# Introduction

RTF64 is another attempt by the author at a 64-bit design.

## 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-entry posit arithmetic register file
* 4 8-bit compare results registers
* 2 dedicated return address registers
* 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

rtf64 is a work in progress beginning in September 2020. rtf64 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 rtf64

#### 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 RTF64 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. RTF64 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 RTF64 operate on 64-bit values, to perform the same operations in the 6502 would require many more bytes. Several instructions in the RTF64 are more powerful than what can be found in the 6502.

#### Registers

The RTF64 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 RTF64 has available many more transistors than were available to the 6502 design. The RTF64 has many special purpose registers. The 6502 does not have any.

#### Instructions

The 6502 uses relative branches to allow a code dense instruction encoding. Since there are enough bits available in the RTF64 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.

The 6502 offers only basic instructions (ADD, SUB, CMP, AND, ORA, EOR, LDA, STA) as examples. There are no complex instructions in the 6502 ISA. All instructions execute within a handful of clock cycles. the RTF64 has a ton of instructions compared to a 6502. It supports floating point and posit arithmetic.

The 6502 is an accumulator-based architecture that allows one memory-based operand for most instructions. RTF64 is register based and the only instructions accessing memory are load and store instructions.

### Case Comparison RISCV

RISCV vs RTF64

#### 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. The RTF64 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 the RTF64 design constants are specified using magic numbers in the constant field of an instruction. This leads to a code dense form of constants requiring no additional registers or instructions to encode. Constants up to 128 bits may be easily accommodated. RISCV does not really provide much for building constants over 32 bits.

#### Instructions

RISCV does not indexed addressing modes in the standard implementation. Indexed addressing is accomplished when required using additional instructions and registers to calculate the effective address. RTF64 directly supports indexed addressing with an optionally scaled index register. When indexed addressing is required RTF64 is more code dense than RISCV. However indexed addressing is not used that often.

#### Register File

RISCV does almost everything using general-purpose registers. This paradigm increases the pressure on the register file. In the RTF64 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.

### Case Comparison MMIX

#### Instruction Format

MMIX comes across as more of a pedantic processor design. MMIX instructions are structure simply for the most part using a 32-bit format divided into four byte regions. The author assumes this is primarily to enhance the readability of instructions. The constant field is often limited to eight bits.

#### Register File

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

### Case Comparison PowerPC

#### Overview

RTF64 is perhaps most like the PowerPC in its design. PowerPC is also a 32-register design.

#### Instruction Format

The PowerPC like RTF64 uses a fixed 32-bit instruction format.

#### Instructions

The PowerPC supports indexed addressing like the RTF64 although index scaling is not present.

#### Registers

The PowerPC has a dedicated link register and eight condition code registers. RTF64 is similar with a dedicated pair of link registers and four compare result registers. While RTF64 has fewer registers for condition codes it is not clear to the author that any more offer a benefit. The PowerPC also has a loop count register used for counted loops. RTF64 does not have a loop count register.

The general register array is the same size – 32-entry.

**Nomenclature**

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

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

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

# Development Aspects

## Device Target

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

## Implementation Language

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

# Programming Model

## **Registers**

### Overview

The RTF64 ISA is a 32-register machine with a separate register file for integer, floating-point, or posit arithmetic. There are 32 sets of integer registers. There are many control and status (CSR) registers which hold an assortment of specific values relevant to processing.

### Register Sets

Because of the use of block memory in an FPGA there are multiple integer register sets available. There are several register sets dedicated to different operating modes of the processor. The remaining register sets are available for general use.

|  |  |
| --- | --- |
| Register Set | Associated Usage |
| 0 to 25 | general usage |
| 26 | user exceptions |
| 27 | supervisor |
| 28 | hypervisor |
| 29 | machine |
| 30 | interrupt |
| 31 | debug |

Each register set includes integer registers plus return address, compare and exception address registers for which there are also multiple sets.

### 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 |  |
|  |  |  |
| eip | exceptioned instruction pointer |  |

### 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. They are also used to store the Boolean results of a set 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 or set instructions. Many other instructions may optionally update one of the compare results registers depending on the instruction. This is option is encoded as the ‘r’ record bit in the instruction.

The set instructions may set the carry bit of a compare results register to a Boolean value of (0 or 1) which may then later be tested with the BCS or BCC instructions.

The carry bit of compare results registers may be used as an operand for some integer operations. For instance, left shift and add operations can make use of the carry bit of a compare result register.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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.*

*The presence of compare results register independent of the general register file helps to reduce the pressure on the general register file. It effectively adds to the register set available.*

### 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 ([RTS](#_RTS_–_Return), [ARTS](#_ARTS_–_Alternate)) 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.*

*Like the compare results registers the return address registers independent of the general- purpose registers helps reduce register pressure. It effectively makes two more registers available.*

### Instruction pointer

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

*There is little reason to increment higher order instruction pointer bits. That would just waste hardware. Most code fragments are small and the JSR instruction is used to set the instruction pointer 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. With the use of a virtual memory system and the fact that individual modules are often far less than 16MB in size the limitations of the instruction pointer are not great.*

|  |  |
| --- | --- |
| 63 24 | 23 0 |
| IP High40 | IP 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.

*The author considered having the stack pointer as an independent register but that would require replicating a number of instructions (add, sub, and, or, etc.) just for the stack pointer. The author feels it is better to keep the stack pointer general-purpose in nature so that it may leverage the usage of the existing instruction set. This design is primarily a load / store architecture. There are no special instructions (push or pop) for manipulating the stack and hence no requirement for a special purpose register.*

### 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 popular architectures have fewer base registers. They may also be called address space or segment registers.*

*The data base register (sg0 to sg7) are chosen to be consecutive starting at zero so that they may be setup to occupy up to ½ of the memory space with consecutive addressing. Addresses may increment onto the next data base register.*

*The stack is often setup to be part of the data area and may share the same base address.*

*There are multiple code segment registers to allow transferring execution between several independent modules.*

There is more descriptive text of base registers in the section on memory management.

### Register Tag Map

The register tag map associates a register with a unique tag used for dependency checking in the core. Logic in the core uses 128-bit wide bit arrays with a bit reserved for each register. The register tags from the register tag map are also used with the MOV instruction to move values between registers.

|  |  |  |
| --- | --- | --- |
| Tag | Associated Register |  |
| 0 to 31 | GP0 to GP31 | general purpose registers |
| 32 to 63 | FP0 to FP31 | floating point registers |
| 64 to 95 | PS0 to PS31 | posit arithmetic registers |
| 96 to 97 | RA0, RA1 | return address registers |
| 98 to 102 | reserved |  |
| 103 | EIP | exception instruction pointer |
| 104 to 111 | reserved |  |
| 112 to 115 | CR0 to CR3 | compare result registers |
| 116 to 123 | reserved |  |
| 124 | reserved |  |
| 125 | CR <all> | all compare results registers |
| 126 | reserved | not used |
| 127 | none | instruction without a target |

## Control and Status Registers

### Overview

There are numerous special purpose control and status registers in the design. Some registers are present to store variables for performance reasons that would otherwise be stored in main memory.

### U\_SEMA (CSR 0x00C) Semaphores

This register is available for user semaphores or flag use. Bits in this CSR may be set or cleared with one of the CSRxx instructions. This register has individual bit set / clear capability.

### S\_TID (CSR 0x110)

This CSR register is reserved for use to contain the task id for the currently running task.

### S\_ASID – (CSR 0x11F)

This register contains the address space identifier (ASID) or memory map index (MMI). The ASID is used in this design to select (index into) a memory map in the paging tables.

### S\_KEYS – (CSR 0x120 to 0x122)

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. Keyed memory enhances the security and reliability of the system.

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

### Control Register Zero (CSR #300)

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.

### M\_HARTID (0x301)

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.

### M\_TICK (0x302)

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.

### M\_CAUSE (0x306)

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.

### M\_BADADDR (CSR 0x307)

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.

### M\_BAD\_INSTR (CSR 0x30B)

This register contains a copy of the exceptioned instruction.

### M\_SEMA (CSR 0x30C) 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 |
| 7-63 | currently unassigned |

### M\_TVEC (0x330 to 0x335)

These registers contain the address of the exception handling routine for a given operating level. TVEC[5] (0x335) is used directly by hardware to form an address of the debug routine. The lower eight bits of TVEC[5] are not used. The lower bits of the exception address are determined from the operating level. TVEC[1] to TVEC[5] are used by the REX instruction.

### M\_PM\_STACK (0x340)

This register contains an eight-entry operating mode and interrupt mask stack. When an exception or interrupt occurs, this register is shifted to the left by four bits and the low order bits are set according to the exception mode, when an RTI instruction is executed this register is shifted to the right by four bits. On RTI the last stack entry is set to $B masking all interrupts on stack underflow. The low order four bits represent the current operating mode and interrupt mask. Only the low order 32 bits of the register are implemented.

### M\_RS\_STACK (0x343)

This register contains an eight-entry register set selection stack. When an exception or interrupt occurs, this register is shifted to the left by five bits and the exception register set is inserted into the low order five bits. When an RTx instruction is executed this register is shifted to the right by five bits. On RTx the last stack entry will be set to 31 which will select register set #31 (the debug register set) on stack underflow. Only the low order 40 bits of the register are implemented.

### M\_EIP (0x348)

This register contains the interrupt or exception instruction pointer register.

