# Introduction

## Features

* 64-bit integer data path
* 64-bit double precision floating-point data path
* 32-entry integer register file
* 32-entry floating-point register file
* 32-bit fixed size instructions

## Future Features

* 4-way out-of-order (ooo) superscalar execution
* precise exception handling
* branch prediction with branch target buffer (BTB)
* Instruction L1, L2 and data L1, L2 caches
* 7 entry write buffer
* Dual memory channels

## History

FT65002 is a work in progress beginning in September 2020. TF65002 originated from NVIO3 which originated from NVIO which originated from FT64 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 FT64.v. The author has tried to be innovative with this design borrowing ideas from many other processing cores.

## Motivation

The author wanted an FPGA based processing core for experimental purposes. This is in part an example of sub-optimal design. For instance, there is an extra unused bit in the JSR and JMP instructions. Using this extra bit for the target address for instance, would cause the target address field to be shift by a bit relative to byte positions. As it is one can read most of the target address directly from the machine code. So, the sub-optimization is to add a human factor to the design. There are other examples to be found.

### Case Comparison 6502

6502 vs FT65002

#### 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 FT65002 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. FT65002 uses a fixed 32-bit instruction encoding. As such for a given single instruction it requires twice the memory of a 6502. However, the instructions in the FT65002 operate on 64-bit values, to perform the same operations in the 6502 would require many more bytes. Several instructions in the FT65002 are more powerful than what can be found in the 6502.

#### Registers

The FT65002 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 FT65002 has available many more transistors than were available to the 6502 design.

#### Instructions

The 6502 uses relative branches to allow a code dense instruction encoding. Since there are enough bits available in the FT65002 branch instructions to encode an absolute address, absolute addressing is used. It takes a little bit less hardware to use absolute addressing rather than relative addressing. It is also much easier to “see” the target address in the machine code if it is properly aligned.

### Case Comparison RISCV

RISCV vs FT65002

#### 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.

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. This design only requires a single bit 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. In this design using only a single bit to specify the constant register allows the constant to be four bits larger. In fact, this design allows a 25-bit constant field which is important as it allows 64-bit constants to be built using only four instructions. RISCV does not really provide much for building constants over 32 bits.

#### Register File

RISCV does almost everything using general-purpose registers. This paradigm increases the pressure on the register file. In the FT65002 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

For this design, the result of a compare operation is stored in a compare result register. A couple of questions come to mind as to the use of compare results registers. Why use them instead of general-purpose registers? And, how many compare results registers is enough? 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. How many compare results registers are needed? Four was deemed sufficient to provide two additional registers in addition to supporting the use of separate registers for integer and floating-point compare results. With register renaming available in a superscalar processor, there does not need to be whole bunches of compare results 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 six operating modes. It has the RISCV operating modes plus separate modes for interrupt and debug. The author has seen a comment to the effect that debug on a RISCV processor really acts like an additional mode. This has been made explicit in this design.

**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 program counter or IP register. The program counter is a synonym for program counter 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

## **Registers**

### Overview

The FT65002 ISA is a 32-register machine with a separate register file for integer or floating-point. There is an exception linkage register associated with each operating mode. There are many control and status (CSR) registers which hold an assortment of specific values relevant to processing.

### General Purpose Registers (x0 to x31)

The register usage convention probably has more to do with software than hardware. Excepting a few special cases, the registers are general purpose in nature. Registers may hold either integer or floating-point values.

x0 always has the value zero. Registers x30 and x31 are used for stack references and subject to stack bounds checking.

x1 may be used with the constant building instructions (LUI, LMI, AMIPC)

|  |  |  |
| --- | --- | --- |
| Register | Description / Suggested Usage | Saver |
| x0 | always reads as zero (hardware) |  |
| x1 | constant building / temporary (cb) |  |
| x2-x8 | temporaries (t0-t6) | caller |
| x9-x19 | register variables (s0-s10) | callee |
| x20-x27 | function arguments (a0-a7) | caller |
| x28 | thread pointer (tp) |  |
| x29 | global data pointer (gp) | callee |
| x30 | base / frame pointer (fp) | callee |
| x31 | current stack pointer (sp) | callee |
|  |  |  |
| cr0-cr3 | compare results |  |
| ra0 | return address register |  |
| ra1 | alternate return address register |  |

### Compare Results Registers

The result of a compare operation is stored in a compare result register. There are four eight-bit compare results registers in the design. The compare results registers store the flag results of a compare operation. Typically, one compare result is used for each of integer and floating-point compares. Compare results registers are updated by one of the compare instructions.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| N | V | O | P | ~ | ~ | Z | C |

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Meaning |
| 0 | C | Carry flag, set if operation overflows |
| 1 | Z | Zero flag, set if result is zero |
| 2 | ~ | reserved |
| 3 | ~ | reserved |
| 4 | P | Parity (exclusive or of all result bits) |
| 5 | O | Odd, set if result is odd |
| 6 | V | Overflow, set if signed result overflows |
| 7 | N | Negative, set if signed result is less than zero |

*The author has chosen in this design to make the compare results registers look like a processor condition code register. There is a large software base that uses condition code registers and programmers are familiar with them. The author feels keeping a familiar look and feel is important.*

*Choosing to support only four compare results registers is because many architectures including superscalar ones can get by with just a single flags register. There is no need for many registers in this case. Having more than two register may be handy. Typically, one register is used by the compiler for integer operations and second register used for floating-point operations. The compare results registers are effectively more powerful in this architecture because they allow result flags to be accumulated into a single register for multiple compare operations.*

### Return Address Registers

There are two return address registers (ra0 and ra1) available in the design. These are designated the normal and alternate return address registers. Return address registers store the address of a JSR operation. A return instruction is used to return to some point after the JSR instruction.

*There are two return address registers as there is no need for more. Often in modern designs a general-purpose register is delegated to store the return address. The ABI typically specifies which register to use. By restricting the number of registers that can be selected as the return address register, it is possible to create an instruction encoding that is a little more bit efficient. The encoding of a JSR instruction in this design allows a 24-bit absolute address specification.*

*Storing the address of the JSR instruction rather than the next program address in the return address register is a little unusual. It is motivated by the observation that the return instruction can return to a point substantially past the place the instruction after the JSR would be located. For inline subroutine parameters and possibly exception handling trampolines the return instruction can return past the normal return address location. Since this capability is built into the return instruction, some hardware may be conserved by simply copying the JSR address to the return address register rather than computing the next address and using that.*

### Program Counter

The program counter identifies which instruction to execute. The program counter increments as instructions are processed. The increment may be overridden using one of the flow control instructions. The program counter addresses 32-bit instruction parcels. The program counter increments by four. The program counter register is also split into two sections. Only the lower 24 bits of the IP increment.

*There is little reason to increment higher order program counter bits. That would just waste hardware. Most code fragments are small and the JSR instruction is used to set the program counter from routine to routine, overriding the increment. The only time there is an issue is if code passes through the 16MB boundary. This can be handled by carefully aligning code so that a subroutine does not span the boundary.*

|  |  |
| --- | --- |
| 63 24 | 23 0 |
| PC High40 | PC Low[23..0] |

### Register Zero

Register zero – r0 – always reads as zero.

*Although forcing register zero to zero all the time uses up a register it is generally considered valuable enough to do. It removes the need to initialize a register to zero for use.*

### 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.

### Base Registers

There are sixteen base address registers in the design. These registers hold the base address of a memory segment and some basic access rights.

|  |  |
| --- | --- |
| Register | Description / Usage |
| sg0 to sg7 | data segment registers |
| sg8, sg9 | reserved |
| sg10 | stack segment register |
| sg11 | I/O segment register |
| sg12 to sg15 | code segment registers |
|  |  |

*The author chose a set of 16 base registers, in part so that the register selection is easily viewable from the address; another human factor. Several architecture have fewer base registers.*

## Control and Status Registers

### Control Register Zero (CSR #000)

This register contains miscellaneous control bits including a bit to enable protected mode.

|  |  |  |
| --- | --- | --- |
| Bit |  | Description |
| 0 | Pe | Protected Mode Enable: 1 = enabled, 0 = disabled |
| 8 to 13 |  |  |
| 16 |  |  |
| 30 | DCE | data cache enable: 1=enabled, 0 = disabled |
| 32 | BPE | branch predictor enable: 1=enabled, 0=disabled |
| 34 | WBM | write buffer merging enable: 1 = enabled, 0 = disabled |
| 35 | SPLE | speculative load enable (1 = enable, 0 = disable) (0 default) |
| 36 |  |  |
| 63 | D | debug mode status. this bit is set during an interrupt routine if the processor was in debug mode when the interrupt occurred. |

This register supports bit set / clear CSR instructions.

DCE

Disabling the data cache is useful for some codes with large data sets to prevent cache loading of values that are used infrequently. Disabling the data cache may reduce security risks for some kinds of attacks. The instruction cache may not be disabled. Enabling / disabling the data cache is also available via the cache instruction.

BPE

Disabling branch prediction will significantly affect the cores performance but may be useful for debugging. Disabling branch prediction causes all branches to be predicted as not-taken. No entries will be updated in the branch history table if the branch predictor is disabled.

