | | imm6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | ASR # |
| 247 | | | | Oe | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | MUL |
| 257 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | MULU |
| 267 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | DIV |
| 277 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | DIVU |
| 287 | | | | ~ | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | MOD |
| 327 | | | | ~ | | Spr12 | | Rt6 | | | | | 27 | | | Pr4 | Pc3 | Rc | MFSPR |
| 337 | | | | ~ | | Spr12 | | Rs6 | | | | | 27 | | | Pr4 | Pc3 | Rc | MTSPR |
| 67 | | | | ~ | | Rb6 | Ra6 | 0 | | Pt5 | | | 27 | | | Pr4 | Pc3 | ~ | CMPU |
| 67 | | | | ~ | | Rb6 | Ra6 | 1 | | Pt5 | | | 27 | | | Pr4 | Pc3 | ~ | CMP |
| 77 | | | | ~ | | Rb6 | Ra6 | ~ | | Pt5 | | | 27 | | | Pr4 | Pc3 | ~ | BIT |
| 02 | Me6 | | | | | Mb6 | Ra6 | Rt6 | | | | | 567 | | | Pr4 | Pc3 | Rc | EXTU |
| 12 | Me6 | | | | | Mb6 | Ra6 | Rt6 | | | | | 567 | | | Pr4 | Pc3 | Rc | EXT |
| 22 | Me6 | | | | | Mb6 | Ra6 | Rt6 | | | | | 567 | | | Pr4 | Pc3 | Rc | DEP |
| 32 | Me6 | | | | | Mb6 | Imm6 | Rt6 | | | | | 567 | | | Pr4 | Pc3 | Rc | DEP # |
| r21 | Rc6 | | | | | Rb6 | Ra6 | Rt6 | | | | | 566 | | r0 | Pr5 | Pc3 | Rc | FMADD |
| r21 | Rc6 | | | | | Rb6 | Ra6 | Rt6 | | | | | 576 | | r0 | Pr5 | Pc3 | Rc | FMSUB |
| r21 | Rc6 | | | | | Rb6 | Ra6 | Rt6 | | | | | 586 | | r0 | Pr5 | Pc3 | Rc | FNMADD |
| r21 | Rc6 | | | | | Rb6 | Ra6 | Rt6 | | | | | 596 | | r0 | Pr5 | Pc3 | Rc | FNMSUB |
| 02 | Rc6 | | | | | Rb6 | Ra6 | Rt6 | | | | | 1207 | | | Pr5 | Pc3 | Rc | FMIN3 |
| 12 | Rc6 | | | | | Rb6 | Ra6 | Rt6 | | | | | 1207 | | | Pr5 | Pc3 | Rc | FMAX3 |
| 44 | | rm3 | | | ~ | Rb6 | Ra6 | Rt6 | | | | | 1217 | | | Pr5 | Pc3 | Rc | FADD |
| 54 | | rm3 | | | ~ | Rb6 | Ra6 | Rt6 | | | | | 1217 | | | Pr5 | Pc3 | Rc | FSUB |
| 64 | | rm3 | | | ~ | Rb6 | Ra6 | Rt6 | | | | | 1217 | | | Pr5 | Pc3 | Rc | FMUL |
| 74 | | rm3 | | | ~ | Rb6 | Ra6 | Rt6 | | | | | 1217 | | | Pr5 | Pc3 | Rc | FDIV |
| 84 | | 03 | | | ~ | Rb6 | Ra6 | Rt6 | | | | | 1217 | | | Pr5 | Pc3 | Rc | FSGNJ |
| 84 | | 13 | | | ~ | Rb6 | Ra6 | Rt6 | | | | | 1217 | | | Pr5 | Pc3 | Rc | FSGNJN |
| 84 | | 23 | | | ~ | Rb6 | Ra6 | Rt6 | | | | | 1217 | | | Pr5 | Pc3 | Rc | FSGNJX |
| 797 | | | | ~ | | Imm12 | | Ca3 | | | ~3 | | 27 | | | Pr5 | Pc3 | ~ | RTS |
| 807 | | | | Sc | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr5 | Pc3 | Rc | LDB |
| 817 | | | | Sc | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr5 | Pc3 | Rc | LDH |
| 827 | | | | Sc | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr5 | Pc3 | Rc | LDW |
| 837 | | | | Sc | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr5 | Pc3 | Rc | LDD |
| 847 | | | | Sc | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr5 | Pc3 | Rc | LDBU |
| 857 | | | | Sc | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr5 | Pc3 | Rc | LDHU |
| 867 | | | | Sc | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr5 | Pc3 | Rc | LDWU |
| 897 | | | | Sc | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr5 | Pc3 | Rc | LDFS |
| 907 | | | | Sc | | Rb6 | Ra6 | Rt6 | | | | | 27 | | | Pr5 | Pc3 | Rc | LDFD |
| 967 | | | | Sc | | Rb6 | Ra6 | Rs6 | | | | | 27 | | | Pr5 | Pc3 | ~ | STB |
| 977 | | | | Sc | | Rb6 | Ra6 | Rs6 | | | | | 27 | | | Pr5 | Pc3 | ~ | STH |
| 987 | | | | Sc | | Rb6 | Ra6 | Rs6 | | | | | 27 | | | Pr5 | Pc3 | ~ | STW |
| 997 | | | | Sc | | Rb6 | Ra6 | Rs6 | | | | | 27 | | | Pr5 | Pc3 | ~ | STD |
| 1017 | | | | Sc | | Rb6 | Ra6 | Rs6 | | | | | 27 | | | Pr5 | Pc3 | ~ | STFS |
| Immediate22..3 | | | | | | | | Rt6 | | | | | 84 | I2..0 | | Pr4 | Pc3 | Rc | MOVI |
| Displacement23..4 | | | | | | | | | Ca3 | | | Ct3 | 777 | | | Pr4 | Pc3 | ~ | LOOP |
| Displacement23..4 | | | | | | | | | Ca3 | | | Ct3 | 787 | | | Pr4 | Pc3 | ~ | JMP |
| Displacement23..4 | | | | | | | | | Ca3 | | | Ct3 | 797 | | | Pr4 | Pc3 | ~ | JML |