### 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 |

### M\_INFO (0x3F0 to 0x3FF)

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

### D\_REGSET (0x512)

This register controls which register set is used for register access for each register field in an instruction. If the bit in the regset CSR is clear then the current register is selected, otherwise the previous register set is selected.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 63 5 | 4 | 3 | 2 | 1 | 0 |
| reserved | EIP | Rs3 | Rs2 | Rs1 | Rd |

Updating the regset CSR returns the current value of the CSR in the newly selected Rd register set.

### D\_DBADx (CSR 0x518 to 0x51B) Debug Address Register

These registers contain addresses of instruction or data breakpoints.

|  |
| --- |
| 63 0 |
| Address 63..0 |

### D\_DBCR (CSR 0x51C) Debug Control Register

This register contains bits controlling the circumstances under which a debug interrupt will occur.

|  |  |  |  |
| --- | --- | --- | --- |
| bits |  |  |  |
| 3 to 0 | Enables a specific debug address register to do address matching. If the corresponding bit in this register is set and the address (instruction or data) matches the address in the debug address register then a debug interrupt will be taken. |  |  |
| 17, 16 | This pair of bits determine what should match the debug address register zero in order for a debug interrupt to occur.   |  |  |  | | --- | --- | --- | | 17:16 |  |  | | 00 | match the instruction address |  | | 01 | match a data store address |  | | 10 | reserved |  | | 11 | match a data load or store address |  | |  |  |
| 19, 18 | This pair of bits determine how many of the address bits need to match in order to be considered a match to the debug address register. These bits are ignored when matching instruction addresses, which are always half-word aligned.   |  |  |  | | --- | --- | --- | | 19:18 |  | Size | | 00 | all bits must match | byte | | 01 | all but the least significant bit should match | char | | 10 | all but the two LSB’s should match | tetra | | 11 | all but the three LSB’s should match | octa | |  |  |
| 23 to 20 | Same as 16 to 19 except for debug address register one. |  |  |
| 27 to 24 | Same as 16 to 19 except for debug address register two. |  |  |
| 31 to 28 | Same as 16 to 19 except for debug address register three. |  |  |
| 55 to 62 | These bits are a history stack for single stepping mode. An exception will automatically disable single stepping mode and record the single step mode state on stack. Returning from an exception pops the single step mode state from the stack. |  |  |
| 63 | This bit enables SSM (single stepping mode) |  |  |

### D\_DBSR (CSR 0x51D) - Debug Status Register

This register contains bits indicating which addresses matched. These bits are set when an address match occurs and must be reset by software.

|  |  |
| --- | --- |
| bit |  |
| 0 | matched address register zero |
| 1 | matched address register one |
| 2 | matched address register two |
| 3 | matched address register three |
| 63 to 4 | not used, reserved |

## 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. One of the exception return instructions (RTI, RTD, RTE) will switch the operating level back to what it was prior to the interrupt or exception.

# Memory Access Alignment

The core supports unaligned data memory access; however, it does not guarantee the atomicity of the access.

# 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.

Associated with each memory page and stored in its own table is a memory key. The memory key is matched against the keyset in CSR registers. Access to the memory page is allowed only if one of the keys in the keyset matches the memory key, or if the page is marked generally accessible with the special key of zero. Memory may be shared between apps that share the same memory key.

## 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, instruction pointer 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 14 bits before being added to the virtual address. This gives potentially a 74-bit address space.

Note there is no limit or bound register. Access is limited by what is mapped into the segment. Pages that are inaccessible use the reserved page number of all ones.

## 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 16kB 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 4096 pages required to map the 256MB address space. Any individual app is limited to maximum of 64MB (one quarter 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. Pages that are inaccessible use the reserved physical page number of all ones.

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, in this case 12 bits. 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 14 bits of an address pass through both linear address generation and paging unchanged.

### The 16kB Page

Many memory systems use a 4kB page size. A 16kB 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 16kB page size there are still 4096 pages of memory available in a map.

*The author was tempted to divide the page mapping table into several different regions capable of mapping different amounts of the address space (small, medium, and large areas). This potentially could allow more memory maps to be present while at the same time not increasing the page table size. However, it would add extra complexity to the memory system which is currently simpler in nature.*

### The MVMAP Instruction

The memory mapping table is managed with a dedicated instruction - [MVMAP](#_MVMAP_–_Move).

*While the memory mapping table could have been managed with CSRs or possibly be mapped into the main memory space, the author feels that having a dedicated instruction makes the software managing the tables simpler and cleaner.*

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.

## Key Memory

### Overview

Associated with each page of memory is a memory key. To access a page of memory the memory key must match with one of the keys in the applications keyset. The keyset is maintained in the keys CSRs. The key size of 20 bits is a minimum size recommended for security purposes.

Since each page of memory requires a 20-bit key and there are 16,384 pages in the system a 40kB memory is required to store keys (10 block rams).

### The SETKEY Instruction

To access the key memory, the SETKEY instruction is used. This instruction may set or get the current key for a memory page. For SETKEY the Rs1 register contains a value in the following format:

|  |  |  |  |
| --- | --- | --- | --- |
| 63 46 | 45 32 | 31 20 | 19 0 |
| ~ | Physical Page Number | ~ | Key Value |

More detail is present in the instruction description.

## Card Memory

### Overview

Also present in the memory system is Card memory. The card memory is a telescopic memory which reflects with increasing detail where in the memory system a pointer write has occurred. This is for the benefit of garbage collection systems. Card memory is updated when a pointer value is stored to memory. (The store pointer [STPTR](#_STPTR_–_Store) instruction is used to do this).

*Garbage collection has become more prevalent in many systems and the author feels it makes sense to provide some hardware support for this function.*

### Organization

Memory is divided into 128-byte card memory pages. Each card has a single bit recording whether a pointer store has taken place in the corresponding memory area. To cover a 256MB memory system 256kB card memory (64 block rams) is required at the outermost layer. The outer most 256kB card memory layer is itself divided into 128-bit regions and covered with a 256B (2048 bit) memory. The 256B memory is further resolved into 128-bit regions and covered with a 2B (16 bit) memory.

Table showing increasing resolution of telescopic memory.

|  |  |  |  |
| --- | --- | --- | --- |
| Layer |  | | |
| 0 | 16 bits |  | |
| 1 | 2048 bits | |  |
| 2 | 2097152 bits | | |

*The choice of 128 comes from the desire to minimize the amount of card memory required. Card memory is implemented using dedicated block memory in an FPGA and there are only so many block memories available. The author would have preferred a better resolution, but it may consume to many memory blocks.*

### Operation

As a program progresses it writes pointer values to memory using the store pointer instruction. Storing a pointer triggers an update to all the layers of card memory corresponding to the main memory location written. A bit is set in each layer of the card memory system corresponding to the memory location of the pointer store.

The garbage collection system can very quickly determine where pointer stores have occurred and skip over memory that has not been modified.

Accessing Card Memory

Card memory is accessible with the [GCCLR](#_GCCLR_–_Garbage) instruction. The GCCLR instruction may be used to return the status of the card memory then clear it.

# Instruction Formats

## Constants

Constants which will not fit into the 13-bit constant field of an instruction are encoded using the LMI and LUI instructions. Constants up to 64-bits in size may be built using only three instructions. 38-bit constants can be built using only two instructions. The focus is on the LMI instruction which can provide constants up to 38 bits in size. It supplies the middle 26 constant bits sign extended to 64 bits. The LUI instruction can provide the remaining 25 bits required for a 64-bit constant. Note that the LUI instruction must read the destination register to provide unmodified low order bits. The capability to read the destination register is also used by other instructions. This read port comes “for free” in an FPGA design.

*The author almost had the design using magic numbers and additional constant words to implement extended constants. It appeared to be appealing for code density reasons and software implementation, but it turns out there was no code density to be gained unless the constant was extremely large. Making use of the LUI and LMI constant building instructions make software slightly more complex, but the benefits from a hardware perspective are probably worth it.*