WBM bit

Merging of values stored to memory may be disabled by setting this bit. On reset write buffer merging is disabled because it is likely desirable to setup I/O devices. Many I/O devices require updates to individual bytes by separate store instructions. (Write buffer merging is not currently implemented).

SPLE

Enabling speculative loads give the processor better performance at an increased security risk to meltdown attacks.

### HARTID (0x001)

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. No core should have the value zero as the hartid.

### TICK (0x002)

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

### CAUSE (0x006)

This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code to determine what to do. Only the low order 8 bits are implemented. The high order bits read as zero and are not updateable.

### BADADDR (CSR 0x007)

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 XL register.

### BAD\_INSTR (CSR 0x00B)

This register contains a copy of the exceptioned instruction.

### SEMA (CSR 0x00C) Semaphores

This register is available for system semaphore or flag use. The least significant bit is tied to the reservation address status input (rb\_i). It will be set if a STOC instruction was successful. The least significant bit is also cleared automatically when an interrupt (BRK) or interrupt return (RTI) instruction is executed. Any one of the remaining bits may also be cleared by an RTI instruction. This could be a busy status bit for the interrupt routine. Bits in this CSR may be set or cleared with one of the CSRxx instructions. This register has individual bit set / clear capability.

|  |  |
| --- | --- |
| Semaphore | Usage Convention |
| 0 | LDDR / STDC status bit |
| 1 | system garbage collection protector |
| 2 | system |
| 3 | input / output focus list |
| 4 | keyboard |
| 5 | system busy |
| 6 | memory management |
| 7-127 | currently unassigned |

### FSTAT (CSR 0x014) Floating Point Status and Control Register

The floating-point status and control register may be read using the CSR instruction. Unlike other CSR’s the control register has its own dedicated instructions for update. See the section on floating point instructions for more information.

|  |  |  |  |
| --- | --- | --- | --- |
| Bit |  | Symbol | Description |
| 51:47 |  |  | reserved |
| 46:44 | **RM** | rm | rounding mode |
| 43 | **E5** | inexe | - inexact exception enable |
| 42 | **E4** | dbzxe | - divide by zero exception enable |
| 41 | **E3** | underxe | - underflow exception enable |
| 40 | **E2** | overxe | - overflow exception enable |
| 39 | **E1** | invopxe | - invalid operation exception enable |
| 38 | **NS** | ns | - non standard floating point indicator |
| **Result Status** | | | |
| 32 |  | fractie | - the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception) |
| 31 | **RA** | rawayz | rounded away from zero (fraction incremented) |
| 30 | **SC** | C | denormalized, negative zero, or quiet NaN |
| 29 | **SL** | neg < | the result is negative (and not zero) |
| 28 | **SG** | pos > | the result is positive (and not zero) |
| 27 | **SE** | zero = | the result is zero (negative or positive) |
| 26 | **SI** | inf ? | the result is infinite or quiet NaN |
| **Exception Occurrence** | | | |
| 21 to 25 |  |  | reserved |
| 20 | **X6** | swt | {reserved} - set this bit using software to trigger an invalid operation |
| 19 | **X5** | inerx | - inexact result exception occurred (sticky) |
| 18 | **X4** | dbzx | - divide by zero exception occurred |
| 17 | **X3** | underx | - underflow exception occurred |
| 16 | **X2** | overx | - overflow exception occurred |
| 15 | **X1** | giopx | - global invalid operation exception – set if any invalid operation exception has occurred |
| 14 | **GX** | gx | - global exception indicator – set if any enabled exception has happened |
| 13 | **SX** | sumx | - summary exception – set if any exception could occur if it was enabled  - can only be cleared by software |
| **Exception Type Resolution** | | | |
| 8 to 12 |  |  | reserved |
| 7 | **X1T** | cvt | - attempt to convert NaN or too large to integer |
| 6 | **X1T** | sqrtx | - square root of non-zero negative |
| 5 | **X1T** | NaNCmp | - comparison of NaN not using unordered comparison instructions |
| 4 | **X1T** | infzero | - multiply infinity by zero |
| 3 | **X1T** | zerozero | - division of zero by zero |
| 2 | **X1T** | infdiv | - division of infinities |
| 1 | **X1T** | subinfx | - subtraction of infinities |
| 0 | **X1T** | snanx | - signaling NaN |

### KEYS – (CSR 0x20 to 0x23)

These registers contain a collection of keys associated with the process for the memory system. Each key is twenty bits in size. Each register contains three keys for a total of nine keys. All three registers are searched in parallel for keys matching the one associated with the memory page.

|  |  |  |  |
| --- | --- | --- | --- |
| 63 60 | 59 40 | 39 20 | 19 0 |
| ~6 | key3 | key2 | key1 |

### INFO (0xF0 to 0xFF)

This set of registers contains general information about the core including the manufacturer name, cpu class and name, and model number.

## Operating Levels

The core has six operating modes. The highest operating mode is operating mode five which is called the debug operating mode. Operating mode five has complete access to the machine including special registers reserved for debug. Other operating levels may have more restricted access. When an interrupt occurs, the operating mode is set to the interrupt mode. The core vectors to an address depending on the current operating mode. When not operating at user mode addresses are not subjected to translation and the virtual address and physical address are the same.

|  |  |
| --- | --- |
| Operating Mode | Moniker |
| 0 | user |
| 1 | supervisor |
| 2 | hypervisor |
| 3 | machine |
| 4 | interrupt |
| 5 | debug |

### Switching Operating Modes

The operating mode is automatically switched to the interrupt mode when an interrupt occurs. The BRK instruction may be used to switch operating modes. The REX instruction may also be used by an interrupt handler to switch the operating mode to a lower mode. The IRET instruction will switch the operating level back to what it was prior to the interrupt.

## Privilege Levels

The core supports a 256-level privilege level system. Privilege level zero is assigned to operating level zero. Privilege level one is assigned to operating level one. Privilege levels 2 to 6 are assigned to operating level two. The remaining privilege levels are assigned to operating level three.

## Memory Management Unit - MMU

### Introduction

Many systems can benefit from the provision of virtual memory management. Virtual memory may be used to protect the address space of one app from another. Virtual memory can enhance the reliability and security of a system.

The simplified system MMU provides minimalistic base and bound and paging capabilities for a small to mid size system. This MMU is not suitable for larger systems as the paging tables would be too large. Base bound and paging are applied only to user mode apps. In other operating modes the system sees a flat address space with no restrictions on access. Base address generation is applied to virtual addresses first to generate a linear address which is then mapped using a paged mapping system. Access rights are governed by the base register since all pages in the based on the same address are likely to require the same access. Support for access rights is optional if it is desired to reduce the hardware cost. To simplify hardware there are no bound registers. Bounds are determined by what memory is mapped into the base address area.

### Base Registers

The upper address bits of a virtual or effective address are not used for addressing memory and are available to select base register. The MMU includes 16 base registers. The base register in use is selected by the upper nybble of the virtual address. In the case of the program address, program counter bits 62 and 63 are used to select one of four registers. Additionally, if the program address has all ones in bits 24 to 63 then base addressing is bypassed. This provides a shared program area containing the BIOS and OS code.

|  |  |  |
| --- | --- | --- |
| Base Regno | Usage | Selected By |
| 0 to 7 | data | bits 60 to 63 of effective address |
| 8, 9 | reserved | bits 60 to 63 of effective address |
| 10 | Stack | bits 60 to 63 of effective address |
| 11 | I/O | bits 60 to 63 of effective address |
| 12 to 15 | code | bits 62, 63 of pc |

### Base Register Format

|  |  |
| --- | --- |
| 63 4 | 3 0 |
| Base Address60 | RWX |

The low order four bits of the base register are reserved for access rights bits. Supporting memory access rights is optional.

R: 1 = segment readable

W: 1 = segment writeable

X: 1 = segment executable

### Linear Address Generation

The base address value contained in the upper 60 bits of a base register is shifted left 16 bits before being added to the virtual address. This gives potentially a 76-bit address space.

Note there is no limit or bound register. Access is limited by what is mapped into the segment.

### The Page Map

The page directly maps virtual address pages to physical ones. The page map is a dedicated memory internal to the processing core accessible with the custom ‘mvmap’ instruction. It is similar in operation to a TLB but is much simpler. TLB’s cache address translations and create TLB miss exceptions. Page walks of mapping tables are required to update the TLB on a miss. There are no exceptions associated with the page mapping table.

In addition to based addresses, memory is divided up into 64kB pages which are mapped. There are 32 memory maps available. A memory map represents an address space; a five-bit address space identifier is in use. Address spaces will need to be shared if more than 32 apps are running in the system. The desire is to keep the mapping tables small so they may fit into a small number of standard memory blocks. For instance, for the sample system there are 8192 pages required to map the 512MB address space. Any individual app is limited to maximum of 256MB (one half of the memory available). The virtual page number is used to lookup the physical page in the page mapping table. Addresses with the top eight bits set are not mapped to allow access to the system ROM.