# Detailed Instruction Set Description

## Control Flow Instructions

### BRK –Break

**Description:**

The Break exception is executed. The core will be switched to kernel mode. The cause code register is set to four. The program counter is reset to $FFFFFFFFFFFC0000 and instructions begin executing.

**Instruction Format: BRK**

**Operation:**

CAUSE = 40h | Const4

OLS = OLS << 2

DLS = DLS << 2

XL = IP + 1

IP = $FFFFFFFFFFFC0000

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

### BSR - Branch to Subroutine

**Description:**

This is an alternate mnemonic for the JMP instruction. A jump is made to the sum of the sign extended displacement supplied in the displacement field of the instruction and the specified code address register Cr. Cr is specified as C7, which reflects the value of the instruction pointer. The result is an instruction pointer relative branch.

The subroutine return address is stored in a code address register specified in the Ct field of the instruction. Typically, code address register #1 is used to store the return address. If not specified code address register #1 will be assumed by the assembler.

**Instruction Formats: JMP**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement23..4 | 73 | 13 | 787 | Pr4 | Pc4 | JMP |

**Execution Units**: Branch

**Clock Cycles:** 0.33

**Exceptions:** none

### JML - Jump to Long Address

**Description:**

A jump is made to the sum of the sign extended displacement supplied in the displacement field of the instruction and the specified code address register Ca. An absolute address may be jumped to by specifying C0 as the code address register for Ca.

A subroutine return address may be stored in a code address register specified in the Ct field of the instruction. Typically, code address register #1 is used.

This instruction is unusual in that it occupies the first two slots of a bundle. A full 64-bit address may be formed.

**Instruction Format**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 127 123 | 122 82 | 81 41 | 40 0 | |
| Control | Slot2 | Address63..24 | Address23..4 | Other |

**Clock cycles**: 0.66 (two slots are occupied).

**Execution Unit:** FCU

**Operation:**

C[t] = ip + 16

ip = C[n] + offset

**Exceptions**: none

### JMP - Jump to Address

**Description:**

A jump is made to the sum of the sign extended displacement supplied in the displacement field of the instruction and the specified code address register Ca. The target address must be bundle (128-bit) aligned.