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | | | | | | | | | | | | | | | | | | |  | | Opcode8 |  |
| Constant16 | | | | | | | | | | | | | | | | | | | | | | Cause8 | | | | | | 00h | BRK |
| r | | Funct3 | | | | | | Rs3 | | | | | | | | Rs2 | | | | | Rs1 | | Rd | | | | | 01h | {Reg3A} |
| r | | Funct5 | | | | | | | | | | Fmt3 | | | | Rs2 | | | | | Rs1 | | Rd | | | | | 02h | {Reg2} |
| r | | 35 | | | | | | | | | | Fn3 | | | | Rs2 | | | | | Rs1 | | Rd | | | | | 02h | BMM |
| r | | 135 | | | | | | | | | | Fmt3 | | | | ~ | | s2 | | d2 | Rs1 | | Rd | | | | | 02h | MOV |
| r | | Funct3 | | | | | | Rs3 | | | | | | | | Rs2 | | | | | Rs1 | | Rd | | | | | 03h | {Reg3B} |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 04h | ADD |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 05h | SUBF |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 06h | MUL |
| ~ | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | | Cd2 | | 07h | CMP |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 08h | AND |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 09h | OR |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 0Ah | EOR |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | | Cd2 | | 0Bh | BIT |
| r | | ~3 | | | | | | | Funct4 | | | | | ~ | | Rs2 | | | | | Rs1 | | Rd | | | | | 0Ch | {SHIFT} |
| r | | ~3 | | | | | | | Funct4 | | | | | Const5..0 | | | | | | | Rs1 | | Rd | | | | | 0Ch | {SHIFT} |
| r | | | Fn3 | | | | ~2 | | | | Fmt3 | | | | | Rs2 | | | | | Rs1 | | mop3 | | Cd2 | | | 0Dh | {SET} |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 0Eh | MULU |
| Fn3 | | | | | | Om3 | | | | Regno8 | | | | | | | | | | | Rs1 | | Rd | | | | | 0Fh | CSR |
| r | | Funct5 | | | | | | | | | | Fmt3 | | | | ~3 | | | Cs2 | | Rs1 | | Rd | | | | | 13h | {Reg2B} |
| r | | 1000h | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 17h | PERM |
| r | | 0 | | Bw6 | | | | | | | | | Bo6 | | | | | | | | Rs1 | | Rd | | | | | 1Ch | EXT |
| r | | 1 | | Bw6 | | | | | | | | | Bo6 | | | | | | | | Rs1 | | Rd | | | | | 1Ch | EXTU |
| r | 0 | | | Bw6 | | | | | | | | | Bo6 | | | | | | | | Rs1 | | Rd | | | | | 1Dh | DEP |
| r | 1 | | | Bw6 | | | | | | | | | Bo6 | | | | | | | | Rs1 | | Rd | | | | | 1Dh | FLIP |
| r | C | | | Bw6 | | | | | | | | | Bo6 | | | | | | | | C4..0 | | Rd | | | | | 1Eh | DEP |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 26h | SEQ |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 27h | SNE |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 28h | SLT |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 29h | SGE |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 2Ah | SLE |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 2Bh | SGT |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 2Ch | SLTU |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 2Dh | SGEU |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 2Eh | SLEU |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | mop3 | | Cd2 | | | 2Fh | SGTU |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | ~ | | Lk1 | 30h | JSR |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | ~ | | ~1 | 31h | JMP |
| ~ | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | ~4 | | | | Lk1 | 32h | JSR d[Rs1] |
| ~ | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | ~4 | | | | ~1 | 33h | JMP d[Rs1] |
| ~ | | Constant12..0 | | | | | | | | | | | | | | | | | | | 31 | | RO4 | | | | Lk1 | 34h | RTS |
| ~10 | | | | | | | | | | | | | | | Sema6 | | | | | | ~ | | RO4 | | | | ~ | 35h | RTE |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 36h | BEQ |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 37h | BNE |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 38h | BLT |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 39h | BGE |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 3Ah | BLE |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 3Bh | BGT |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 3Ch | BLTU |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 3Dh | BGEU |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 3Eh | BLEU |
| Target23..2 | | | | | | | | | | | | | | | | | | | | | | | | | Cs | | | 3Fh | BGTU |
| Constant38..16 | | | | | | | | | | | | | | | | | | | | | | | | | | | Rd | 48-4Fh | LMI |
| Constant63..41 | | | | | | | | | | | | | | | | | | | | | | | | | | | Rd | 50h-53h | LUI |
| Constant33..15 | | | | | | | | | | | | | | | | | | | | | | | Rd | | | | | 54h-57h | AMIPC |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 75h | GCSUB |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 78h | LLAL |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 79h | LLAH |
| r | | Funct5 | | | | | | | | | ~3 | | | | | Rs2 | | | | | Rs1 | | Rd | | | | | 7Ah | {OSR2} |
| r | | 2 | | | | | | | | | ~2 | | S | | | Rs2 | | | | | Rs1 | | DC3 | | | IC2 | | 7Ah | CACHE |
| ~ | | 8 | | | | | | | | | PL8 | | | | | | | | | | Rs1 | | ~2 | Tm3 | | | | 7Ah | REX |
| ~ | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | DC3 | | | IC2 | | 7Bh | CACHE |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 7Ch | LPAL |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 7Dh | LPAH |
| **Memory** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 80h | LDB |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 81h | LDBU |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 82h | LDW |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 83h | LDWU |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 84h | LDT |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 85h | LDTU |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 86h | LDO |
| r | | Constant12..0 | | | | | | | | | | | | | | | | | | | Rs1 | | Rd | | | | | 87h | LDOR |
| r | | ~2 | | | | S | | Rs3 | | | | | | | | | 05 | | | | Rs1 | | Rd | | | | | 8Fh | LDB |
| r | | ~2 | | | | S | | Rs3 | | | | | | | | | 15 | | | | Rs1 | | Rd | | | | | 8Fh | LDBU |
| r | | ~2 | | | | S | | Rs3 | | | | | | | | | 25 | | | | Rs1 | | Rd | | | | | 8Fh | LDW |
| r | | ~2 | | | | S | | Rs3 | | | | | | | | | 35 | | | | Rs1 | | Rd | | | | | 8Fh | LDWU |
| r | | ~2 | | | | S | | Rs3 | | | | | | | | | 45 | | | | Rs1 | | Rd | | | | | 8Fh | LDT |
| r | | ~2 | | | | S | | Rs3 | | | | | | | | | 55 | | | | Rs1 | | Rd | | | | | 8Fh | LDTU |
| r | | ~2 | | | | S | | Rs3 | | | | | | | | | 65 | | | | Rs1 | | Rd | | | | | 8Fh | LDO |
| r | | ~2 | | | | 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 |
| Const11..8 | | | | | | | | Rs3 | | | | | | | | | Rs2 | | | | Rs1 | | Const7..3 | | | | | AEh | STM |
| ~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 | {R3A} | {R2} | {R3B} | ADD | SUBF | MUL | CMP |
| 00001 | AND | OR | EOR | BIT | {SHIFT} | {SET} | MULU | CSR |
| 00010 | DIV | DIVU | DIVSU | {R2B} |  |  | MULSU | PERM |
| 00011 | REM | REMU | BYTNDX | WYDNDX | EXT | DEP | DEPI |  |
| 00100 | REMSU |  |  |  |  |  | SEQ | SNE |
| 00101 | SLT | SGE | SLE | SGT | SLTU | SGEU | SLEU | SGTU |
| **Branch Unit** | | | | | | | | |
| 00110 | JSR abs | JMP abs | JSR d[xn] | JMP d[xn] | RTS | RTE | BEQ | BNE |
| 00111 | BLT | BGE | BLE | BGT | BLTU | BGEU | BLEU | BGTU |
| 01000 | BVC | BVS | BOD |  | BPS |  |  |  |
|  | | | | | | | | |
| 01001 | LMI | | | | | | | |
| 01010 | LUI | | | | AMIPC | | | |
| 01011 |  |  |  |  |  |  |  |  |
| 01100 |  |  |  |  |  |  |  |  |
| 01101 |  |  |  |  |  |  |  |  |
| 01110 |  |  |  |  |  | GCSUB |  |  |
| 01111 | LLAL | LLAH | {OSR2} | CACHE | LPAL | LPAH |  |  |
| **Memory Unit** | | | | | | | | |
| 10000 | LDB | LDBU | LDW | LDWU | LDT | LDTU | LDO | LDOR |
| 10001 |  |  |  |  |  |  | FLDO | {LNDX} |
| 10010 |  |  |  |  |  |  | PLDO | LDM |
| 10011 |  |  |  |  |  |  |  |  |
| 10100 | STB | STW | STT | STO | STOC | SPTR |  |  |
| 10101 |  |  |  | FSTO |  |  |  | {SNDX} |
| 10110 |  |  |  | PSTO |  |  |  | STM |
| 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 |  |  |  |  |  |  |

## {R3A} Triadic Integer Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|  | MIN | MAX | MAJ | MUX | ADD | SUB |  | FLIP |

## {R3B} Triadic Integer Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|  | AND | OR | EOR | DEP | EXT | EXTU | BLEND | RGF |

## {R2} Dyadic Integer Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 00 | AND | OR | EOR | BMM | ADD | SUB | MUL | CMP |
| 01 | NAND | NOR | ENOR | BIT | {R1} | MOV | MULU | MULH |
| 10 | DIV | DIVU | DIVSU | REM | REMU | REMSU | MULSU | PERM |
| 11 | PTRDIF | DIF | BYTNDX | WYDNDX |  | MULSUH | MULUH | RGF |

## {R1} Monadic Integer Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 00 | CNTLZ | CNTLO | CNTPOP | COM | NOT | NEG |  |  |
| 01 |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |

## Fmt3 For Dyadic and MonadicOps

|  |  |
| --- | --- |
| Fmt3 | Size of Operation |
| 0 | octa |
| 1 | tetra |
| 2 | wyde |
| 3 | byte |

## {SET} Fn3 Set Integer Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|  | SEQ | SNE |  |  | SLT | SGE | SLTU | SGEU |

## {R2B} Dyadic Integer-Boolean Register Ops

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 00 | AND | OR | EOR |  | ADD | SUB | MUL | CMP |
| 01 | NAND | NOR | ENOR | BIT |  |  | MULU |  |
| 10 |  |  |  |  |  |  | MULSU |  |
| 11 |  | DIF |  |  |  |  |  |  |

## Floating-Point 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 |

## Floating-Point 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 |  |  |  |  |  |  | FFDP |
| 11 | CPYSGN | SGNINV | SGNAND | SGNOR | SGNXOR | SGNXNOR | FCLASS | FRGF |

## Posit Arithmetic Monadic Ops – {FLT1} Funct5

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 00 | PMOV | PRSQRTE | PTOI | ITOP |  |  | PSIGN | PMAN |
| 01 |  | PS2D | PS2Q | PD2Q | PSTAT | PSQRT | PISNAN | PFINITE |
| 10 | PTX | PCX | PEX | PDX |  | PTRUNC | PSYNC | PRES |
| 11 | PSIG | PD2S | PQ2S | PQ2D |  |  | FCLASS | PUNORD |

## Posit Arithmetic Dyadic Ops – {PST2} Funct5

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

## {OSR2} Funct6

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 000 | LLAL | LLAH | CACHE | GCSUB | LPAL | LPAH |  |  |
| 001 | PUSHQ | POPQ | PEEKQ |  |  |  |  |  |
| 010 | REX |  |  |  |  |  |  |  |
| 011 |  |  |  |  | MVMAP | MVSEG |  |  |
| 100 |  |  |  |  |  |  |  |  |
| 101 |  |  |  |  |  |  |  |  |
| 110 |  |  |  |  |  |  |  |  |
| 111 |  |  |  |  |  |  |  |  |

# ALU Operations

## Features

Compare results registers may be operands to some instructions such as shift and add operations. The least significant bit of the compare results register (the C flag) is used. It is only possible to shift a compare results register value to the left. (Shifting to the right would result in a zero value).