The page mapping table is indexed by the ASID and the virtual page number to determine the physical page. The ‘mvmap’ instruction uses Rs1 to contain a mapping table index. Bits 16 to 20 of Rs1 are the ASID, bits 0 to 15 of Rs1 are used for the virtual page number. It is expected that the virtual page number is a small number. Rs2 contains the new value of the physical page. The current value of the physical page is placed in Rd when the instruction executes.

|  |  |  |
| --- | --- | --- |
| ASID5 | Virtual Page | Physical Page |
| 0 | 0 | 10 |
| 1 | 11 |
| … |  |
| 4094 | 18 |
| 4095 | 19 |
| 1 | 0 |  |
| 1 |  |
| … |  |
| 4094 |  |
| 4095 |  |
| … 30 more address spaces | |  |

The low order 16 bits of an address pass through both linear address generation and paging unchanged.

### The 64kB Page

Many memory systems use a 4kB page size. A 64kB page size is used here mainly to restrict the number of page entries in the page map table. A smaller page size would result in too many pages of memory to support multiple tasks. Even given a 64kB page size there are still 8192 pages of memory available.

MVMAP

Rs1:

|  |  |  |
| --- | --- | --- |
| 31 20 | 20 16 | 15 0 |
| Unused - should be zero | ASID5 | Virtual page number 16 bits max |

Physical Memory Attributes

Physical memory attributes are stored in an eight-entry table. This table includes the address range the attributes apply to and the attributes themselves.

# Instruction Formats

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | |  | | | Opcode8 |  |
| Constant16 | | | | | | | | | | | | Cause8 | | | | | | | 00h | BRK |
| Funct6 | | | | | | | ~3 | | | Rs2 | Rs1 | | | Rd | | | | | 02h | {Reg2} |
| 36 | | | | | | | Fn3 | | | Rs2 | Rs1 | | | Rd | | | | | 02h | BMM |
| Funct4 | | | | Rs3 | | | | | | Rs2 | Rs1 | | | Rd | | | | | 03h | {Reg3} |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 04h | ADD |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 05h | SUBF |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 06h | MUL |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Mop3 | | | Cd2 | | 07h | CMP |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 08h | AND |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 09h | OR |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 0Ah | EOR |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Mop3 | | | Cd2 | | 0Bh | BIT |
| 0 | ~3 | | | | Funct4 | | | ~ | | Rs2 | Rs1 | | | Rd | | | | | 0Ch | {SHIFT} |
| 1 | ~3 | | | | Funct4 | | | Const5..0 | | | Rs1 | | | Rd | | | | | 0Ch | {SHIFT} |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 0Eh | MULU |
| Fn3 | | | Om3 | | | Regno8 | | | | | Rs1 | | | Rd | | | | | 0Fh | CSR |
| ~ | | | | | | | | | | | Rs1 | | | Rd | | | | | 17h | PERM |
| Target23..2 | | | | | | | | | | | | | | | ~ | | | Lk1 | 20h | JSR |
| Target23..2 | | | | | | | | | | | | | | | ~ | | | ~1 | 21h | JMP |
| Constant13..0 | | | | | | | | | | | Rs1 | | | ~4 | | | | Lk1 | 22h | JSR d[Rs1] |
| Constant13..0 | | | | | | | | | | | Rs1 | | | ~4 | | | | Lk1 | 23h | JMP d[Rs1] |
| Constant13..0 | | | | | | | | | | | 31 | | | RO4 | | | | Lk1 | 24h | RTS |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 28h | BEQ |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 29h | BNE |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 2Ah | BLT / BMI |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 2Bh | BGE / BPL |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 2Ch | BLE |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 2Dh | BGT |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 2Eh | BVS |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 2Fh | BVC |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 30h | BOD |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 32h | BLTU |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 33h | BGEU |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 34h | BLEU |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 35h | BGTU |
| Target23..2 | | | | | | | | | | | | | | | Cd2 | | | | 36h | BPS |
| Constant63..41 | | | | | | | | | | | | | | | Rd1 | | | | 40h-43h | LUI |
| Constant38..16 | | | | | | | | | | | | | | | Rd1 | | | | 44h-47h | LMI |
| Constant38..16 | | | | | | | | | | | | | | | Rd1 | | | | 48h-4Bh | AMIPC |
| Constant13..0 | | | | | | | | | | | Lvl3 | | Sema/RO6 | | | | | Lk1 | 24h | {RTS} |
| **Memory** | | | | | | | | | | | | | | | | | | | | |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 80h | LDB |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 81h | LDBU |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 82h | LDW |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 83h | LDWU |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 84h | LDT |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 85h | LDTU |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 86h | LDO |
| Constant13..0 | | | | | | | | | | | Rs1 | | | Rd | | | | | 87h | LDOR |
| ~3 | | | S | Rs3 | | | | | 05 | | Rs1 | | | Rd | | | | | 8Fh | LDB |
| ~3 | | | S | Rs3 | | | | | 15 | | Rs1 | | | Rd | | | | | 8Fh | LDBU |
| ~3 | | | S | Rs3 | | | | | 25 | | Rs1 | | | Rd | | | | | 8Fh | LDW |
| ~3 | | | S | Rs3 | | | | | 35 | | Rs1 | | | Rd | | | | | 8Fh | LDWU |
| ~3 | | | S | Rs3 | | | | | 45 | | Rs1 | | | Rd | | | | | 8Fh | LDT |
| ~3 | | | S | Rs3 | | | | | 55 | | Rs1 | | | Rd | | | | | 8Fh | LDTU |
| ~3 | | | S | Rs3 | | | | | 65 | | Rs1 | | | Rd | | | | | 8Fh | LDO |
| ~3 | | | S | Rs3 | | | | | 75 | | Rs1 | | | Rd | | | | | 8Fh | LDOR |
| Constant14..5 | | | | | | | | | Rs2 | | Rs1 | | | Const4..0 | | | | | A0h | STB |
| Constant14..5 | | | | | | | | | Rs2 | | Rs1 | | | Const4..0 | | | | | A1h | STW |
| Constant14..5 | | | | | | | | | Rs2 | | Rs1 | | | Const4..0 | | | | | A2h | STT |
| Constant14..5 | | | | | | | | | Rs2 | | Rs1 | | | Const4..0 | | | | | A3h | STO |
| Constant14..5 | | | | | | | | | Rs2 | | Rs1 | | | Const4..0 | | | | | A4h | STOC |
| Constant14..5 | | | | | | | | | Rs2 | | Rs1 | | | Const4..0 | | | | | A5h | STPTR |
| ~3 | | | S | Rs3 | | | | | Rs2 | | Rs1 | | | 05 | | | | | AFh | STB |
| ~3 | | | S | Rs3 | | | | | Rs2 | | Rs1 | | | 15 | | | | | AFh | STW |
| ~3 | | | S | Rs3 | | | | | Rs2 | | Rs1 | | | 25 | | | | | AFh | STT |
| ~3 | | | S | Rs3 | | | | | Rs2 | | Rs1 | | | 35 | | | | | AFh | STO |
| ~3 | | | S | Rs3 | | | | | Rs2 | | Rs1 | | | 45 | | | | | AFh | STOC |
| ~3 | | | S | Rs3 | | | | | Rs2 | | Rs1 | | | 55 | | | | | AFh | STPTR |
| **Posit Arithmetic** | | | | | | | | | | | | | | | | | | | | |
| ~3 | | | ~ | Funct5 | | | | | Prs2 | | Prs1 | | | Prd | | | | | E2h | {PST2} |
| ~3 | | | ~ | 15 | | | | | Funct5 | | Prs1 | | | Prd | | | | | E2h | {PST1} |
| ~ | | | ~ | Prs3 | | | | | Prs2 | | Prs1 | | | Prd | | | | | E4h | PMA |
| ~ | | | ~ | Prs3 | | | | | Prs2 | | Prs1 | | | Prd | | | | | E5h | PMS |
| ~ | | | ~ | Prs3 | | | | | Prs2 | | Prs1 | | | Prd | | | | | E6h | PNMA |
| ~ | | | ~ | Prs3 | | | | | Prs2 | | Prs1 | | | Prd | | | | | E7h | PNMS |
| ~ | | | ~ | Prs3 | | | | | Prs2 | | Prs1 | | | Prd | | | | | E8h | PMIN |
| ~ | | | ~ | Prs3 | | | | | Prs2 | | Prs1 | | | Prd | | | | | E9h | PMAX |
| Constant24 | | | | | | | | | | | | | | | | | | | EAh | NOP |
| **Floating Point** | | | | | | | | | | | | | | | | | | | | |
| Rm3 | | 0 | | Funct5 | | | | | Frs2 | | Frs1 | | | Frd | | | | | F2h | {FLT2} |
| Rm3 | | 0 | | 15 | | | | | Funct5 | | Frs1 | | | Frd | | | | | F2h | {FLT1} |
| Rm3 | | 0 | | Frs3 | | | | | Frs2 | | Frs1 | | | Frd | | | | | F4h | FMA |
| Rm3 | | 0 | | Frs3 | | | | | Frs2 | | Frs1 | | | Frd | | | | | F5h | FMS |
| Rm3 | | 0 | | Frs3 | | | | | Frs2 | | Frs1 | | | Frd | | | | | F6h | FNMA |
| Rm3 | | 0 | | Frs3 | | | | | Frs2 | | Frs1 | | | Frd | | | | | F7h | FNMS |
| ~3 | | 0 | | Frs3 | | | | | Frs2 | | Frs1 | | | Frd | | | | | F8h | FMIN |
| ~3 | | 0 | | Frs3 | | | | | Frs2 | | Frs1 | | | Frd | | | | | F9h | FMAX |