A subroutine return address may be stored in a code address register specified in the Ct field of the instruction. Typically, code address register #1 is used.

**Instruction Formats: JMP**

**Clock Cycles:** 0.33

**Execution Units:** All ALU’s

**Operation:**

Cr[t] = pc

pc = Cr[n] + displacement

**Exceptions**: none

### JSR - Jump to Subroutine

**Description:**

A jump is made to the sum of the sign extended displacement supplied in the displacement field of the instruction and the specified code address register Ca. Normally for JSR the code address register is assumed to be C0.

The subroutine return address is stored in a code address register specified in the Ct field of the instruction. Typically, code address register #1 is used.

**Instruction Formats: JMP**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Displacement23..4 | 03 | 13 | 787 | Pr4 | Pc4 | JMP |

**Clock Cycles:** 0.33 to 1.0

**Execution Unit:** All ALU’s

**Operation:**

Ct[t] = ip

ip = Ca[n] + displacement

**Exceptions:** none

### LOOP – Loop Branch

**Description:**

A branch is made if the loop count register is non-zero. The loop count register is decremented by this instruction. The predicate condition must also be met. The loop branch is predicted as always taken and does not consume room in the branch predication tables. The target address is the sum of a code address register and displacement specified in the instruction. The target address must be bundle (128-bit) aligned.

**Instruction Format: JMP**

**Clock Cycles:** 0.33 to 1.0

**Execution Units:** Branch

**Operation:**

If LC <> 0

IP <= Ca + displacement

LC = LC – 1

### RTS – Return from Subroutine

**Description:**

This is an alternate mnemonic for the JMP instruction where the code address register used is assumed to be C1 (the default return address).

The instruction pointer is loaded with the value contained in the specified code address register plus a displacement constant specified in the instruction. This allows the return instruction to return a few bytes past the usual return address. This is used to allow static parameters to be passed to the subroutine in inline code.

**Instruction Formats: JMP**

**Execution Unit:** Branch

**Operation:**

PC = Ca[N] +Imm

**Exceptions:** none

## Arithmetic / Logical

### ADD - Addition

**Description:**

Add two registers or a register and an immediate value and place the sum in the target register. The register form of the instruction may cause an overflow exception is enabled.

**Instruction Format: R2, RI**

**Clock Cycles:** 0.33

**Execution Unit:** AllALU’s

**Operation:**

Rt = Ra + Rb

Rt = Ra + Imm

**Exceptions**: integer overflow

### AND – Bitwise ‘And’

**Description:**

Bitwise and’s two registers or a register and an immediate value and places the result in a target register.

**Instruction Format: R2, RI**

**Clock Cycles:** 0.33

**Execution Unit:** AllALU’s

**Operation:**

Rt = Ra & Rb

Rt = Ra & Imm

**Exceptions:** none

### ANDCM – And with Compliment

**Description:**

Bitwise and’s a register Ra with the compliment of register Rb and places the result in a target register. There is no immediate form for this instruction.

**Instruction Format: R2**

**Clock Cycles:** 0.33

**Execution Unit:** AllALU’s

**Operation:**

Rt = Ra & ~Rb

**Exceptions:** none

### BIT – Test bits

**Description:**

Logically and’s register and an immediate value or two registers and places the result in a predicate register. If the result of the ‘and’ operation is zero the predicate register’s zero flag is set, otherwise it is cleared. If the result is negative the predicate’s less than flag is set, otherwise it is cleared.

**Instruction Format: R2, RI**

**Clock Cycles:** 0.33

**Execution Unit:** All ALU’s

**Operation:**

Pt = flag results( Ra & immediate)

Pt = flag results (Ra & Rb)

**Predicate Results:**

|  |  |
| --- | --- |
| Predicate flag | Setting |
| eq | set if result is zero |
| lt | set if result is negative |
| ltu | set if result is odd (bit 0 is set) |
|  |  |

**Exceptions:** none

### CMP - Compare

**Description:**

The compare instruction compares two registers or a register and an immediate value and sets the flags in the target predicate register as a result.