## Summary

Almost all ALU operations except for compare and bit have the capacity to update cr0 with results status. The compare and bit instruction may update any compare result register with status.

The following example shows the power of results merging operations and the utility of the set instructions and other operations.

**Example**: convert an ascii character to upper case

|  |
| --- |
| toUpper:  sge $cr0,$a0,#'a'  sle.and $cr0,$a0,#'z'  asl $t1,$cr0.C,#5  sub $a0,$a0,$t1  ret |

There are no branches in the code. The code is short enough that it may be implemented as a macro.

**Example:** extended precision addition:

|  |
| --- |
| add. $a2,$a0,$a1 ; add low, record carry  add $a5,$a3,a4 ; add high  add $a5,$a5,cr0.C ; add carry from low |

## 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. There is also a form of this instruction which sums the values of three registers (Rs1, Rs2, and Rs3).

The status result of the addition may optionally be copied to cr0.

**Formats Supported**: R2, R3, RI

**R2 Supported Formats**: .b .w, .t, .o

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | 45 | | Fmt3 | Rs2 | | Rs1 | Rd | 02h | ADD |
| r | 45 | | Fmt3 | ~3 | Bs2 | Rs1 | Rd | 13h | ADD |
| r | 43 | Rs3 | | Rs2 | | Rs1 | Rd | 01h | ADD |
| r | Constant12..0 | | | | | Rs1 | Rd | 04h | ADD |

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## AND[.] – Bitwise ‘And’

**Description**:

Bitwise ‘And’ 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. There is another form of this instruction which will bitwise and together three registers (Rs1, Rs2, Rs3).

The status result of the bitwise and may optionally be copied to cr0.

**Formats Supported**: R2, R2B, R3, RI

**R2 Supported Formats**: .b .w, .t, .o

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | 05 | | Fmt3 | Rs2 | | Rs1 | Rd | 02h | AND |
| r | 05 | | Fmt3 | ~3 | Bs2 | Rs1 | Rd | 13h | AND |
| r | 03 | Rs3 | | Rs2 | | Rs1 | Rd | 03h | AND |
| r | Constant12..0 | | | | | Rs1 | Rd | 08h | AND |

**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.

This shift may shift either a value in integer register Rs1 or a value in Boolean register Bs1.

**Formats Supported**: RR, RI6

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Fmt3 | 04 | 0 | Rs2 | Rs1 | | Rd | 0Ch | ASL |
| r | Fmt3 | 04 | 1 | Rs2 | ~3 | Cs1 | Rd | 0Ch | ASL |
| r | Fmt3 | 84 | Const5..0 | | Rs1 | | Rd | 0Ch | ASL |

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

**Example**: convert an ascii character to upper case

|  |
| --- |
| toUpper:  sge $cr0,$a0,#'a'  sle.and $cr0,$a0,#'Z'  asl $t1,$cr0,#5  sub $a0,$a0,$t1  ret |

There are no branches in the code. The code is short enough that it may be implemented as a macro.

## 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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Fmt3 | 44 | 0 | Rs2 | Rs1 | Rd | 0Ch | ASR |
| r | Fmt3 | 124 | Const5..0 | | Rs1 | Rd | 0Ch | ASR |

**Execution Units**: ALU

**Clock Cycles**: 0.5

**Exceptions**: none

## AMIPC – Add Middle Immediate to PC

**Description**:

Add an immediate value to the instruction pointer register and place the result into the destination register Rd. The immediate constant is composed of 13 bits of zeros on the right-hand side, 21 constant bits for bits 13 to 33, and bit 33 of the constant is sign extended to 64 bits. This instruction may be used to form instruction pointer relative addresses.

**Formats Supported**: LUI

|  |  |  |  |
| --- | --- | --- | --- |
| Constant33..15 | Rd | 54h-57h | AMIPC |

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

**Notes:**

*There is no facility to add to the PC higher order bits. Since this instruction is used mainly to generate PC relative addresses a 34-bit PC displacement was considered to be adequate for almost all cases.*

## BFCHG[.] – Bitfield Change

**Description**:

A bitfield in the source is inverted, the result is copied to the target register. There are two forms of this instruction, one uses registers to specify the offset and width, the other uses immediate constants supplied in the instruction to specify the offset and width.

**Instruction Format**: BFR

A bitfield in the source specified by Rs1 is inverted, the result is copied to the target register. Rs2 specifies the bit offset. Rs3 specifies the bit width.

**Instruction Format**: BFI

A bitfield in the source specified by Rs1 is inverted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | 73 | | Rs3 | | Rs2 | Rs1 | Rd | 01h | FLIP |
| r | 1 | Bw6 | | Bo6 | | Rs1 | Rd | 1Dh | FLIP |

**Clock Cycles**: 1

**Execution Units:** Integer ALU #0 only

**Exceptions**: none

## BFCLR[.] – Bitfield Clear

**Description**:

This is an alternate mnemonic for the [DEP](#_DEP_–_Bitfield) instruction where the source register is assumed to be x0. A bitfield in the source is cleared, the result is copied to the target register. There are two forms of this instruction, one uses registers to specify the offset and width, the other uses immediate constants supplied in the instruction to specify the offset and width.

**Instruction Format**: R3

A bitfield in the source specified by Rs1 is cleared, the result is copied to the target register. Rs2 specifies the bit offset. Rs3 specifies the bit width.

**Instruction Format**: BFI

A bitfield in the source specified by Rs1 is cleared, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 0 | Bw6 | Bo6 | 0 | Rd | 1Dh | DEP |

**Clock Cycles**: 0.25

**Execution Units:** Integer ALU #0 only

**Exceptions**: none

**Notes**:

Normally Rs3 is a register which is the same as the target register Rd.

## 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**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

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**: Integer ALU

**Clock Cycles**: 1

**Exceptions**: none

## BLEND[.] – Blend Colors

**Description**:

This instruction blends two colors whose values are in Rs1 and Rs2 according to an alpha value in Rs3. The resulting color is placed in register Rd. The alpha value is an eight-bit value assumed to be a binary fraction less than one. The color values in Rs1 and Rs2 are assumed to be RGB888 format colors. The result is a RGB888 format color. The high order eight bits of the result register are set to the high order eight bits of Rs1. Note that a close approximation to 1.0 – alpha is used.

**Instruction Format**: R3

**Operation**: Rd = (Rs1 \* alpha) + (Rs2 \* ~alpha)

**Clock Cycles**: 1

## 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.

## BYTNDX[.] – Byte Index

**Description:**

This instruction searches Rs1 for a byte specified by Rs2 or an immediate value and places the index of the byte into the target register Rd. If the byte is not found -1 is placed in the target register. A common use would be to search for a null byte. The index result may vary from -1 to +7. The index of the first found byte is returned (closest to zero).

**Instruction Format:** R2

**R2 Supported Formats**: .b .w, .t, .o

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rd = Index of (Rs2 in Rs1)

**Exceptions:** none

## 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**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: Integer ALU

**Clock Cycles**: 1

## CNTLO[.] – Count Leading Ones

**Description**:

Count the number of leading ones (starting at the MSB) in Rs1 and place the count in the target register.

**Instruction Format**: R1

**R1 Supported Formats**: .b .w, .t, .o

**Clock Cycles**: 1

**Execution Units:** Integer ALU

**Exceptions**: none

## CNTLZ[.] – Count Leading Zeros

**Description**:

Count the number of leading zeros (starting at the MSB) in Rs1 and place the count in the target register.

**Instruction Format**: R1

**R1 Supported Formats**: .b .w, .t, .o

**Clock Cycles**: 1

**Execution Units:** Integer ALU

**Exceptions**: none

## CNTPOP[.] – Count Population

**Description**:

Count the number of one bits in Rs1 and place the count in the target register.

**Instruction Format**: R1

**R1 Supported Formats**: .b .w, .t, .o

**Clock Cycles**: 1

**Execution Units:** Integer ALU

**Exceptions**: none

## COM[.] – One’s Complement

**Description:**

This instruction takes the one’s complement of a register and places the result in a target register. This is almost the same operation as exclusive or’ing with minus one, however the operation size may be set to operate on only a byte, wyde, tetra or octa value.

