# Preface

## Who This Book is For

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

## Motivation

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

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

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

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

## About the Author

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

# Nomenclature

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

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

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

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

## Little Endian vs big Endian

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

## Endian

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

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

Little Endian:

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

Big Endian:

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

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

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

# Programming Model

## Register File

### Rn – General Purpose Registers

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

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

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

#### Register ABI

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

### Fn – Floating-Point Registers

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

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

### Vn – SIMD Registers

The SIMD register file contains 32 512-bit registers.

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

### Pn - Predicate Registers

There are 16 128-bit predicate registers.

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

## Vector Length (VL register)

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

Vector length has register tag #87.

|  |  |
| --- | --- |
| 15 8 | 7 0 |
| 0 | Elements7..0 |

### Code Address Registers

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

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

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

#### LRn – Link Registers

There are four registers in the Thor2023 architecture reserved for subroutine linkage. These registers are used to store the address after the calling instruction. They may be used to implement fast returns for several levels of subroutines or to used to call milli-code routines. The jump to subroutine, [JSR](#_JSR_–_Jump), and branch to subroutine, [BSR](#_BSR_–_Branch), instructions update a link register. The return from subroutine,. [RTS](#_RTS_–_Return), instruction is used to return to the next instruction.

#### PC – Program Counter

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

### LC - Loop Counter (reg 55)

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

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

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

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

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

T indicates that trace mode is active.

OM processor operating mode.

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

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

IPL is the interrupt mask level

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

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

#### Decimal Mode

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

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

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

### Register-Register Format

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

# Operating Modes

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

A subset of instructions is limited to machine mode.

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

## Arithmetic Operations

### Representations

#### long

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

#### int

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

#### short

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

#### char

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

#### byte

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

#### decimal

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

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

### Arithmetic Operations

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

**ADD Rt,Ra,Imm15**

**Instruction Format:** RI

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

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

**ADD Rt,Ra,Imm24**

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

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

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

**MULSU Rt, Ra, Rb**

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

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

### ADD - Register-Register

**Description:**

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

**Instruction Format:** R2

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

**Operation: R2**

Rt = Ra + Rb

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### ADDI - Add Immediate

**Description:**

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

**Instruction Format:** RIS

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

**Instruction Format:** RI

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

**Instruction Format:** RIP

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 30 | 29 26 | 25 17 | 16 12 | 11 7 | 6 0 |
| Fmt2 | Pr4 | Immediate8..0 | Ra5 | Rt5 | 207 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Ra + immediate

**Exceptions:**

**Notes:**

### AND – Bitwise And

**Description:**

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

**Instruction Format:** R2

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

**Operation: R2**

Rt = Ra + Rb

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### PFX – Constant Postfix

**Description:**

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

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

**Instruction Format: PFX0**

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

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

**Instruction Format: LPFX0**

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

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

**Instruction Format: PFX1**

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

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

**Instruction Format: LPFX1**

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

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

**Instruction Format: PFX2**

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

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

**Instruction Format: LPFX2**

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

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

**Instruction Format: PFX3**

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

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

**Instruction Format: LPFX3**

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

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

## Shift and Rotate Operations

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

Thor2024 has a full complement of shift instructions including rotates.

### ASL –Arithmetic Shift Left

**Description**:

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

**Instruction Format:** R2

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

**Operation:**

Rt = Ra << Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### ASLI –Arithmetic Shift Left

**Description**:

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

**Instruction Format:** RI7

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

**Operation:**

Rt = Ra << Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### LSR –Logic Shift Right

**Description**:

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

**Instruction Format:** R2

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

**Operation:**

Rt = Ra >> Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

### LSRI –Logical Shift Right

**Description**:

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

**Instruction Format:** RI7

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

**Operation:**

Rt = Ra >> Rb

**Operation Size:** .o

**Execution Units**: integer ALU

**Exceptions**: none

**Example**:

## Branch / Flow Control Instructions

### Overview

#### Mnemonics

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

#### Conditions

Conditional branches branch to the target address only if the condition is true. The condition is determined by the comparison of two general-purpose registers or by the comparison of a general purpose register and a postfixed immediate constant.

*The original Thor machine used instruction predicates to implement conditional branching. Another instruction was required to set the predicate before branching. Combining compare and branch in a single instruction may reduce the dynamic instruction count. An issue with comparing and branching in a single instruction is that it may lead to a wider instruction format.*

The comparison used is determined by a three-bit field in the instruction. There are five comparison types that may be performed as outlined in the table below.

|  |  |
| --- | --- |
| Cm3 | Comparison Type |
| 0 | signed integer comparisons |
| 1 | quad float comparison |
| 2 | quad decimal float comparison |
| 3 | posit comparison |
| 4 | unsigned integer comparisons |
| 5 to 7 | reserved |

### Conditional Branch Format

Branches use 32 or 48-bit opcodes.