**Instruction Format: R2, RI**

**Clock Cycles:** 0.33

**Execution Unit:** AllALU’s

**Operation:**

if signed Opa < signed Opb

P = -1

else if Opa = Opb

P = 0

else

P = 1

**Exceptions:** none

### CMPU – Unsigned Compare

**Description:**

The compare instruction compares two registers or a register and an immediate value and sets the flags in the target predicate register as a result.

**Instruction Format: R2, RI**

**Clock Cycles:** 0.33

**Execution Unit:** AllALU’s

**Operation:**

if unsigned Opa < unsigned Opb

P = -1

else if Opa = Opb

P = 0

else

P = 1

**Exceptions:** none

### DIV - Divide

**Description:**

Performs a signed division of two registers or a register and an immediate value and places the quotient in the target register. This instruction may cause an overflow or divide by zero exception.

**Instruction Format: R2, RI**

**Clock Cycles:** 68

**Execution Units:** ALU

**Operation:**

Rt = Ra / Rb

Rt = Ra / #imm

**Exceptions**: divide by zero

### DIVU – Unsigned Divide

**Description:**

Performs an unsigned division of two registers or a register and an immediate value and places the quotient in the target register. This instruction will not cause an overflow or divide by zero exception.

**Instruction Format: R2, RI**

**Clock Cycles:** 68

**Execution Units:** ALU

**Operation:**

Rt = Ra / Rb

Rt = Ra / #imm

**Exceptions:** none

### ENOR – Exclusive Nor

**Description:**

Bitwise exclusive or register with register and place inverted result in target register. There is no immediate form of this instruction.

**Instruction Format: R2**

**Clock Cycles:** 0.33

**Execution Unit:** All ALU’s

**Operation:**

Rt = ~(Ra ^ Rb)

**Exceptions:** none

### EOR – Bitwise Exclusive Or

**Description:**

Bitwise exclusive or register with register or register with immediate and place result in target register.

**Instruction Format: R2, RI**

**Clock Cycles:** 0.33

**Execution Unit:** All ALU’s

**Operation:**

Rt = Ra ^ Rb

**Exceptions:** none

### MUL - Multiply

**Description:**

Performs a signed multiply of two registers or a register and an immediate value and places the product in the target register.

**Instruction Format: R2, RI**

**Clock Cycles:** 20

**Execution Unit:** ALU

**Operation:**

Rt = Ra \* Rb

### MULU – Unsigned Multiply

**Description:**

Performs a unsigned multiply of two registers or a register and an immediate value and places the product in the target register.

**Instruction Format: R2, RI**

**Clock Cycles:** 20

**Execution Unit:** ALU

**Operation:**

Rt = Ra \* Rb

### OR – Bitwise ‘Or’

**Description:**

Bitwise inclusively or two registers or a register and an immediate value and place the result in the target register.

**Instruction Format: R2, RI**

**Clock Cycles:** 0.33

**Execution Units:** ALU

**Operation:**

Rt = Ra | Rb

Rt = Ra | #imm

**Exceptions:** none

### ORCM – Or with Compliment

**Description:**

Bitwise inclusively or register Ra and the compliment of register Rb and place the result in the target register. There is no immediate form for this instruction.

**Instruction Format: R2**

**Clock Cycles:** 0.33

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra | ~Rb

**Exceptions:** none

### SHL – Shift Left

**Description:**

Shift register Ra left by Rb bits or an immediate value and place result into register Rt. A zero is shifted into the least significant bit.

**Instruction Format: R2, RSI**

**Clock Cycles:** 0.33

**Execution Units:** ALU

**Operation:**

Rt = Ra << Rb

**Exceptions:** none

### SHR – Shift Right

**Description:**

Shift register Ra right by Rb bits or an immediate value and place result in register Rt. Zeros are shifted into the most significant bits.

**Instruction Format: R2, RSI**

**Clock Cycles:** 0.33

**Execution Units:** ALU

**Operation:**

Rt = Ra >> Rb

**Exceptions:** none

### SUB - Subtract

**Description:**

This instruction subtracts one register from another and places the result into a third register. There is no immediate form for this instruction, use ADD # instead.