**Instruction Format:** Integer R1

**R1 Supported Formats**: .b .w, .t, .o

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rd = ~Rs1

**Exceptions:** none

## CSR – Control and Status Access

**Description**:

The CSR instruction group provides access to control and status registers in the core. For the read-write operation the current value of the CSR is placed in the target register Rd then the CSR is updated from register Rs1. The CSR read / update operation is an atomic operation.

**Instruction Format**: CSR

|  |  |  |
| --- | --- | --- |
| Op3 |  | Operation |
| 0 | CSRRD | Only read the CSR, no update takes place, Rs1 should be R0. |
| 1 | CSRRW | Both read and write the CSR |
| 2 | CSRRS | Read CSR then set CSR bits |
| 3 | CSRRC | Read CSR then clear CSR bits |
| 4 |  | reserved |
| 5 | CSRRWI | Read and Write CSR with immediate |
| 6 | CSRRSI | Read and set using immediate |
| 7 | CSRRCI | Read and clear using immediate |

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

The OM3 field is reserved to specify the operating mode. Note that registers cannot be accessed by a lower operating mode.

|  |  |  |  |
| --- | --- | --- | --- |
| Regno8 |  | Access | Description |
| 01 | HARTID | R | hardware thread identifier (core number) |
| 02 | TICK | R | tick count, counts every cycle from reset |
| 30-37 | TVEC | RW | trap vector handler address |
| 48 | EPC | RW | exceptioned pc, pc value at point of exception |
| 44 | STATUSL | RWSC | status register, contains interrupt mask, operating level |
| 45 | STATUSH | RW | status register bits 64 to 127 |
| 80-BF | CODE | RW | code buffers |
| F0 | INFO | R | Manufacturer name |
| F1 | “ | R | “ |
| F2 | “ | R | cpu class |
| F3 | “ | R | “ |
| F4 | “ | R | cpu name |
| F5 | “ | R | “ |
| F6 | “ | R | model number |
| F7 | “ | R | serial number |
| F8 | “ | R | cache sizes instruction (bits 32 to 63), data (bits 0 to 31) |

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

**Clock Cycles**: 1

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

## DEP[.] – Bitfield Deposit

**Description**:

The target register Rd is used as the source data. A bitfield whose value is contained in Rs1 is inserted into the source data by copying low order bits from Rs1 shifted to the left. The result is placed in the target register Rd. There are two forms of this instruction, one uses registers to specify the offset and width, the other uses immediate constants supplied in the instruction to specify the offset and width.

**Instruction Format**: R3, BFI

**Instruction Format**: BFI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 0 | Bw6 | Bo6 | Rs1 | Rd | 1Dh | DEP |

**Clock Cycles**: 1

**Execution Units: Integer** ALU #0 Only

**Exceptions**: none

## DEPI[.] – Bitfield Deposit Immediate

**Description**:

The target register Rd is used as the source data. A bitfield whose value is a constant specified in the Rs1 field of the instruction is inserted into the source data by copying low order bits from the constant shifted to the left. The bitfield may not be wider than six bits. Use multiple instructions to achieve a wider field width, or load a register with the value first then use the registered form of the instruction. The result is placed in the target register Rd.

This instruction may be used to clear or set a bitfield.

**Instruction Format**: DEPI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | C | Bw6 | Bo6 | C | Rd | 1Eh | DEP |

**Clock Cycles**: 1

**Execution Units: Integer** ALU #0 Only

**Exceptions**: none

## DIF[.] – Difference

**Description:**

This instruction computes the difference between two signed values in registers Rs1 and Rs2 and places the result in a target Rd register. The difference is calculated as the absolute value of Rs1 minus Rs2.

**Instruction Format:** R2

**Supported Formats**: .b .w, .t, .o

**Clock Cycles:** 1

**Execution Units:** Integer

**Operation:**

Rd = Abs(Rs1 - Rs2)

**Exceptions**: none

## 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**: R2, 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**: R2, RI

**Execution Units**: ALU

**Clock Cycles**: 67

**Exceptions**: none

## ENOR[.] – Bitwise Exclusive Nor

**Description**:

Perform a bitwise exclusive or operation between two operands then invert the result. Operands must be in registers.

**Instruction Format**: R2

**R2 Supported Formats**: .b .w, .t, .o

**Clock Cycles**: 1

**Execution Units: Integer ALU**

**Scalar Operation**

Rd = ~(Rs1 ^ Rs2)

## EOR[.] – Bitwise Exclusive ‘Or’

**Description**:

Bitwise exclusive ‘Or’ 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. The immediate value is zero extended to the left from bit 14 to the machine width.

The status result of the exclusive or may optionally be copied to cr0.

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: Integer ALU

**Clock Cycles**: 1

**Exceptions**: none

## EXT[.] –Extract Bitfield

**Description**:

A bitfield is extracted from the source by shifting the source to the right and ‘and’ masking. The result is sign extended to the width of the machine. This instruction may be used to sign extend a value from an arbitrary bit position. There are two forms of this instruction, one uses registers to specify the offset and width, the other uses immediate constants supplied in the instruction to specify the offset and width.

**Instruction Format**: BFR

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Rs2 specifies the bit offset. Rs3 specifies the bit width.

**Instruction Format**: BFI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 0 | Bw6 | Bo6 | Rs1 | Rd | 1Ch | EXT |

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

**Clock Cycles**: 0.25

**Execution Units:** Integer ALU #0 Only

**Exceptions**: none

**Notes**:

## EXTU[.] –Extract Unsigned Bitfield

**Description**:

A bitfield is extracted from the source by shifting the source to the right and ‘and’ masking. The result is zero extended to the width of the machine. This instruction may be used to zero extend a value from an arbitrary bit position. There are two forms of this instruction, one uses registers to specify the offset and width, the other uses immediate constants supplied in the instruction to specify the offset and width.

**Instruction Format**: BFR

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Rs2 specifies the bit offset. Rs3 specifies the bit width.

**Instruction Format**: BFI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 1 | Bw6 | Bo6 | Rs1 | Rd | 1Ch | EXTU |

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

**Clock Cycles**: 0.25

**Execution Units:** Integer ALU #0 Only

**Exceptions**: none

**Notes**:

## FLIP[.] – Flip Bits

**Description**:

A bitfield in the destination is bitwise exclusively or’d, with a source value in Rs1. The result is copied bask to the destination register. There are two forms of this instruction, one uses registers to specify the offset and width, the other uses immediate constants supplied in the instruction to specify the offset and width.

**Instruction Format**: BFR

A bitfield in the source specified by Rs1 is exclusive or’d with the target, the result is copied to the target register. Rs2 specifies the bit offset. Rs3 specifies the bit width.

**Instruction Format**: BFI

A bitfield in the source specified by Rs1 is exclusive or’d with the target, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | 73 | | Rs3 | | Rs2 | Rs1 | Rd | 01h | FLIP |
| r | 1 | Bw6 | | Bo6 | | Rs1 | Rd | 1Dh | FLIP |

**Clock Cycles**: 1

**Execution Units:** Integer ALU #0 only

**Exceptions**: none

## LMI – Load Middle Immediate

**Description**:

Load an immediate value into the destination register Rd. The immediate constant is composed of 13 bits of zeros on the right-hand side, 26 constant bits for bits 13 to 38, and bit 38 of the constant is sign extended to 64 bits. The destination register must be either x1 or x2.

**Formats Supported**: LUI

|  |  |  |  |
| --- | --- | --- | --- |
| Constant38..16 | Rd | 48h-4Fh | LMI |

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

**Notes:**

## 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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Fmt3 | 14 | ~ | Rs2 | Rs1 | Rd | 0Ch | LSR |
| r | Fmt3 | 94 | 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 destination register Rd bits 39 to 63. Bits 0 to 38 of the destination register are not affected. The destination register must be either x1 or x2.

**Formats Supported**: LUI

|  |  |  |  |
| --- | --- | --- | --- |
| Constant63..41 | Rd | 50h-53h | LUI |

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

**Notes:**

## 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 register to another register. This instruction may be used to move between different types of registers. The source and destination registers are identified by seven-bit tags. The Rd and Rs1 fields of the instructions are extended by d2 and s2 fields to seven bits.

|  |  |  |
| --- | --- | --- |
| Tag | Associated Register |  |
| 0 to 31 | GP0 to GP31 | general purpose registers |
| 32 to 63 | FP0 to FP31 | floating point registers |
| 64 to 95 | PS0 to PS31 | posit arithmetic registers |
| 96 to 97 | RA0, RA1 | return address registers |
| 98 to 102 | reserved |  |
| 103 | EIP | exception instruction pointer |
| 104 to 111 | reserved |  |
| 112 to 115 | CR0 to CR3 | compare result registers |
| 116 to 123 | reserved |  |
| 124 | reserved |  |
| 125 | CR <all> | all compare results registers |
| 126 | reserved | not used |
| 127 | none | instruction without a target |

**Formats Supported**: R2

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | 135 | Fmt3 | ~ | s2 | d2 | Rs1 | Rd | 02h | MOV |

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: ALU

**Clock Cycles**: 1

**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

## NEG[.] – Negate

**Description:**

This instruction negates the value in register Rs1 and places the result in target register Rd.

**Instruction Format: R1**

**Supported Formats**: .b .w, .t, .o

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rd = -Rs1

**Exceptions**: none

**Notes**:

## NOR[.] – Bitwise Nor

**Description**:

Perform a bitwise or operation between two operands then invert the result. Both operands must be in registers.