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

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

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

### Branch Conditions

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

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

|  |  |  |  |
| --- | --- | --- | --- |
| 2x | Integer Comparison Test | Float / Decimal Float | Posit |
| 28h | signed less than | less than | less than |
| 29h | signed greater or equal | greater than or equal | greater than or equal |
| 2Ah | signed less than or equal | less than or equal | less than or equal |
| 2Bh | signed greater than | greater than | greater than |
| 2Ch |  | magnitude less than |  |
| 2Dh |  |  |  |
| 2Eh |  |  |  |
| 2Fh |  |  |  |
| 26h | equal | equal | equal |
| 27h | not equal | not equal | not equal |
| 24h |  | ordered |  |
| 25h | bit set or clear | unordered |  |
| 22h | bit set or clear immediate | bit set or clear immediate | bit set or clear immediate |

### Linkage

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

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

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

### Branch Target

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

*The target displacement field is recommended to be at least 16-bits. It is possible to get by with a displacement as small as 12-bits before a significant percentage of branches must be implemented as two or more instructions.*

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

### Branch to Register

The branch to register instruction allows a conditional return from subroutine to be used or a branch to a value in a register. Branching to a value in a register allows all bits of the instruction pointer to be set. Since addresses are formed as the sum of a code address register and a constant in the instruction, branching to a register is inherent in the instruction. The target constant may be set to zero. Specifying Ca = 0 will use the value zero rather than the contents of Ca zero. This allows absolute address branches to be formed.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ▼ | ▼ |  |  |  |  |  |  |
| 47 27 | 26 24 | 23 19 | 18 14 | 13 11 | 10 9 | 8 | 7 0 |
| Target21 | Ca3 | Rb5 | Ra5 | Cm3 | Lk2 | 0 | 2xh8 |

### BBC – Branch if Bit Clear

**Description**:

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

**Formats Supported**: B

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

**Operation:**

Lk = next IP

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

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### BBCI – Branch if Bit Clear Immediate

**Description**:

This instruction branches to the target address if a bit specified in an immediate field of the instruction of Ra is set, otherwise program execution continues with the next instruction. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: B

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

**Operation:**

Lk = next IP

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

IP = Ca + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### BRA – Branch Always

**Description**:

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

**Formats Supported**: BSR

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

**Formats Supported**: LBSR

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

**Operation:**

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### BSR – Branch to Subroutine

**Description**:

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

**Formats Supported**: BSR

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

**Formats Supported**: LBSR

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

**Operation:**

Lk = next IP

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### NOP – No Operation

NOP

**Description:**

This instruction does not perform any operation.

**Instruction Format:**

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

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

### RTD – Return from Subroutine and Deallocate

**Description**:

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

**Formats Supported**: RTS

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

**Operation:**

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

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

### RTE – Return from Exception

**Description**:

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

**Formats Supported**: RTS

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

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

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

**Operation:**

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

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

### RTS – Return from Subroutine

**Description**:

This instruction returns from a subroutine by transferring program execution to the address calculated as the sum of a link register and a constant. The const field is shifted left once before use.

**Formats Supported**: RTS

|  |  |  |  |
| --- | --- | --- | --- |
| 15 11 | 10 9 | 8 7 | 6 0 |
| Const5 | 02 | Lk2 | 357 |