# Opcode Maps

## Root Level

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| **ALU** | | | | | | | | |
| 00000 | BRK |  | {R2} | {R3} | ADD | SUBF | MUL | CMP |
| 00001 | AND | OR | EOR | BIT | {SHIFT} |  | MULU | CSR |
| 00010 | DIV | DIVU |  |  |  |  |  | PERM |
| 00011 |  |  |  |  |  |  |  |  |
| **Branch Unit** | | | | | | | | |
| 00100 | JSR abs | JMP abs | JSR d[xn] | JMP d[xn] | RTS | RTI | SYS |  |
| 00101 | BEQ | BNE | BLT | BGE | BLE | BGT | BVS | BVC |
| 00110 | BOD |  | BLTU | BGEU | BLEU | BGTU | BPS |  |
| 00111 |  |  |  |  |  |  |  |  |
|  | | | | | | | | |
| 01000 | LUI | | | | LMI | | | |
| 01001 | AUIPC | | | |  |  |  |  |
| 01010 |  |  |  |  |  |  |  |  |
| 01011 |  |  |  |  |  |  |  |  |
| 01100 |  |  |  |  |  |  |  |  |
| 01101 |  |  |  |  |  |  |  |  |
| 01110 |  |  |  |  |  |  |  |  |
| 01111 |  |  |  |  |  |  |  |  |
| **Memory Unit** | | | | | | | | |
| 10000 | LDB | LDBU | LDW | LDWU | LDT | LDTU | LDO | LDOR |
| 10001 |  |  |  |  |  |  |  | {LNDX} |
| 10010 |  |  |  |  |  |  |  |  |
| 10011 |  |  |  |  |  |  |  |  |
| 10100 | STB | STW | STT | STO | STOC | SPTR |  |  |
| 10101 |  |  |  |  |  |  |  | {SNDX} |
| 10110 |  |  |  |  |  |  |  |  |
| 10111 |  |  |  |  |  |  |  |  |
| 11000 |  |  |  |  |  |  |  |  |
| 11001 |  |  |  |  |  |  |  |  |
| 11010 |  |  |  |  |  |  |  |  |
| 11011 |  |  |  |  |  |  |  |  |
| **Floating Point / Posit Arithmetic Unit** | | | | | | | | |
| 11100 |  |  | {PST2} |  | PMA | PMS | PNMA | PNMS |
| 11101 | PMIN | PMAX | NOP |  |  |  |  |  |
| 11110 |  |  | {FLT2} |  | FMA | FMS | FNMA | FNMS |
| 11111 | FMIN | FMAX |  |  |  |  |  |  |

## {R3} Triple Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0 | MIN | MAX | MAJ | MUX | ADD | SUB |  |  |
| 1 | AND | OR | EOR |  |  |  |  | RGF |

## {R2} Double Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 000 | NAND | NOR | ENOR | BMM |  |  | MUL | CMP |
| 001 | AND | OR | EOR | BIT |  |  | MULU |  |
| 010 | DIV | DIVU |  |  |  |  |  | PERM |
| 011 | PTRDIF |  |  |  |  |  |  | RGF |
| 100 |  |  |  |  |  |  |  |  |
| 101 |  |  |  |  |  |  |  |  |
| 110 |  |  |  |  |  |  |  |  |
| 111 |  |  |  |  |  |  |  |  |

## Monadic Ops – {FLT1} Funct5

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 00 | FMOV | FRSQRTE | FTOI | ITOF |  |  | FSIGN | FMAN |
| 01 |  | FS2D | FS2Q | FD2Q | FSTAT | FSQRT | ISNAN | FINITE |
| 10 | FTX | FCX | FEX | FDX | FRM | TRUNC | FSYNC | FRES |
| 11 | FSIG | FD2S | FQ2S | FQ2D |  |  | FCLASS | UNORD |

## Dyadic Ops – {FLT2} Funct5

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 00 | SCALEB | {FLT1} |  |  | FADD | FSUB |  |  |
| 01 | FMUL | FDIV | FREM | FNXT | FAND | FOR |  |  |
| 10 | FCMP |  |  |  |  |  |  |  |
| 11 | CPYSGN | SGNINV | SGNAND | SGNOR | SGNXOR | SGNXNOR | FCLASS |  |

## Dyadic Ops – {PST2} Funct5

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 00 |  | {PST1} |  |  | PADD | PSUB |  |  |
| 01 | PMUL | PDIV | PREM |  |  |  |  |  |
| 10 | PCMP |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  | PCLASS |  |

# ALU Operations

## ADD – Addition

**Description**:

Add two operand values and place the result in the target register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

**Formats Supported**: RR, RI

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## AUIPC – Add Upper Immediate to PC

**Description**:

Add an immediate value to the program counter register and place the result into either x1 or x2. The immediate constant is composed of 14 bits of zeros on the right-hand side, 25 constant bits for bits 14 to 38, and bit 38 of the constant is sign extended to 64 bits. This instruction may be used to form program counter relative addresses.

**Formats Supported**: LUI

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## AND – Bitwise ‘And’

**Description**:

Bitwise ‘And’ two operand values and place the result in the target register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

**Formats Supported**: RR, RI

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

## ASL – Arithmetic Shift Left

**Description**:

Left shift one operand value by a second 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 Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

**Formats Supported**: RR, RI6

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | ~3 | 04 | ~ | Rs2 | Rs1 | Rd | 0Ch | ASL |
| 1 | ~3 | 04 | Const5..0 | | Rs1 | Rd | 0Ch | ASL |

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

## ASR – Arithmetic Shift Right

**Description**:

Right shift one operand value by a second operand value while preserving the sign bit and place the result in the target register. The sign bit is preserved as the shift takes place. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

**Formats Supported**: RR, RI6

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | ~3 | 44 | ~ | Rs2 | Rs1 | Rd | 0Ch | ASR |
| 1 | ~3 | 44 | Const5..0 | | Rs1 | Rd | 0Ch | ASR |

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

## BIT – Bitwise ‘And’

**Description**:

Bitwise ‘And’ two operand values and place the resulting status in a compare results register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

The difference between this instruction and the AND instruction is that the result status is stored rather than the result itself.

The Z flag of the compare result register is set if the result is zero. The N flag of the result register is set if the most significant bit of the result is set. The O flag of the result register is set if the least significant bit of the result is set.

The BIT instruction features results merging, where the current value in the result register is logically combined with the new result. This allows several BIT operations to be easily cascaded for use with a branch operation. The direct copy merging operation is the default and does not need to be specified in assembler code.

|  |  |  |
| --- | --- | --- |
| MOp3 | Mnemonic |  |
| 0 | .CPY | Cd = compare result |
| 1 | .OR | Cd = Cd | compare result |
| 2 | .AND | Cd = Cd & compare result |
| 3 | .ORCM | Cd = Cd | ~compare result |
| 4 | .ANDCM | Cd = Cd & ~compare result |
| 5 to 7 |  | reserved |

**Formats Supported**: RR, RI

Example:

BIT.CPY cr1,x10,#$20 ; check bit five of register x10

BIT.AND cr1,x10,#$40 ; and bit six

BEQ cr1,target ; branch if bit is clear

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## BMM – Bit Matrix Multiply

BMM Rd, Rs1, Rs2

**Description**:

The BMM instruction treats the bits of register Rs1 and register Rs2 as an 8x8 matrix and performs a bit matrix multiply of the two registers and stores the result in the target register. An alternate mnemonic for this instruction is MOR.