**Instruction Format**: R2

**Supported Formats**: .b .w, .t, .o

**Clock Cycles**: 1

**Execution Units: Integer ALU**

**Exceptions**: none

## NOT[.] – Logical Not

**Description:**

This instruction takes the logical ‘not’ value of a register and places the result in a target register. If the source register contains a non-zero value, then a zero is loaded into the target. Otherwise if the source register contains a zero a one is loaded into the target register.

**Instruction Format: R1**

**R1 Supported Formats**: .b .w, .t, .o

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rd = !Rs1

**Exceptions**: none

**Notes**:

## 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 13 to the machine width.

The status result of the inclusive or may optionally be copied to cr0.

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: Integer 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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | 1000h13 | | | | | Rs1 | | Rd | | | 17h | | PERM |
| ~3 | | S7 | S6 | S5 | S4 | | S3 | | S2 | S1 | 85 | S0 | NOP |

**Formats Supported**: R2

**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.

The first operand for the instruction may also be a Boolean value register.

**Formats Supported**: RR, RI6

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Fmt3 | 24 | 0 | Rs2 | Rs1 | | Rd | 0Ch | ROL |
| r | Fmt3 | 24 | 1 | Rs2 | ~3 | Bs1 | Rd | 0Ch | ROL |
| r | Fmt3 | 104 | 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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Fmt3 | 34 | ~ | Rs2 | Rs1 | Rd | 0Ch | ROR |
| r | Fmt3 | 114 | Const5..0 | | Rs1 | Rd | 0Ch | ROR |

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SEQ – Set if Equal

**Description**:

Test two operand values for equality and place the result in the target 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 immediate value is sign extended to the left from bit 13 to the machine width. A branch instruction may make use of the comparison result.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if operands are equal, otherwise 0 |
| 1 | Z | 1 if operands are equal, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if operands are equal, otherwise 0 |
| 5 | O | 1 if operands are equal, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SGE – Set if Greater Than or Equal

**Description**:

Test if the first operand is greater than or equal to the second one and place the result in the target 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 immediate value is sign extended to the left from bit 13 to the machine width. Both operands are treated as signed values. A branch instruction may make use of the comparison result. The R2 register form of this instruction may also be used to determine less than or equal ([SLE](#_SLE[.]_–_Set)) by swapping the operands.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if first operand greater than or equal to second, otherwise 0 |
| 1 | Z | 1 if first operand greater than or equal to second, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if first operand greater than or equal to second, otherwise 0 |
| 5 | O | 1 if first operand greater than or equal to second, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SGEU – Set if Greater Than or Equal Unsigned

**Description**:

Test if the first operand is greater than or equal to the second one and place the result in the target 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 immediate value is sign extended to the left from bit 13 to the machine width. Both operands are treated as unsigned values. A branch instruction may make use of the comparison result. The R2 register form of this instruction may also be used to determine less than or equal ([SLEU](#_SLEU_–_Set)) by swapping the operands.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if first operand greater than or equal to second, otherwise 0 |
| 1 | Z | 1 if first operand greater than or equal to second, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if first operand greater than or equal to second, otherwise 0 |
| 5 | O | 1 if first operand greater than or equal to second, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SGT – Set if Greater Than

**Description**:

Test if the first operand is greater than or equal to the second one and place the result in the target compare results register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand must be an immediate value. The immediate value is sign extended to the left from bit 13 to the machine width. Both operands are treated as signed values. A branch instruction may make use of the comparison result. There is no R2 register form of this instruction. The [SLT](#_SLT[.]_–_Set) instruction may be used to detect greater than by swapping operands.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if first operand greater than to second, otherwise 0 |
| 1 | Z | 1 if first operand greater than to second, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if first operand greater than to second, otherwise 0 |
| 5 | O | 1 if first operand greater than to second, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SGTU – Set if Greater Than Unsigned

**Description**:

Test if the first operand is greater than or equal to the second one and place the result in the target compare results register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand must be an immediate value. The immediate value is sign extended to the left from bit 13 to the machine width. Both operands are treated as unsigned values. A branch instruction may make use of the comparison result. There is no R2 register form of this instruction. The [SLTU](#_SLTU_–_Set) instruction may be used to detect greater than by swapping operands.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if first operand greater than to second, otherwise 0 |
| 1 | Z | 1 if first operand greater than to second, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if first operand greater than to second, otherwise 0 |
| 5 | O | 1 if first operand greater than to second, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SLE – Set if Less Than or Equal

**Description**:

Test if the first operand is less than or equal to the second one and place the result in the target compare results register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand must be an immediate value. The immediate value is sign extended to the left from bit 13 to the machine width. Both operands are treated as signed values. A branch instruction may make use of the comparison result. There is no R2 register form of this instruction. Instead the [SGE](#_SGE[.]_–_Set) instruction may be used with the operands swapped.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if first operand less than second, otherwise 0 |
| 1 | Z | 1 if first operand less than second, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if first operand less than second, otherwise 0 |
| 5 | O | 1 if first operand less than second, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: RI

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SLEU – Set if Less Than or Equal Unsigned

**Description**:

Test if the first operand is less than or equal to the second one and place the result in the target compare results register. The first operand must be in a register specified by the Rs1 field of the instruction. The second operand must be an immediate value. The immediate value is sign extended to the left from bit 13 to the machine width. Both operands are treated as signed values. A branch instruction may make use of the comparison result. There is no R2 register form of this instruction. Instead the [SGEU](#_SGEU[.]_–_Set) instruction may be used with the operands swapped.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if first operand less than second, otherwise 0 |
| 1 | Z | 1 if first operand less than second, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if first operand less than second, otherwise 0 |
| 5 | O | 1 if first operand less than second, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: RI

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SLT – Set if Less Than

**Description**:

Test if the first operand is less than the second one and place the result in the target 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 immediate value is sign extended to the left from bit 13 to the machine width. Both operands are treated as signed values. A branch instruction may make use of the comparison result.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if first operand less than second, otherwise 0 |
| 1 | Z | 1 if first operand less than second, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if first operand less than second, otherwise 0 |
| 5 | O | 1 if first operand less than second, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SLTU – Set if Less Than Unsigned

**Description**:

Test if the first operand is less than the second one and place the result in the target 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 immediate value is sign extended to the left from bit 13 to the machine width. Both operands are treated as unsigned values. A branch instruction may make use of the comparison result.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if first operand less than second, otherwise 0 |
| 1 | Z | 1 if first operand less than second, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if first operand less than second, otherwise 0 |
| 5 | O | 1 if first operand less than second, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: ALU

**Clock Cycles**: 1

**Exceptions**: none

## SNE – Set if Not Equal

**Description**:

Test two operand values for inequality and place the result in the target 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 immediate value is sign extended to the left from bit 13 to the machine width. A branch instruction may make use of the comparison result.

This instruction will set the compare results register as follows:

|  |  |  |
| --- | --- | --- |
| Bit | Flag | Value Set |
| 0 | C | 1 if operands are not equal, otherwise 0 |
| 1 | Z | 1 if operands are not equal, otherwise 0 |
| 2 | ~ | 0 |
| 3 | ~ | 0 |
| 4 | P | 1 if operands are not equal, otherwise 0 |
| 5 | O | 1 if operands are not equal, otherwise 0 |
| 6 | V | 0 |
| 7 | N | 0 |

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**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.

The status result of the subtraction may optionally be copied to cr0.

**Formats Supported**: R2

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: ALU

**Clock Cycles**: 1

**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.

The status result of the subtraction may optionally be copied to cr0.

**Formats Supported**: RI

**Execution Units**: ALU

**Clock Cycles** 1

**Exceptions**: none

## SXB[.] –Sign Extend Byte

**Description**:

This is an alternate mnemonic for the bitfield extract (EXT) operation.

**Instruction Format**: BFI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 0 | 86 | 06 | Rs1 | Rd | 1Ch | EXT |

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

**Clock Cycles**: 1

**Execution Units:** Integer ALU #0 Only

**Exceptions**: none

**Notes**:

## SXW[.] –Sign Extend Wyde

**Description**:

This is an alternate mnemonic for the bitfield extract (EXT) operation.

**Instruction Format**: BFI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 0 | 166 | 06 | Rs1 | Rd | 1Ch | EXT |

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

**Clock Cycles**: 1

**Execution Units:** Integer ALU #0 Only

**Exceptions**: none

**Notes**:

## SXT[.] –Sign Extend Tetra

**Description**:

This is an alternate mnemonic for the bitfield extract (EXT) operation.

**Instruction Format**: BFI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 0 | 326 | 06 | Rs1 | Rd | 1Ch | EXT |

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

**Clock Cycles**: 1

**Execution Units:** Integer ALU #0 Only

**Exceptions**: none

**Notes**:

## WYDNDX[.] – Wyde Index

**Description:**

This instruction searches Rs1, which is treated as an array of four wydes, for a wyde value specified by Rs2 or an immediate value and places the index of the wyde into the target register Rd. If the wyde is not found -1 is placed in the target register. A common use would be to search for a null wyde. The index result may vary from -1 to +3. The index of the first found wyde is returned (closest to zero).