**Instruction Format: R2**

**Clock Cycles:** 0.33

**Execution Unit:** All ALU’s

**Operation:**

Rt = Ra - Rb

### SUBF – Subtract from Immediate

**Description:**

This instruction subtracts a register from an immediate value and places the result into a target register. There is no register form for this instruction, use SUB (swap operands) instead.

**Instruction Format: R2**

**Clock Cycles:** 0.33

**Execution Unit:** All ALU’s

**Operation:**

Rt = Imm - Ra

## Bit Field

### CLR – Bit-field Clear

**Description:**

This is an alternate mnemonic for the DEP instruction where the value deposited is zero. Sets the bits to zero in the target register located between the mask begin (mb) and mask end (me) bits.

**Instruction Format: BF3**

**Clock Cycles:** 0.33

**Execution Units:** ALU

**Exceptions:** none

### DEP – Deposit (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 or from a six-bit immediate constant. Note that the target register is used as a source operand.

**Instruction Format: BF3**

**Clock Cycles:** 0.33

**Execution Units:** ALU

**Exceptions:** none

### EXT – Bit-field Extract

**Description:**

Extracts a bit-field from register Ra located between the mask begin (mb) and mask end (me) bits and places the sign extended result into the target register.

**Instruction Format: BF3**

**Clock Cycles:** 0.33

**Execution Units:** ALU

**Exceptions:** none

### EXTU – Bit-field Extract Unsigned

**Description:**

Extracts a bit-field from register Ra located between the mask begin (mb) and mask end (me) bits and places the zero extended result into the target register.

**Instruction Format: BF3**

**Clock Cycles:** 0.33

**Execution Units:** ALU

**Exceptions:** none

## Memory Operations

### LDB – Load Byte (8 bits)

**Description:**

An eight-bit value is loaded from memory and placed in the target register. The memory address is the sum of the sign extended displacement and register Ra OR the sum of register Ra and Rb. The memory address must be double word aligned. Setting the record bit in the instruction will store +1, 0, or -1 to predicate register #1 depending on the sign of the value loaded.

**Instruction Format: LD, LDX**

**Clock Cycles:** 1+ (one memory access)

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

**Exceptions:**

If the target register is R0 then this instruction will not cause an exception. Otherwise an exception may be caused by a data-bus error signal input or a TLB miss.

**Operation:**

Rt = mem[Ra + displacement]

Rt = mem[Ra + Rb]

if (Record)

p1 = sign\_of(Rt)

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

### LDD – Load Double Word (64 bits)

**Description:**

A sixty-four-bit value is loaded from memory and placed in the target register. The memory address is the sum of the sign extended displacement and register Ra OR the sum of register Ra and Rb. Rb may be scaled by the size of the operand (eight). The memory address must be double word aligned. Setting the record bit in the instruction will store +1, 0, or -1 to predicate register #1 depending on the sign of the value loaded.

**Instruction Format: LD, LDX**

**Clock Cycles:** 1+ (one memory access)

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

**Exceptions:**

If the target register is R0 then this instruction will not cause an exception. Otherwise an exception may be caused by a data-bus error signal input or a TLB miss.

**Operation:**

Rt = mem[Ra + displacement]

Rt = mem[Ra + Rb \* Scale] (scale = 1 or 8)

if (Record)

p1 = sign\_of(Rt)

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

### STD – Store Double Word (64-bits)

**Description:**

A sixty-four-bit value is stored to memory from the source register Rs. The memory address is the sum of the sign extended displacement and register Ra OR the sum of register Ra and register Rb. Register Rb may be scaled by the size of the operand (eight). The memory address must be double word aligned.

**Instruction Format: ST, STX**

**Clock Cycles:** 1+ (one memory access)

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

**Operation:**

memory[Ra+displacement] = Rs

memory[Ra + Rb \* Scale] = Rs (scale = 1 or 8)

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

## MOVI – Move Immediate

Description:

The immediate move instruction is unusual in that it occupies two slots of the bundle as shown below. The movi instruction may be placed only in the first two instruction slots.

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 127 123 | 122 82 | 81 41 | 40 0 | |
| Control | Slot2 | Immediate63..23 | Immed22..3 | Other |

Clock cycles: 0.66 (two slots are occupied).