**Operation:**

**Execution Units**: Branch

**Exceptions**: none

**Notes**:

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

# Opcode Maps

## Thor2021 Root Opcode

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | BRK | {R1} | {R2} | {R3} | ADDI | SUBFI | MULI | {SYS} | ANDI | ORI | EORI |  |  |  | MULUI | {CSR} |
| 1x | BEQZ | REP | BNEZ |  | JGATE | MULFI | SEQI | SNEI | SLTI | ADD | AND | SGTI | SLTUI | OR | EOR | SGTUI |
| 2x | BRA | DBRA |  |  | BBC | BBS | BEQ | BNE | BLT | BGE | BLE | BGT | BLTU | BGEU | BLEU | BGTU |
| 3x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4x | DIVI | CPUID | BRA | BRA | BLEND | CHKI | EXI7 | EXI23 |  | EXI55 | EXIM | CMPI | BMAP | CHK | SLT | DIVUI |
| 5x | CMPI | MLO | {VM} | VMFILL | CMOVNZ | BYTNDX | WYDENDX | UTF21NDX | SLL |  |  |  |  |  | MFLK | MTLK |
| 6x | {SIMD} | {FLT1} | {FLT2} | {FLT3} | MUX | {DFLT1} | {DFLT2} | {DFLT3} |  | {PST1} | {PST2} | {PST3} | EXI41 | | | |
| 7x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 8x | LDB | LDBU | LDW | LDWU | LDT | LDTU | LDO | LDOS | LLAL | LLAH | LEA | LDVOAR | LDOO | LDCTX | LDOU | LDH |
| 9x | STB | STW | STT | STO | STOC | STOS | STOO / STH | CAS | STSET | STMOV | STCMP | STFND |  | STCTX |  | CACHE |
| Ax |  |  |  |  |  | SYS | INT | MOV |  |  | {BTFLD} | MOVS | PUSH | PUSH 2R | PUSH 3R | ENTER |
| Bx | LDBX | LDBUX | LDWX | LDWUX | LDTX | LDTUX | LDOX | LDOOX | LLALX | LLAHX | LEAX | LDORX | POP | POP 2R | POP 3R | LEAVE |
| Cx | STBX | STWX | STTX | STOX | STOCX | STHX | STOOX |  |  |  |  | LINK | UN  LINK | LDHX | LDOUX | CACHEX |
| Dx | CMPIL |  | MULIL | SLTIL | ADDIL | SUBFIL | SEQIL | SNEIL | ANDIL | ORIL | EORIL | SGTIL | SLTUIL | DIVIL | MULUIL | SGTUIL |
| Ex |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Fx | DEFCAT | NOP | RTS | CARRY |  | {BCD} | STP | SYNC | MEMSB | MEMDB | WFI | SEI | MBNEZ |  |  |  |

## Thor2024 Root Opcode

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **0x** | 0  BRK / SYS | 1  {R1} | 2  {R2} | 3  {R3} | 4  ADDI | 5  SUBFI | 6  MULI | 7  {SYS} |
|  | 8  ANDI | 9  ORI | 10  EORI | 11  CMPI | 12  ADDIS | 13  DIVI | 14  MULUI | 15  CSR |
| **1x** | 16 | 17 | 18 | 19 | 20  ADDIP | 21  DIVUI | 22  MULSUI | 23  MODI |
|  | 24 | 25 | 26 | 27 | 28 | 29  DIVSUI | 30  MODSUI | 31  MODUI |
| **2x** | 32  BSR / BRA | 33  DBRA | 34  BBCI / BBSI | 35  RTx | 36  JSR | 37 | 38  BEQ | 39  BNE |
|  | 40  BLT | 41  BGE | 42  BLE | 43  BGT | 44  BLTU | 45  BGEU | 46  BLEU | 47  BGTU |
| **3x** | 48  {VM} | 49  VMFILL | 50  CHKI | 51  CMOVZ | 52  BMAP | 53  BLEND | 54 | 55  DIVUI |
|  | 56  INDEXOF | 57  BCMP | 58  BFND | 59  BMOV | 60  {BITFLD} | 61 | 62 | 63 |
| **4x** | 64  LDB | 65  LDBU | 66  LDW | 67  LDWU | 68  LDT | 69  LDTU | 70  LDO | 71  LDOU |
|  | 72  LDH | 73  LDG | 74  PUSH | 75  POP | 76  ENTER | 77  LEAVE | 78  CACHE | 79  {LDX} |
| **5x** | 80  STB | 81  STW | 82  STT | 83  STO | 84  STH | 85  STPTR | 86  STG | 87  {STX} |
|  | 88  ASL ASLI | 89  LSR LSRI | 90  ASR ASRI | 91  ROL ROLI | 92  ROR RORI | 93 | 94  MFLK | 95  MTLK |
| **6x** | 96 | 97  {FLT1} | 98  {FLT2} | 99  {FLT3} | 100  MUX | 101  {DFLT1} | 102  {DFLT2} | 103  {DFLT3} |
|  | 104 | 105  {PST1} | 106  {PST2} | 107  {PST3} | 108 | 109  SRLI | 110  SRAI | 111 |
| **7x** | 112  IRQ | 113  STOP | 114  SYNC | 1115  WFI / PFI | 116  FENCE | 117 | 118 | 119 |
|  | 120  REP | 121 | 122  CPUID | 123  PFX0 | 124  PFX1 | 125  PFX3 | 126  PFX3 | 127  NOP |