**Instruction Format:** R2

**R2 Supported Formats**: .b .w, .t, .o

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Operation:**

Rd = Index of (Rs2 in Rs1)

**Exceptions:** none

## XOR[.] – Bitwise Exclusive ‘Or’

**Description**:

This is an alternate mnemonic for the [EOR](#_EOR_–_Bitwise) function. Bitwise exclusive ‘Or’ 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. The immediate value is zero extended to the left from bit 14 to the machine width.

The status result of the exclusive or may optionally be copied to cr0.

**Formats Supported**: R2, RI

**R2 Supported Formats**: .b .w, .t, .o

**Execution Units**: Integer ALU

**Clock Cycles**: 1

**Exceptions**: none

## ZXB[.] –Zero Extend Byte

**Description**:

This is an alternate mnemonic for the bitfield extract (EXTU) operation.

**Instruction Format**: BFI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 1 | 86 | 06 | Rs1 | Rd | 1Ch | EXTU |

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

**Clock Cycles**: 1

**Execution Units:** Integer ALU #0 Only

**Exceptions**: none

**Notes**:

## ZXW[.] –Zero Extend Wyde

**Description**:

This is an alternate mnemonic for the bitfield extract (EXTU) operation.

**Instruction Format**: BFI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 1 | 166 | 06 | Rs1 | Rd | 1Ch | EXT |

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

**Clock Cycles**: 1

**Execution Units:** Integer ALU #0 Only

**Exceptions**: none

**Notes**:

## ZXT[.] –Zero Extend Tetra

**Description**:

This is an alternate mnemonic for the bitfield extract (EXTU) operation.

**Instruction Format**: BFI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| r | 1 | 326 | 06 | Rs1 | Rd | 1Ch | EXTU |

A bitfield in the source specified by Rs1 is extracted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. Bo and Bw are constants supplied in the instruction.

**Clock Cycles**: 1

**Execution Units:** Integer ALU #0 Only

**Exceptions**: none

**Notes**:

# Memory Operations

## FLDO[.] – Float Load Octa (64 bits)

**Description**:

Data is loaded into Frd 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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Frd | 8Eh | FLDO |
| r | | ~2 | S | Rs3 | 145 | Rs1 | Frd | 8Fh | FLDO |

**Operation:**

Rt = Memory32[d+Ra]

or

Rt = Memory32[Ra+Rb]

**Execution Units**: Mem

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

**Exceptions**: none

## FSTO – Float Store Octet (64 bits)

**Description**:

Data from Frs2 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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~ | Constant12..5 | | | Frs2 | Rs1 | Const4..0 | ABh | FSTO |
| ~3 | | S | Rs3 | Frs2 | Rs1 | 115 | AFh | FSTO |

**Flags Affected**: none

**Operation:**

Memory64[d+Rs1] = Frs2

or

Memory64[d+Rs1+Rs3\*Sc] = Frs2

**Execution Units**: Mem

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

**Exceptions**: none

## 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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Rd | 80h | LDB |
| r | | ~2 | S | Rs3 | 05 | 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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Rd | 81h | LDBU |
| r | | ~2 | S | Rs3 | 15 | 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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Rd | 86h | LDO |
| r | | ~2 | S | Rs3 | 65 | 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

## LDOR[.] – Load Octa (64 bits) and Reserve

**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. Additionally, a reservation is placed on the load address.

**Formats Supported**: RR,RI

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Rd | 87h | LDOR |
| r | | ~2 | S | Rs3 | 75 | Rs1 | Rd | 8Fh | LDOR |

**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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Rd | 84h | LDT |
| r | | ~2 | S | Rs3 | 45 | 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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Rd | 85h | LDTU |
| r | | ~2 | S | Rs3 | 55 | 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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Rd | 82h | LDW |
| r | | ~2 | S | Rs3 | 25 | 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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Rd | 83h | LDWU |
| r | | ~2 | S | Rs3 | 35 | 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

## PLDO[.] – Posit Load Octa (64 bits)

**Description**:

Data is loaded into posit register Prd 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

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| r | Constant12..0 | | | | | Rs1 | Prd | 96h | LDO |
| r | | ~2 | S | Rs3 | 225 | Rs1 | Prd | 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

## 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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~ | Constant12..5 | | | Rs2 | Rs1 | Const4..0 | A0h | STB |
| ~3 | | S | Rs3 | Rs2 | Rs1 | 05 | AFh | STB |

**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

## STM – Store Multiple Registers

**Description**:

This instruction stores multiple registers to memory at the address which is the sum of Rs1 and an immediate constant, beginning with the register specified in Rs2 and continuing upwards for the immediate count specified in the Rs3 field of the instruction.

**Instruction Formats**: LM

**Clock Cycles**: 4 minimum depending on memory access time

## 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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~ | Constant12..5 | | | Rs2 | Rs1 | Const4..0 | A3h | STO |
| ~3 | | S | Rs3 | Rs2 | Rs1 | 35 | AFh | STO |

**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

## STOC – Store Octet (64 bits) and Clear Reservation

**Description**:

Conditionally store data from Rs2 to memory if an address reservation if present. If no reservation is present the Z bit of cr0 will be cleared and the store will not be done. Otherwise the Z bit of cr0 will be set. The address 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. Additionally, a reservation set on the address is cleared.

**Formats Supported**: STR, STI

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~ | Constant12..5 | | | Rs2 | Rs1 | Const4..0 | A4h | STO |
| ~3 | | S | Rs3 | Rs2 | Rs1 | 45 | AFh | STO |

**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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~ | Constant12..5 | | | Rs2 | Rs1 | Const4..0 | A2h | STT |
| ~3 | | S | Rs3 | Rs2 | Rs1 | 25 | AFh | STT |

**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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~ | Constant12..5 | | | Rs2 | Rs1 | Const4..0 | A1h | STW |
| ~3 | | S | Rs3 | Rs2 | Rs1 | 15 | AFh | STW |

**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

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ~ | Constant12..5 | | | Rs2 | Rs1 | Const4..0 | A5h | STO |
| ~3 | | S | Rs3 | Rs2 | Rs1 | 55 | AFh | STO |

**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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer. The remaining bits of the instruction pointer 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 instruction pointer is reset to the contents of tvec[5] 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 EPC register.

**Formats Supported**: BRK

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

**Operation:**

PMSTACK = (PMSTACK << 4) | 10

RSSTACK = (RSSTACK << 5) | 31

CAUSE = Const8

EPC = PC

PC = tvec[5]

**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**:

## REX – Redirect Exception

**Description**:

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

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

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

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

**Instruction Format**: REX

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ~ | 8 | PL8 | Rs1 | ~2 | Tm3 | 7Ah | REX |

|  |  |
| --- | --- |
| Tm3 |  |
| 0 | redirect to user mode |
| 1 | redirect to supervisor mode |
| 2 | redirect to hypervisor mode |
| 3 | redirect to machine mode |
| 4 | redirect to interrupt mode |
| 5 to 7 | not used |

**Clock Cycles**: 4

**Execution Units: Branch**

Example:

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

**Notes**:

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

## RTD – Return from Debug Mode

**Description**:

This instruction is an alternate mnemonic for the [RTE](#_RTE_–_Return) instruction. 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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ~10 | Sema6 | ~ | RO4 | ~ | 35h | RTE |

**Flags Affected**: none

**Operation:**

PMSTACK = PMSTACK >> 4

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = EPC

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## RTE – Return from Exception

**Description**:

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

This instruction may be encoded to return a short distance past the exception address point. This may be useful to return to the next instruction or return to a point past inline parameters. The RO4 field specifies a return offset in terms of instruction words.

There is really only a single instruction to return from any mode for an exception. Although there are several additional mnemonics. The processor “knows” which exception address to return to according to the register set currently in use.

**Formats Supported**: RTE

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ~10 | Sema6 | ~ | RO4 | ~ | 35h | RTE |

The constant field is not used.

**Flags Affected**: none

**Operation:**

PMSTACK = PMSTACK >> 4

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = EPC

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## RTH – Return from Hypervisor Mode Subroutine

**Description**:

This instruction is an alternate mnemonic of the [RTE](#_RTE_–_Return) instruction.

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**: RTE

**Flags Affected**: none

**Operation:**

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes:**

## RTI – Return from Interrupt Subroutine

**Description**:

This instruction is an alternate mnemonic for the [RTE](#_RTE_–_Return) instruction. Restore the previous interrupt enable setting, register set and operating level and transfer program execution back to the address in the exception address register (EPC). One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RTE

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ~10 | Sema6 | ~ | RO4 | ~ | 35h | RTE |

The constant field is not used.

**Flags Affected**: none

**Operation:**

PMSTACK = PMSTACK >> 4

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = EPC

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## RTM – Return from Machine Mode Subroutine

**Description**:

This instruction is an alternate mnemonic for the [RTE](#_RTE_–_Return) instruction. Restore the previous interrupt enable setting, register set and operating level and transfer program execution back to the address in the exception address register (EPC). One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RTE

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ~10 | Sema6 | ~ | RO4 | ~ | 35h | RTE |

The constant field is not used.

**Flags Affected**: none

**Operation:**

PMSTACK = PMSTACK >> 4

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = EPC

**Execution Units**: Branch

**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 (RO4) 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 RO4 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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ~ | Constant12..0 | 31 | RO4 | Lk1 | 34h | RTS |

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

The RO4 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 RO4 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 RO4 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 + RO4 \* 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**:

## RTU – Return from User Mode Exception

**Description**:

This instruction is an alternate mnemonic for the [RTE](#_RTE_–_Return) instruction. Restore the previous interrupt enable setting, register set and operating level and transfer program execution back to the address in the exception address register (EPC). One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RTE

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ~10 | Sema6 | ~ | RO4 | ~ | 35h | RTE |

The constant field is not used.