**Instruction Format**: Integer R2

|  |  |
| --- | --- |
| Fn3 | Function |
| 0 | MOR |
| 1 | MXOR |
| 2 | MORT (MOR transpose) |
| 3 | MXORT (MXOR transpose) |
| 4 to 7 | reserved |

**Operation**:

for I = 0 to 7

for j = 0 to 7

Rt.bit[i][j] = (Ra[i][0]&Rb[0][j]) | (Ra[i][1]&Rb[1][j]) | … | (Ra[i][15]&Rb[15][j])

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions**: none

**Notes**:

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

## CMP – Compare

**Description**:

Compare two operand values and store the relationship in the target compare result register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

Flags are set in the compare result register as if a subtract operation were performed between operands. If the result is zero the Z flag is set. If the signed result is less than zero then the N flag is set. The carry flag C is set on unsigned overflow. The overflow flag V is set on signed overflow. Parity P is set if the exclusive or of all result bits is a one. The odd flag, O, is set if the result is odd. The remaining bits of the result register are unused.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| N | V | O | P | ~ | ~ | Z | C |

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Meaning |
| 0 | C | Carry flag, set if operation overflows |
| 1 | Z | Zero flag, set if result is zero |
| 2 | ~ | reserved |
| 3 | ~ | reserved |
| 4 | P | Parity (exclusive or of all result bits) |
| 5 | O | Odd, set if result is odd |
| 6 | V | Overflow, set if signed result overflows |
| 7 | N | Negative, set if signed result is less than zero |

The compare instructions feature results merging, where the current value in the result register is logically combined with the new result. This allows several comparisons to be easily cascaded for use with a branch operation. The direct copy merging operation is the default and does not need to be specified in assembler code.

|  |  |  |
| --- | --- | --- |
| MOp3 | Mnemonic |  |
| 0 | .CPY | Cd = compare result |
| 1 | .OR | Cd = Cd | compare result |
| 2 | .AND | Cd = Cd & compare result |
| 3 | .ORCM | Cd = Cd | ~compare result |
| 4 | .ANDCM | Cd = Cd & ~compare result |
| 5 to 7 |  | reserved |

Example: compute a0 == a1 and a2 == a3 and branch

CMP.CPY c0,a0,a1

CMP.AND c0,a2,a3

BEQ c0,target

**Formats Supported**: RR, RI

**Execution Units**: ALU

**Clock Cycles**: 0.5

## DIV – Division

**Description**:

Divide two operand values and place the result in the target register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, an immediate value. Both operands are treated as signed values.

**Formats Supported**: RR, RI

**Execution Units**: ALU

**Clock Cycles**: 67

**Exceptions**: none

## DIVU – Division Unsigned

**Description**:

Divide two operand values and place the result in the target register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, an immediate value. Both operands are treated as unsigned values.

**Formats Supported**: RR, RI

**Execution Units**: ALU

**Clock Cycles**: 67

**Exceptions**: none

## LMI – Load Middle Immediate

**Description**:

LMI sets the stage for building constants larger than can fit into the 14-bit immediate field of a register-immediate (RI) type instruction.

An immediate value is loaded into bits 14 to 38 of the destination register. The value is sign extended to 64 bits on the left and zero extended on the right. The destination register must be x1 or x2. This instruction combined with the LUI instruction and another ALU operation can be used to build a 64-bit constant in a register using only three instructions. Constants up to 39 bits may be built using only two instructions.

Note the two least significant bits of the opcode contain two constant bits.

**Formats Supported**: LUI

|  |  |  |  |
| --- | --- | --- | --- |
| Constant39..16 | Rd1 | 44h-47h | LMI |

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions:** none

## LSR – Logical Shift Right

**Description**:

Right shift one operand value by a second 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 Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

**Formats Supported**: RR, RI6

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | ~3 | 14 | ~ | Rs2 | Rs1 | Rd | 0Ch | LSR |
| 1 | ~3 | 14 | Const5..0 | | Rs1 | Rd | 0Ch | LSR |

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

## LUI – Load Upper Immediate

**Description**:

Load an immediate value into the upper 25 bits of the destination register. The lower 39 bits of the register are not affected. The destination register must be x1 or x2. This instruction combined with the LMI instruction and another ALU operation can be used to build a 64-bit constant in a register using only three instructions.

**Formats Supported**: RR, RI

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## MAJ – Majority Logic

**Description**:

Combine three operand values using majority logic and place the result in the target register. All three operands must be in registers.

**Formats Supported**: R3

**Operation:**

Rd = (Rs1 & Rs2) | (Rs1 & Rs3) | (Rs2 & Rs3)

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## MAX – Maximum of Three Values

**Description**:

Find the maximum of three values and place the result in the target register. All three operands must be in registers. To find the maximum of two values use a source register twice.

**Formats Supported**: R3

**Operation:**

if (Rs1 > Rs2 and Rs1 > Rs3)

Rd = Rs1

else if (Rs2 > Rs3)

Rd = Rs2

else

Rd = Rs3

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## MIN – Minimum of Three Values

**Description**:

Find the minimum of three values and place the result in the target register. All three operands must be in registers. To find the minimum of two values use a source register twice.

**Formats Supported**: R3

**Operation:**

if (Rs1 < Rs2 and Rs1 < Rs3)

Rd = Rs1

else if (Rs2 < Rs3)

Rd = Rs2

else

Rd = Rs3

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## MOV – Move Register to Register

**Description**:

This instruction moves from one general-purpose register to another general-purpose register. It is an alternate mnemonic for the OR instruction where Rs1 is assumed to be x0.

**Formats Supported**: RR

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

## MUL – Multiplication

**Description**:

Multiply two operand values and place the result in the target register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value. Both operands are treated as signed values.

**Formats Supported**: RR, RI

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

## MULU – Multiplication Unsigned

**Description**:

Multiply two operand values and place the result in the target register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value. Both operands are treated as unsigned values. Unsigned multiplication is commonly used to calculate array indexes.

**Formats Supported**: RR, RI

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

## OR – Bitwise ‘Or’

**Description**:

Bitwise ‘Or’ two operand values and place the result in the target register, updating status flags. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value. The immediate value is zero extended to the left from bit 14 to the machine width.

**Formats Supported**: RR, RI

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## PERM – Permute Bytes

**Description**:

This instruction allows any combination of bytes in a source register to be copied to a target register. The low order twenty-four bits of register Rs2 or twenty-four bits from a postfix constant are used to identify which source bytes are copied to the destination. The twenty-four-bit value is composed of eight three-bit fields. Field S0 indicates the source byte for target byte position 0. S1 indicates the source byte for target byte position 1. S2 to S7 work similarly for the remaining target bytes. There are many interesting possibilities with this instruction. A single source byte could be copied to all target byte positions for instance. Or the order of bytes in a word could be reversed.

**Formats Supported**: RI

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~ | | | | | Rs1 | | Rd | | | 17h | PERM |
| S7 | S6 | S5 | S4 | S3 | | S2 | | S1 | S0 | EAh | NOP |

**Formats Supported**: RR

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## PTRDIF – Difference Between Pointers

**Description**:

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

**Instruction Format**: Integer R2

**Operation**:

Rd = Abs(Rs1 – Rs2) >> Sc

**Clock Cycles**: 1

**Execution Units: Integer**

**Exceptions**:

None

## ROL – Rotate Left

**Description**:

Rotate left one operand value by a second operand value and place the result in the target register, updating status flags. The most significant bits are placed in the least significant bits. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

**Formats Supported**: RR, RI6

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | ~3 | 24 | ~ | Rs2 | Rs1 | Rd | 0Ch | ROL |
| 1 | ~3 | 24 | Const5..0 | | Rs1 | Rd | 0Ch | ROL |

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## ROR – Rotate Right

**Description**:

Rotate right one operand value by a second operand value and place the result in the target register, updating status flags. The least significant bits are placed in the most significant bits. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand may be either a register specified by the Rs2 field of the instruction, or an immediate value.

**Formats Supported**: RR, RI6

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | ~3 | 34 | ~ | Rs2 | Rs1 | Rd | 0Ch | ROR |
| 1 | ~3 | 34 | Const5..0 | | Rs1 | Rd | 0Ch | ROR |

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SUB – Subtraction

**Description**:

Subtract two operand values and place the result in the target register. Both operands must be in registers specified by the Rs1 and Rs2 fields of the instruction. There is no RI immediate form of this instruction. Subtracting an immediate value can be done with the ADD instruction.

**Formats Supported**: RR

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

## SUBF – Subtraction from Immediate

**Description**:

Subtract two operand values and place the result in the target register. The first operand must be an immediate value specified in the instruction the second value is specified by the Rs1 field of the instruction. There is no RR form for this instruction. Register based subtract from can be accomplished by swapping operands to the SUB instruction.

**Formats Supported**: RI

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

# Memory Operations

## LDB – Load Byte (8 bits)

**Description**:

Data is loaded from the memory address which is the sum of Rs1 and an immediate value or the sum of Rs1 and Rs2.. The value loaded is sign extended from bit 7 to the machine width.

**Formats Supported**: RR,RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Constant13..0 | | | | Rs1 | Rd | 80h | LDB |
| ~2 | ~ | 00h6 | Rs2 | Rs1 | Rd | 8Fh | LDB |

**Operation:**

Rt = Memory8[d+Ra]

or

Rt = Memory8[Ra+Rb]

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## LDBU – Load Byte Unsigned (8 bits)

**Description**:

Data is loaded from the memory address which is the sum of Rs1 and an immediate value or the sum of Rs1 and Rs2.. The value loaded is zero extended from bit 7 to the machine width.

**Formats Supported**: RR,RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Constant13..0 | | | | Rs1 | Rd | 81h | LDBU |
| ~2 | ~ | 01h6 | Rs2 | Rs1 | Rd | 8Fh | LDBU |

**Operation:**

Rt = Memory8[d+Ra]

or

Rt = Memory8[Ra+Rb]

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## LDO – Load Octa (64 bits)

**Description**:

Data is loaded from the memory address which is the sum of Rs1 and an immediate value or the sum of Rs1 and Rs2 scaled by one or eight.