**Flags Affected**: none

**Operation:**

PMSTACK = PMSTACK >> 4

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = EPC

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

## SRET – Return from Supervisor Mode Subroutine

**Description**:

This instruction is an alternate mnemonic for the [RTE](#_RTE_–_Return) instruction. Restore the previous interrupt enable setting, register set and operating level and transfer program execution back to the address in the exception address register (EPC). One of sixty-four semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

**Formats Supported**: RTE

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ~10 | Sema6 | ~ | RO4 | ~ | 35h | RTE |

The constant field is not used.

**Flags Affected**: none

**Operation:**

PMSTACK = PMSTACK >> 4

RSSTACK = RSSTACK >> 5

Semaphore[0] = 0

Semaphore[Sema6] = 0

PC = EPC

**Execution Units**: Branch

**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

PMSTACK = (PMSTACK << 4) | 8

EPC = PC + 4;

PC = $FFFFFFFFE0000

Else

PC = PC (clock stopped)

**Execution Units**: Fetch stage

**Clock Cycles**:

**Exceptions**: none

**Notes**:

# Posit Arithmetic Instructions

## PABS – Posit Absolute Value

**Description:**

Take the absolute value of a posit number in register Prs1 and places the result into target register Prd. No rounding of the number occurs.

**Instruction Format: PST1**

**Clock Cycles: 1**

**Execution Units:** Posit Arithmetic

## PADD – Posit addition

**Description:**

Add two posit numbers in registers Prs1 and Prs2 or a short immediate value and place the result into target register Prd. The result is rounded.

**Instruction Format: PST2**

**Clock Cycles: 6**

**Execution Units:** Posit Arithmetic

## PCMP - Posit Compare

**Description:**

The register compare instruction compares two registers as posit 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: PST2**

**Clock Cycles:** 0.5

**Execution Units:** Posit Arithmetic

**Operation:**

if Prs1 < Prs2

Cr.N = 1

else if Prs1 = Prs2

Cr.Z = 1

else

Cr.N = 0

Cr.Z = 0

if unordered (Prs1, Prs2)

Cr.V = 1

else

Cr.V = 0

## PDIV – Posit Divide

**Description:**

Divide two posit numbers in registers Prs1 and Prs2 and place the result into target register Prd.

**Instruction Format: PST2**

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

**Execution Units:** Posit Arithmetic

## PMUL – Posit Multiplication

**Description:**

Multiply two posit numbers in registers Prs1 and Prs2 and place the result into target register Prd.

**Instruction Format: PST2**

**Clock Cycles: 7**

**Execution Units:** Posit Arithmetic

## PSUB – Posit Subtraction

**Description:**

Subtract two posit numbers in registers Prs1 and Prs2 and place the result into target register Prd.

**Instruction Format: PST2**

**Clock Cycles: 6**

**Execution Units:** Posit Arithmetic

# 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

# Operating Systems Support

## 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:

## GCCLR – Garbage Collect Clear Memory

**Description**:

The GCCLR instruction returns the current status of card memory associated with garbage collection identified by Rs1 in Rd then clears the memory in preparation for a subsequent garbage collection pass. The garbage collector can tell very quickly whether a store pointer has occurred in a memory region. It needs only examine two bytes at the innermost layer. The GCCLR instruction returns the status 64 bits at a time.

**Rs1 Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| 61 32 | 31 | 30 28 | 27 0 |
| ~ | P | Layer | Address27..0 |

The P bit causes the instruction to preserve the current contents of the card and allows reading the memory without clearing it.

|  |  |  |
| --- | --- | --- |
| Layer | Size | Address field (64-bit chunks) |
| 0 | 2 bytes | 0 |
| 1 | 256 bytes | 0 to 31 |
| 2 | 256kB | 0 to 32767 |
|  |  |  |

**Instruction Format**: GCCLR

**Exceptions**: none

## GCSUB – Garbage Collect Subtract

**Description**:

Subtract Rs2 or an immediate value from Rs1 and place the result in the destination register Rd. Also clear the garbage collect interrupt enable bit in the user interrupt enable CSR (CSR $004) and load a lockout count into an internal instruction count register. Once the lockout count has expired the interrupt enable bit will be set enabling GC interrupts. The value loaded into the lockout count is four plus the value in Rs2 or the immediate value shift right twice.

**Instruction Format**: R2, RI

**Exceptions:** none

## MVMAP – Move Mapping Register

**Description**:

MVMAP instruction is used for mapping memory pages into the address space of a task.

MVMAP works in a manner like the CSR instruction, but is applied for mapping register access only. Register Rs2 indirectly identifies the map register to access. Note that Rs2 is an integer register that contains the map register number. Rs1 identifies new source data for the map register, and Rd specifies the register to put the current map register value into. New source data and the current data in the map register are swapped in an atomic fashion.

Specifying Rs1 as x0 causes the map move operation to only output the current map value without updating it.

The Rs2 field specifies a 32-bit value broken into two fields. The low order twelve bits are a map register number for a given task. Bits 16 to 20 specify the task number for which the map is updated. The mapping register is only nine bits wide. Upper bits from the source register are ignored.

### Rs2 Value Format

|  |  |  |  |
| --- | --- | --- | --- |
| 31 21 | 20 16 | 15 12 | 11 0 |
| ~ | ASID | ~ | Virtual Page Number |

### Rs1 / Rd Value Format

|  |  |  |  |
| --- | --- | --- | --- |
| 31 21 | 20 16 | 15 14 | 13 0 |
| ~ | ~ | ~ | Physical Page Number |

**Instruction Format**: OSR2

**Execution Units**: OSU

**Exceptions**: none

## MVSEG – Move Segment Register

**Description**:

MVSEG works in a manner like the CSR instruction, but is applied for segment register access only. Register Rs2 indirectly identifies the segment register to access. Note that Rs2 is an integer register that contains the segment register number. Rs1 identifies source data for the segment register, and Rd specifies the register to put the current segment register value into. New source data and the current data in the segment register are swapped in an atomic fashion.

**Instruction Format**: MVSEG

**Exceptions**: none

## PEEKQ – Peek at Queue

**Description**:

This instruction returns the top value into Rd from the hardware queue specified in Rs1. The hardware queue position is not advanced. The value returned in Rd includes status bits in addition to the value pushed. The value field is an N-bit field between 1 and 48 bits in size which is configuration dependent. Unused value bits should read as zero.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 | 62 | 60 54 | 53 48 | 47 0 |
| Qe | Dv | ~ | DC | Value |

Fields

Qe: queue empty.If set, this bit indicates that the queue is empty.

Dv: data valid. If this bit is set it indicates that the N-bit value field is a valid queue data.

Dc: data count: The number of items left in the queue

Value: the value that was pushed to the queue

**Instruction Format**: PUSHQ

**Exceptions:** none

## PFI – Poll for Interrupt

**Description**:

This instruction causes the processor to check for the presence of an interrupt then perform interrupt processing if an interrupt is present. Otherwise program execution continues with the next instruction. Interrupts do no have to be enabled for the PFI instruction to perform interrupt processing. Effectively PFI temporarily enables interrupts for the duration of the instruction.

**Instruction Format**: PFI

**Exceptions**: none

## POPQ – Pop from Queue

**Description**:

This instruction pops a value into Rd from the hardware queue specified in Rs1. The hardware queue position is advanced. The value returned in Rd includes status bits in addition to the value pushed. The value field is an N-bit field between 1 and 48 bits in size where N is configuration dependent. Unused value bits should read as zero.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 | 62 | 61 54 | 53 48 | 47 0 |
| Qe | Dv | ~ | DC | Value |

Fields

Qe: queue empty.If set, this bit indicates that the queue is empty.

Dv: data valid. If this bit is set it indicates that the N-bit value field is a valid queue data.

Dc: data count: The number of items left in the queue

Value: the value that was pushed to the queue

**Instruction Format**: PUSHQ

**Exceptions:** none

## PUSHQ – Push on Queue

**Description**:

This instruction pushes an N-bit value in Rs1 onto the hardware queue specified in Rs2. Where N is implementation defined between 1 and 48 bits.

**Instruction Format**: PUSHQ

**Exceptions:** none

## SETKEY – Set Memory Key

**Description**:

The SETKEY instruction both gets the current key and sets the memory key for a given physical page of memory. Memory keys are stored in their own dedicated memory, one key for each memory page in the system. Register Rs1 is used to specify both which key to set and the key value to set. The lower 20 bits of Rs1 specify the key value, bits 32 to 45 specify the memory page. The current value of the key is returned in Rd.

It is possible to simply retrieve the key without setting it. If the G bit is set in Rs1 the key will be retrieved without setting it.

*The author feels that using a dedicated instruction for the purpose of setting a memory key is worth it for the conveyance of meaning in software programs. While the key memory could be part of the memory map and accessible with load / store instructions it is not implemented as such.*

*There is room to increase the size of the key or the number of memory pages in the register value format.*

**Rs1 Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 46 | 45 32 | 31 | 30 20 | 19 0 |
| ~ | Physical Page Number | G | ~ | Key Value |

**Instruction Format**: MVSEG

**Exceptions**: none