**Formats Supported**: RR,RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Constant13..0 | | | | Rs1 | Rd | 86h | LDO |
| ~2 | S | 06h6 | Rs2 | Rs1 | Rd | 8Fh | LDO |

**Operation:**

Rt = Memory32[d+Ra]

or

Rt = Memory32[Ra+Rb]

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## LDT – Load Tetra (32 bits)

**Description**:

Data is loaded from the memory address which is the sum of Rs1 and an immediate value or the sum of Rs1 and Rs2 scaled by one or four. The value loaded is sign extended from bit 31 to the machine width.

**Formats Supported**: RR,RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Constant13..0 | | | | Rs1 | Rd | 84h | LDT |
| ~2 | S | 04h6 | Rs2 | Rs1 | Rd | 8Fh | LDT |

**Operation:**

Rt = Memory32[d+Ra]

or

Rt = Memory32[Ra+Rb]

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## LDTU – Load Tetra Unsigned (32 bits)

**Description**:

Data is loaded from the memory address which is the sum of Rs1 and an immediate value or the sum of Rs1 and Rs2 scaled by one or four. The value loaded is zero extended from bit 31 to the machine width.

**Formats Supported**: RR,RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Constant13..0 | | | | Rs1 | Rd | 85h | LDTU |
| ~2 | S | 05h6 | Rs2 | Rs1 | Rd | 8Fh | LDTU |

**Operation:**

Rt = Memory32[d+Ra]

or

Rt = Memory32[Ra+Rb]

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## LDW – Load Wyde (16 bits)

**Description**:

Data is loaded from the memory address which is the sum of Rs1 and an immediate value or the sum of Rs1 and Rs2 scaled by one or two. The value loaded is sign extended from bit 15 to the machine width.

**Formats Supported**: RR,RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Constant13..0 | | | | Rs1 | Rd | 82h | LDW |
| ~2 | S | 02h6 | Rs2 | Rs1 | Rd | 8Fh | LDW |

**Operation:**

Rt = Memory16[d+Ra]

or

Rt = Memory16[Ra+Rb]

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## LDWU – Load Wyde Unsigned (16 bits)

**Description**:

Data is loaded from the memory address which is the sum of Rs1 and an immediate value or the sum of Rs1 and Rs2 scaled by one or two. The value loaded is sign extended from bit 15 to the machine width.

**Formats Supported**: RR,RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Constant13..0 | | | | Rs1 | Rd | 83h | LDWU |
| ~2 | S | 03h6 | Rs2 | Rs1 | Rd | 8Fh | LDWU |

**Operation:**

Rt = Memory16[d+Ra]

or

Rt = Memory16[Ra+Rb]

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## STB – Store Byte (8 bits)

**Description**:

Data from Rs2 is stored to the memory address which is either the sum of Rs1 and an immediate value or the sum of Rs1 and Rs3. Both register indirect with displacement and indexed addressing are supported.

**Formats Supported**: STR, STI

**Flags Affected**: none

**Operation:**

Memory8[d+Rs1] = Rs2

or

Memory8[d+Rs1+Rs3\*Sc] = Rs2

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## STO – Store Octet (64 bits)

**Description**:

Data from Rs2 is stored to the memory address which is either the sum of Rs1 and an immediate value or the sum of Rs1 and Rs3. Both register indirect with displacement and indexed addressing are supported. Rs3 may be scaled by either one or eight before use.

**Formats Supported**: STR, STI

**Flags Affected**: none

**Operation:**

Memory64[d+Rs1] = Rs2

or

Memory64[d+Rs1+Rs3\*Sc] = Rs2

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## STT – Store Tetra (32 bits)

**Description**:

Data from Rs2 is stored to the memory address which is either the sum of Rs1 and an immediate value or the sum of Rs1 and Rs3. Both register indirect with displacement and indexed addressing are supported. Rs3 may be scaled by either one or four before use.

**Formats Supported**: STR, STI

**Flags Affected**: none

**Operation:**

Memory32[d+Rs1] = Rs2

or

Memory32[d+Rs1+Rs3\*Sc] = Rs2

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## STW – Store Wyde (16 bits)

**Description**:

Data from Rs2 is stored to the memory address which is either the sum of Rs1 and an immediate value or the sum of Rs1 and Rs3. Both register indirect with displacement and indexed addressing are supported. Rs3 may be scaled by either one or two before use.

**Formats Supported**: STR, STI

**Flags Affected**: none

**Operation:**

Memory16[d+Rs1] = Rs2

or

Memory16[d+Rs1+Rs3\*Sc] = Rs2

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

## STPTR – Store Pointer (64 bits)

**Description**:

A pointer value is stored to the memory address which is either the sum of Rs1 and an immediate value or the sum of Rs1 and Rs3. Both register indirect with displacement and indexed addressing are supported. Rs3 may be scaled by either one or eight before use. Store pointer activates the card memory associated with garbage collection.

**Formats Supported**: STR, STI

**Flags Affected**: none

**Operation:**

Memory64[d+Ra] = Rs

or

Memory64[d+Ra+Rb\*Sc] = Rs

**Execution Units**: Mem

**Clock Cycles**: 4 if data is in the cache.

**Exceptions**: none

# Flow Control (Branch Unit) Operations

## ARTS – Alternate Return from Subroutine

**Description**:

Transfer program execution to an address which is an offset from the call address stored in return address register #1 (ra1). The return address register will have been previously set by a subroutine call (JSR) operation. Also add a constant to the stack pointer. This instruction, unlike other return operations, does not affect semaphores.

**Formats Supported**: RTS

|  |  |  |  |
| --- | --- | --- | --- |
| Constant14..0 | RO9 | 11 | 24h |

The constant field is shifted left three times and zero extended before being added to the stack pointer.

The RO9 field specifies an offset in words for the return point from the calling instruction. Typically, this value would be one to cause a return to the next instruction. The RO9 field is shifted left twice before being added to the return address register (ra1). To skip over more words at the return site, adjust the RO9 field accordingly. This may be useful to skip over inline parameters or a short code sequence, perhaps for exception handling. Up to 2kB may be skipped over.

**Flags Affected**: none

**Operation:**

PC = ra1 + RO6\*4

SP = SP + Constant \* 8

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

**Notes**:

## BCC – Branch if Carry Clear

**Description**:

This instruction branches to the target address if the C flag is clear in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (!Cr.C)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BCS – Branch if Carry Set

**Description**:

This instruction branches to the target address if the C flag is set in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (Cr.C)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BEQ – Branch if Equal to Zero

**Description**:

This instruction branches to the target address if the Z flag is set in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

|  |  |  |  |
| --- | --- | --- | --- |
| Target23..2 | Cd2 | 28h | BEQ |

**Operation:**

If (Cr.Z)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BGE – Branch if Greater Than or Equal

**Description**:

This is an alternate mnemonic for the [BPL](#_BPL_–_Branch) instruction. This instruction branches to the target address if the N flag is clear in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (!Cr.N)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BLE – Branch if Less Than or Equal

**Description**:

This instruction tests two flags (Z and N) at the same time. This instruction branches to the target address if the N flag is set or the Z flag is set in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (Cr.N or Cr.Z)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BLT – Branch if Less Than

**Description**:

This is an alternate mnemonic for the [BMI](#_BMI_–_Branch) instruction. This instruction branches to the target address if the N flag is set in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (Cr.N)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BMI – Branch if Minus

**Description**:

This instruction branches to the target address if the N flag is set in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (Cr.N)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BNE – Branch if Not Equal to Zero

**Description**:

This instruction branches to the target address if the Z flag is clear in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (!Cr.Z)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BOD – Branch if Odd

**Description**:

This instruction branches to the target address if the O flag is set in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (Cr.O)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BPL – Branch if Plus

**Description**:

This instruction branches to the target address if the N flag is clear in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (!Cr.N)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BVC – Branch if Overflow Clear

**Description**:

This instruction branches to the target address if the V flag is clear in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (!Cr.V)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BVS – Branch if Overflow Set

**Description**:

This instruction branches to the target address if the V flag is set in the specified condition register, otherwise program execution continues with the next instruction. The target address is an absolute address. The target field is loaded into the low order 24 bits of the program counter. The remaining bits of the program counter are unchanged. The branch range is 16MB.

**Formats Supported**: BR

**Operation:**

If (Cr.V)

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

## BRK – Break

**Description**:

This instruction initiates the processor debug routine. The processor enters debug mode. The cause code register is set to the value specified in the instruction. Interrupts are disabled and register set #31 is selected. The program counter is reset to $FFF…FFEC 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 DEPC register.

**Formats Supported**: BRK

|  |  |  |  |
| --- | --- | --- | --- |
| Constant16 | Cause8 | 00h | BRK |

**Operation:**

PMSTACK = (PMSTACK << 4) | 10

RSSTACK = (RSSTACK << 5) | 31

CAUSE = Const8

DEPC = PC

PC = $FFFFFFFFFFFFFFFC

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## RTD – Return from Debug Mode

**Description**:

Restore the previous interrupt enable setting, register set and operating mode and transfer program execution back to the address in the debug exception link register. One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RET

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Constant14..0 | 53 | Sema6 | 11 | 23h |

**Flags Affected**: none

**Operation:**

OMS = OMS >> 4

PLS = PLS >> 8

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = DEPC

**Execution Units**: Mem

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## HRET – Return from Hypervisor Mode Subroutine

**Description**:

Restore the previous interrupt enable setting, register set and operating mode and transfer program execution back to the address in the hyper-visor exception link register. One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RET

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Constant14..0 | 23 | Sema6 | 11 | 23h |

The constant field is not used.

**Flags Affected**: none

**Operation:**

OMS = OMS >> 4

PLS = PLS >> 8

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = MEPC

**Execution Units**: Mem

**Clock Cycles**:

**Exceptions**: none

**Notes:**

## IRET – Return from Interrupt Subroutine

**Description**:

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

**Formats Supported**: RET

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Constant14..0 | 43 | Sema6 | 11 | 23h |

The constant field is not used.

**Flags Affected**: none

**Operation:**

OMS = OMS >> 4

PLS = PLS >> 8

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = IEPC

**Execution Units**: Mem

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## JALR – Jump and Link Register

**Description**:

Store the return address in the specified link register then jump to the address specified as the sum of register Rs1 and a 18-bit immediate constant. The link register must be one of x0 to x3. A 17-bit constant field is shifted left once before use.

**Formats Supported**: JAL

**Flags Affected**: none

**Operation:**

Lk = NextPC

PC = Rs1+Constant18\*2

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## JMP – Jump

**Description**:

This instruction jumps to a target address. The address specified is an absolute address. The address range is 24 bits 16MB. The jump instruction should be used in preference to branch instructions as it will not occupy space in the predictor tables.

**Formats Supported**: JMP

**Flags Affected**: none

**Operation:**

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 0.5

**Exceptions**: none

**Notes**:

## JSR – Jump to Subroutine

**Description**:

Store the address of the JSR instruction in the specified return address register (ra0 or ra1) then jump to the address specified in the instruction. The address range is 22 bits shifted left twice or 16MB. The return address register is assumed to be ra0 if not otherwise specified. The JSR instruction does not require space in branch predictor tables.

**Formats Supported**: JSR

**Flags Affected**: none

**Operation:**

Ra = PC

PC = {PC[63:24],Target[23:2],00}

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

**Notes**:

## MRET – Return from Machine Mode Subroutine

**Description**:

Restore the previous interrupt enable setting, register set and operating mode and transfer program execution back to the address in the machine exception link register. One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RET

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Constant14..0 | 33 | Sema6 | 11 | 23h |

The constant field is not used.

**Flags Affected**: none

**Operation:**

OMS = OMS >> 4

PLS = PLS >> 8

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = MEPC

**Execution Units**: Mem

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## RTS – Return from Subroutine

**Description**:

Transfer program execution to an address which is the sum of a value stored in a return register (ra0) and an offset (RO9) specified in the instruction. The return address register will have been previously set by a subroutine call JSR operation. Also add a constant to the stack pointer. This instruction, unlike other return operations, does not affect semaphores. The assembler assumes ra0 with an offset of one word is used unless otherwise specified.

The RO9 field is used to return to a point past the normal return point of the next instruction. This is useful in some circumstances such as the presence of inline subroutine parameters or exception handling code.

**Formats Supported**: RTS

|  |  |  |  |
| --- | --- | --- | --- |
| Constant14..0 | RO9 | 01 | 24h |

The constant field is shifted left three times and zero extended before being added to the stack pointer.

The RO9 field specifies an offset in words for the return point from the calling instruction. Typically, this value would be one to cause a return to the next instruction. The RO9 field is shifted left twice before being added to the return address register (ra0). To skip over more words at the return site, adjust the RO9 field accordingly. This may be useful to skip over inline parameters or a short code sequence, perhaps for exception handling.

**Flags Affected**: none

**Operation:**

PC = Ra + RO9 \* 4

SP = SP + Constant \* 8

**Examples:**

RTS ; return from the subroutine

RTS #$200 ; return and add $200 to the stack pointer

RTS ra1,#$400 ; return using ra1 instead of ra0, add onto stack pointer

**Execution Units**: Branch

**Clock Cycles**: 1

**Exceptions**: none

**Notes**:

## SRET – Return from Supervisor Mode Subroutine

**Description**:

Restore the previous interrupt enable setting, register set and operating mode and transfer program execution back to the address in the supervisor exception link register. One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RET

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Constant14..0 | 13 | Sema6 | 11 | 23h |

The constant field is not used.

**Flags Affected**: none

**Operation:**

OMS = OMS >> 4

PLS = PLS >> 8

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = SEPC

**Execution Units**: Mem

**Clock Cycles**:

**Exceptions**: none

**Notes:**

## U1RET – Return from User Mode Subroutine

**Description**:

Restore the previous interrupt enable setting, register set and operating mode and transfer program execution back to the address in the user1 exception link register. One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RTI

**Flags Affected**: none

**Operation:**

OMS = OMS >> 4

PLS = PLS >> 8

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = U1EPC

**Execution Units**: Mem

**Clock Cycles**:

**Exceptions**: none

**Notes:**

## U2RET – Return from User Mode Subroutine

**Description**:

Restore the previous interrupt enable setting, register set and operating mode and transfer program execution back to the address in the user2 exception link register. One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RTI

**Flags Affected**: none

**Operation:**

OMS = OMS >> 4

PLS = PLS >> 8

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = U2EPC

**Execution Units**: Mem

**Clock Cycles**:

**Exceptions**: none

**Notes:**

## WAI – Wait for Interrupt

**Description**:

The WAI instruction waits for an interrupt to occur stopping the processor clock until an interrupt occurs. This instruction is like the PFI instruction except that it stops and waits for an interrupt whereas PFI doesn’t wait. WAI does not check for a non-maskable (NMI) interrupt or a reset (RST).

**Formats Supported**: WAI

**Flags Affected**: none

**Operation:**

If (IRQ)

Cause Code = 50h | IRQ Level

OLS = OLS << 3

DLS = DLS << 3

IMS = (IMS << 3) | 7

PLS = PLS << 13

XLR = PC + 1;

PC = $FFFFFFFFE0000

Else

PC = PC (clock stopped)

**Execution Units**: Fetch stage

**Clock Cycles**:

**Exceptions**: none

**Notes**:

# Floating Point Instructions

## Overview

The floating-point unit provides basic floating-point operations including addition, subtraction, multiplication, division, square root, and float to integer and integer to float conversions. The core contains two identical floating-point units. Only 64-bit precision floating-point operations are supported. The core features results caching, if the same operation is performed on the same values as is present in the cache then the result is returned in a single clock cycle.

The rounding mode is normally specified directly in the instruction. However, if the instruction indicates to use dynamic rounding mode then the rounding mode in the floating-point control and status register is used.

**Representation**

The floating-point format is like an IEEE-754 representation for double precision. Briefly,

**64-bit Precision Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 63 | 62 | 61 52 | 51 0 |
| SM | SE | Exponent | Mantissa |

SM – sign of mantissa

SE – sign of exponent

The exponent and mantissa are both represented as two’s complement numbers, however the sign bit of the exponent is inverted.

|  |  |
| --- | --- |
| SeEEEEEEEEEE |  |
| 11111111111 | Maximum exponent |
| …. |  |
| 01111111111 | exponent of zero |
| …. |  |
| 00000000000 | Minimum exponent |

The exponent ranges from -1023 to +1024

### Short Immediates

Some floating-point operations allow a short immediate format to be used as the second operand. These instructions include FADD, FSUB, FCMP, FMUL, FDIV, FSEQ, FSNE, FSLT, FSLE. The short immediate format assumes a positive number with four bits for the exponent and four for the mantissa. The range of these numbers is 2-7 to 28 with four bits of precision. The short immediate is converted into a 52-bit floating-point number before use.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 7 | 6 4 | 3 0 |
| 0 | SE | Exp. | Mant. |

## FABS – Floating Absolute Value

**Description:**

Take the absolute value of a floating-point number in register Frs1 and places the result into target register Frd. The sign bit (bit 63) of the register is set to zero. No rounding of the number occurs.

**Instruction Format: FLT1**

**Clock Cycles: 1**

**Execution Units:** Floating Point

## FADD – Floating point addition

**Description:**

Add two floating point numbers in registers Frs1 and Frs2 or a short immediate value and place the result into target register Frd. The result is rounded according to the selected rounding mode in the instruction. If the rounding mode is encoded as 7 then the rounding mode used in the floating-point status register is used.

**Instruction Format: FLT2**

**Clock Cycles: 6**

**Execution Units:** Floating Point

## FCLASS – Classify Value

**Description**:

FCLASS classifies the value in register Frs1 and returns the information as a bit vector in the integer register Rd.

|  |  |
| --- | --- |
| Bit | Meaning |
| 0 | 1 = negative infinity |
| 1 | 1 = negative number |
| 2 | 1 = negative subnormal number |
| 3 | 1 = negative zero |
| 4 | 1 = positive zero |
| 5 | 1 = positive subnormal number |
| 6 | 1 = positive number |
| 7 | 1 = positive infinity |
| 8 | 1 = signalling nan |
| 9 | 1 = quiet nan |
| 10 to 62 | not used |
| 63 | 1 = negative, 0 = positive number |

## FCMP - Float Compare

**Description:**

The register compare instruction compares two registers as floating-point values and sets the compare result register as a result.

The compare instructions feature results merging, where the current value in the result register is logically combined with the new result. This allows several comparisons to be easily cascaded for use with a branch operation. The direct copy merging operation is the default and does not need to be specified in assembler code.

|  |  |  |
| --- | --- | --- |
| MOp3 | Mnemonic |  |
| 0 | .CPY | Cd = compare result |
| 1 | .OR | Cd = Cd | compare result |
| 2 | .AND | Cd = Cd & compare result |
| 3 | .ORCM | Cd = Cd | ~compare result |
| 4 | .ANDCM | Cd = Cd & ~compare result |
| 5 to 7 |  | reserved |

**Instruction Format: FLT2**

**Clock Cycles:** 0.5

**Execution Units:** Floating Point

**Operation:**

if Frs1 < Frs2

Cr.N = 1

else if Frs1 = Frs2

Cr.Z = 1

else

Cr.N = 0

Cr.Z = 0

if unordered (Frs1, Frs2)

Cr.V = 1

else

Cr.V = 0

## FCX – Clear Floating-Point Exceptions

**Description:**

This instruction clears floating point exceptions. The Exceptions to clear are identified as the bits set in the union of integer register Rs1 and an immediate field in the instruction. Either the immediate or Rs1 should be zero.

**Instruction Format: FLT1**

**Execution Units:** All Floating Point

**Operation:**

**Exceptions:**

|  |  |
| --- | --- |
| Bit | Exception Enabled |
| 0 | global invalid operation clears the following:   * division of infinities * zero divided by zero * subtraction of infinities * infinity times zero * NaN comparison * division by zero |
| 1 | overflow |
| 2 | underflow |
| 3 | divide by zero |
| 4 | inexact operation |
| 5 | summary exception |

## FDX – Floating Disable Exceptions

**Description:**

This instruction disables floating point exceptions. The Exceptions disabled are identified as the bits set in the union of integer register Rs1 and an immediate field in the instruction. Either the immediate or Rs1 should be zero. Exceptions will not be disabled until the instruction commits and the state of the machine is updated. This instruction should be followed by a synchronization instruction (FSYNC) to ensure that following floating point operations recognize the disabled exceptions.

|  |  |
| --- | --- |
| Bit | Exception Enabled |
| 0 | global invalid operation clears the following:   * division of infinities * zero divided by zero * subtraction of infinities * infinity times zero * NaN comparison * division by zero |
| 1 | overflow |
| 2 | underflow |
| 3 | divide by zero |
| 4 | inexact operation |
| 5 | summary exception |

**Instruction Format: FXX**

**Clock Cycles: 2**

**Execution Units:** Floating Point

## FDIV – Floating point divide

**Description:**

Divide two floating point numbers in registers Frs1 and Frs2 and place the result into target register Frd.

**Instruction Format: FLT2**

**Clock Cycles: 28 (est).**

**Execution Units:** Floating Point

## FEX – Floating Enable Exceptions

**Description:**

This instruction enables floating point exceptions. The Exceptions enabled are identified as the bits set in the union of integer register Rs1 and an immediate field in the instruction. Either the immediate or Rs1 should be zero. Exceptions won’t be enabled until the instruction commits, and the state of the machine is updated. This instruction should be followed by a synchronization instruction (FSYNC) to ensure that following floating point operations recognize the enabled exceptions.

**Instruction Format: FXX**

**Clock Cycles: 2**

**Execution Units:** Floating Point

## FINITE – Number is Finite

**Description:**

Test the value in Frs1 to see if it’s a finite number and return Z=1 or Z = 0 in compare result register Crt.

**Instruction Format: FLT1**

**Clock Cycles: 1**

**Execution Units:** Floating Point

**Example**:

finite $cr1,$f7

## FMUL – Floating point multiplication

**Description:**

Multiply two floating point numbers in registers Frs1 and Frs2 and place the result into target register Frd.

**Instruction Format: FLT2**

**Clock Cycles: 7**

**Execution Units:** Floating Point

## FNABS – Floating Negative Absolute Value

**Description:**

Take the negative absolute value of the floating-point number in register Fs1 and place the result into target register Frd. The sign bit (bit 63) of the register is set to one. No rounding of the number occurs.

**Instruction Format: FLT1**

**Clock Cycles: 1**

**Execution Units:** Floating Point

## FNEG – Floating Negative Value

**Description:**

Negate the value of the floating-point number in register Frs1 and place the result into target register Frd. The sign bit (bit 63) of the register is inverted. No rounding of the number occurs.

**Instruction Format: FLT1**

**Clock Cycles: 1**

**Execution Units:** Floating Point

## FRES – Reciprocal Estimate

**Description:**

This function uses a 1024 entry 16-bit precision lookup table to create a piece-wise approximation of the reciprocal and linear interpolation to approximate the reciprocal of the value in Frs1. The value is returned in Frd as a 64-bit floating-point value. The value returned is accurate to about eight bits.

**Instruction Format: FLT1**

**Clock Cycles: 5**

**Execution Units:** Floating Point

## FRSQRTE – Float Reciprocal Square Root Estimate

**Description:**

Estimate the reciprocal of the square root of the number in register Frs1 and place the result into target register Frd.

**Instruction Format: FLT1**

**Clock Cycles: 5**

**Execution Units:** Floating Point

**Notes**:

The estimate is only accurate to about 3%. The estimate is performed in single precision (32-bit) floating point, then converted to a 64-bit format. That means that input values must in the range of a 32-bit floating point number. Values outside of this range will return infinity or zero as a result.

Taking the reciprocal square root of a negative number results in a Nan output.

## FSIGN – Floating Sign

**Description:**

FSIGN returns a value indicating the sign of the floating-point number. If the value is zero, the target register is set to zero. If the value is negative the target register is set to the floating-point value -1.0. Otherwise the target register is set to the floating-point value +1.0. No rounding of the result occurs.

**Instruction Format: FLT1**

**Clock Cycles: 1**

**Execution Units:** Floating Point

## FSQRT – Floating point square root

**Description:**

Take the square root of the floating-point number in register Frs1 and place the result into target register Frd. The sign bit (bit 63) of the register is set to zero. This instruction can generate NaNs.

**Instruction Format: FLT1**

**Clock Cycles: 64 (est).**

**Execution Units:** Floating Point

## FSUB – Floating point subtraction

**Description:**

Subtract two floating-point numbers in registers Frs1 and Frs2 and place the result into target register Frd.

**Instruction Format: FLT2**

**Clock Cycles: 6**

**Execution Units:** Floating Point

## FSYNC -Synchronize

**Description**:

All floating-point instructions before the FSYNC are completed and committed to the architectural state before floating-point instructions after the FSYNC are issued. This instruction is used to ensure that the machine state is valid before subsequent instructions are executed.

**Instruction Format**: FSYNC

**Clock Cycles**: varies depending on queue contents

## FTOI – Floating Convert to Integer

**Description:**

Convert the floating-point value in Frs1 into an integer and place the result into a target register. The target register may be either another floating-point register or an integer register. If the result overflows the value placed in the target is a maximum integer value. Note that the result in the target register is no longer of a floating-point representation.

**Instruction Format: FLT1**

**Clock Cycles: 3**

**Execution Units:** Floating Point

## FTRUNC – Truncate Value

**Description**:

The FTRUNC instruction truncates off the fractional portion of the number leaving only a whole value. For instance, ftrunc(1.5) equals 1.0. Ftrunc does not change the representation of the number. To convert a value to an integer in a fixed-point representation see the FTOI instruction.

**Instruction Format**: FLT1

**Clock Cycles**: 1

**Execution Units:** Floating Point

## ISNAN – Is Not a Number

**Description:**

Test the value in Frs1 to see if it’s a nan (not a number) and return true Z=1 or false Z=0 in compare result register Ct.

**Instruction Format: FLT1**

**Clock Cycles: 1**

**Execution Units:** Floating Point

**Example**:

isnan $cr1,$f7

## ITOF – Convert Integer to Float

**Description:**

Convert the integer value in Rs1 into a floating-point value and place the result into target register Ft. Rs1 is from either the floating-point register set or the integer register set, Frd is in the floating-point register set. Some precision of the integer converted may be lost if the integer is larger than 52 bits. 64-bit precision floating-point values only have a precision of 52 bits.

**Instruction Format: FLT1**

**Clock Cycles: 3**

**Execution Units:** Floating Point

# Oddball

An assortment of instructions that are not executed on one of the regular functional units.

## CACHE – Cache Command

CACHE Cmd, [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.

**Instruction Formats**: CACHE

**Commands:**

|  |  |  |
| --- | --- | --- |
| IC2 | Mne. | Operation |
| 0 | NOP | no operation |
| 1 | invline | invalidate line associated with given address |
| 2 | invall | invalidate the entire cache (address is ignored) |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| 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) |
|  |  |  |

Notes: