



# NVIO3

## ABSTRACT

Details of the NVIO3 superscalar processor core including instruction set description.

Robert Finch

# NVIO3

©2019 Robert Finch

“If you build it, he will come” – Field of Dreams

## Introduction

### Features

- 128-bit integer data path
  - 128-bit quad precision floating-point data path
  - 32-entry integer register file
  - 32-entry floating-point register file
  - 32-entry vector register file
  - 8 link registers
  - 8 mask registers
  - 40-bit fixed size instructions
  - 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
- nvio3 fetches four instructions at once and can issue up to seven instructions in a single cycle (2 alu, 1 flow control, 2 floating point, 2 memory) and is capable of committing up to four instructions in a single cycle.

### History

NVIO3 is a work-in-progress beginning in July 2019. NVIO3 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 a number of other processing cores.

### Design Rationale

Some may argue it's too soon and unnecessary to design a 128-bit cpu. This is true, designing the core is a reaching exercise for the author. Some of the rationale for doing so is quad precision floating-point is 128-bits. In order to support the quad precision fp internal registers and busses need to be at least 128 bits in size. The author decided to just do a full 128-bit design in this case. Quad floating-point offers good precision with which to do business apps. It's approximately 38 digits of precision. Ordinary double-precision arithmetic only offers about 19 digits, which isn't quite enough for some business apps once things like rounding are considered.

## 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     |
| 40             | penta | LDP, STP     |
| 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.

### 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 NVIO3 ISA is a 32-register machine with separate register files for integer, floating-point and array operations. There are a large number of control and status (CSR) registers which hold an assortment of specific values relevant to processing. The ISA features code address registers (link registers) and condition code registers in addition to the integer and float register files.

### General Purpose Vector Registers (r0 to r31)

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.

R0 always has the value zero. Registers r31 and r30 are used for stack references and subject to stack bounds checking.

Each register may hold a vector of values to a total width of 256 bits.

| Register | Description / Suggested Usage     | Saver  |
|----------|-----------------------------------|--------|
| r0       | always reads as zero              |        |
| r1-r2    | return values / exception         | caller |
| r3-r10   | temporaries                       | caller |
| r11-r16  | register variables                | callee |
| r17-r21  | function arguments                | caller |
| r22-r24  | garbage collector                 |        |
| r25      | type number / function argument   | caller |
| r26      | class pointer / function argument | caller |
| r27      | thread pointer                    | callee |
| r28      | global pointer                    |        |
| r29      | exception SP offset               | callee |
| r30      | base / frame pointer              | callee |
| r31      | stack pointer (hardware)          | callee |

### Floating Point Vector Registers (fp0 to fp31)

| Register | Description / Suggested Usage | Saver |
|----------|-------------------------------|-------|
| fp0      | always reads as positive zero |       |
| fp1-fp31 |                               |       |

### Link Registers (Lk0 to Lk7)

Link registers (also maybe called code address registers) are used to store the subroutine return address or the target address for register indirect calls. There are eight link registers available (Lk0 to Lk7), however link register zero always contains the value zero.

| Register | Usage                             |
|----------|-----------------------------------|
| Lk0      | always zero                       |
| Lk1      | subroutine return address         |
| Lk2      | subroutine return address level 2 |
| Lk3      | subroutine return address level 3 |
| Lk4      |                                   |
| Lk5      |                                   |
| Lk6      | catch link address                |
| Lk7      | interrupted instruction pointer   |

### Compare Results (Cr0 to Cr7)

Each of these registers may contain the result flags from a comparison operation. Each register is only eight bits wide. Many ALU and memory instructions may set a compare results register.

| Register | Usage                       |
|----------|-----------------------------|
| Cr0      | integer instructions        |
| Cr1      | floating point instructions |
| Cr2      |                             |
| Cr3      |                             |
| Cr4      |                             |
| Cr5      |                             |
| Cr6      |                             |
| Cr7      |                             |

### Integer Results

|    |   |    |    |    |    |    |    |
|----|---|----|----|----|----|----|----|
| 7  | 6 | 5  | 4  | 3  | 2  | 1  | 0  |
| UF | ~ | OF | OD | CF | GT | LT | EQ |

### Floating Point Results

|    |   |     |    |    |    |    |    |
|----|---|-----|----|----|----|----|----|
| 7  | 6 | 5   | 4  | 3  | 2  | 1  | 0  |
| UF | ~ | INF | OD | UN | GT | LT | EQ |

### Mask Registers (m0 to m7)

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

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

Mask registers are organized with 32 bits reserved each vector register. The number of bits that are significant depend on the size of the operation performed. For example with an operation size of 32 bits, eight mask register bits are significant ( $32 \times 8 = 256$ ).



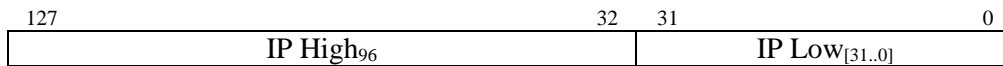
## Register Tag Map

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.

| Tag        | Associated Register |                               |
|------------|---------------------|-------------------------------|
| 0 to 31    | GP0 to GP31         | general purpose registers     |
| 32 to 63   | FP0 to FP31         | floating point registers      |
| 64 to 95   | reserved            |                               |
| 96 to 103  | LK0 to LK7          | link registers                |
| 104 to 111 | M0 to M7            | mask registers                |
| 112 to 119 | CR0 to CR7          | compare result registers      |
| 120        | VL                  | vector length register        |
| 121        | CR <all>            | all compare results registers |
| 122 to 126 | reserved            | not used                      |
| 127        | none                | instruction without a target  |

## Instruction Pointer

The instruction pointer 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 40-bit instruction parcels. The instruction pointer increments in terms of instructions not bytes. The memory byte address is five times the instruction pointer address. The instruction pointer register is also split into two sections. Conditional jump instructions only update the lower 25-bits. Only the lower 32 bits of the IP increment.



## Register Zero

Register zero – r0 – always reads as zero.

## 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 hexibyte boundaries. That is, they should be a multiple of sixteen, which has the least significant bit as zero. There is currently no hardware in the core to enforce alignment.

## Control and Status Registers

### Overview

There are a large number of control and status registers. All bits of a CSR may not be implemented in hardware.

### Control Register Zero (CSR #000)

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

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

This register supports bit set / clear CSR instructions.

#### DCE

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

#### BPE

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

#### WBM bit

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

#### SPLE

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

### HARTID (0x001)

This register contains a number that is externally supplied on the hartid\_i input bus to represent the hardware thread id or the core number. No core should have the value zero as the hartid.

### TICK (0x002)

This register contains a tick count of the number of clock cycles that have passed since the last reset. Note that this register should not be used for precise timing as the processor's clock

frequency may vary for performance and power reasons. The TIME CSR may be used for wall-clock timing as it has its own timing source.

### PCR Paging Control (CSR 0x003)

This register controls the paged memory management unit. A more detailed description is available under the section on memory management.

### AEC Arithmetic Exception Control (CSR 0x004)

This register has controls to enable arithmetic exceptions and status bits to indicate the occurrence of exception conditions.

| Exception Occurrence |     |     |     |     |     | Exception Enable |     |     |     |     |     |
|----------------------|-----|-----|-----|-----|-----|------------------|-----|-----|-----|-----|-----|
| 63<br>37             | 36  | 35  | 34  | 33  | 32  | 31<br>5          | 4   | 3   | 2   | 1   | 0   |
|                      | DIV | MUL | ASL | SUB | ADD |                  | DIV | MUL | ASL | SUB | ADD |

### PMR Power Management Register (CSR 0x005)

This register contains bits to disable functional units in the core to conserve power. There is a bit for each functional unit.

|  |  |  |       |       |       |      |           |
|--|--|--|-------|-------|-------|------|-----------|
|  |  |  | 39 32 | 31 24 | 23 16 | 15 8 | 7 0       |
|  |  |  | FCU   | MEM   | FPU   | ALU  | Inst. Dec |

It is not possible to disable all instruction decoders at the same time. There will always be at least one active decoder. Similarly, for the ALU; ALU #0 is always enabled. There will always be at least one memory channel on. The FCU can't be turned off, however the performance enhancing components can be. The branch predictor, branch target buffer and return stack buffer may all be turned off as a single unit. Power management features are not currently implemented.

### CAUSE (0x006)

This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code in order to determine what to do. Only the low order 16 bits are implemented. The high order bits read as zero and are not updateable. The low order eight bits are loaded from the cause field of the BRK instruction. The next eight bits are loaded from the user<sub>6</sub> field of the break instruction.

### BADADDR (CSR 0x007)

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

### PCR2 Paging Control (CSR 0x008)

This register controls the paged memory management unit. A more detailed description is available under the section on memory management.

### Scratch (CSR 0x009)

This register is available for scratchpad use. It is typically swapped with a GPR during exception processing.

**WBRC D (CSR 0x00A)**

WBRC D stands for ‘write barrier record’. This register records the occurrence of a pointer store operation done by an instruction using a particular register set. There is a separate bit for each register set in the machine. This register is used by the garbage collector (GC) to determine if a scan should occur on a set of registers.

**BAD\_INSTR (CSR 0x00B)**

This register contains a copy of the exceptioned instruction.

**SEMA (CSR 0x00C) Semaphores**

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

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

**KEYS – (CSR 0x00D)**

This register contains the collection of keys associated with the process for the memory system. Each key is twenty bits in size. The register contains six keys. CSR 0x00D is searched in parallel for keys matching the one associated with the memory page.

|         |         |       |       |       |       |      |
|---------|---------|-------|-------|-------|-------|------|
| 127 120 | 119 100 | 99 80 | 79 60 | 59 40 | 39 20 | 19 0 |
| ~       | key6    | key5  | key4  | key3  | key2  | key1 |

**KEYS – (CSR 0x00E)**

This register is reserved for future use to expand the number of keys searched.

**TCB (CSR 0x010)**

This CSR register is reserved for use as a pointer to a control block for the currently running thread.

**WRS (CSR 0x011)**

This register indicates which register sets are wired, or perpetually present in the core. Register sets beginning with register set #0 and progressing to the value supplied in this register are wired. There are only 16 register sets in the machine. Wiring too many register sets may make it difficult for the OS to switch threads.

## 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   |
|----------------------------------|------------|----------|---|
| 79:44                            | ~          |          | reserved  |
| 43:40                            | <b>PRC</b> |          | Default precision   |
| 39:32                            | <b>RGS</b> |          | Register set  |
| 31:29                            | <b>RM</b>  | rm       | rounding mode   |
| 28                               | <b>E5</b>  | inexe    | - inexact exception enable  |
| 27                               | <b>E4</b>  | dbzxe    | - divide by zero exception enable   |
| 26                               | <b>E3</b>  | underxe  | - underflow exception enable  |
| 25                               | <b>E2</b>  | overxe   | - overflow exception enable   |
| 24                               | <b>E1</b>  | invopxe  | - invalid operation exception enable  |
| 23                               | <b>NS</b>  | ns       | - non standard floating point indicator   |
| <b>Result Status</b>             |            |          |   |
| 22                               |            | fractie  | - the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception) |
| 21                               | <b>RA</b>  | rawayz   | rounded away from zero (fraction incremented)   |
| 20                               | <b>SC</b>  | C        | denormalized, negative zero, or quiet NaN   |
| 19                               | <b>SL</b>  | neg <    | the result is negative (and not zero)   |
| 18                               | <b>SG</b>  | pos >    | the result is positive (and not zero)   |
| 17                               | <b>SE</b>  | zero =   | the result is zero (negative or positive)   |
| 16                               | <b>SI</b>  | inf ?    | the result is infinite or quiet NaN   |
| <b>Exception Occurrence</b>      |            |          |   |
| 15                               | <b>X6</b>  | swt      | {reserved} - set this bit using software to trigger an invalid operation  |
| 14                               | <b>X5</b>  | inerx    | - inexact result exception occurred (sticky)  |
| 13                               | <b>X4</b>  | dbzx     | - divide by zero exception occurred   |
| 12                               | <b>X3</b>  | underx   | - underflow exception occurred  |
| 11                               | <b>X2</b>  | overx    | - overflow exception occurred   |
| 10                               | <b>X1</b>  | giopx    | - global invalid operation exception – set if any invalid operation exception has occurred                              |
| 9                                | <b>GX</b>  | gx       | - global exception indicator – set if any enabled exception has happened  |
| 8                                | <b>SX</b>  | sumx     | - summary exception – set if any exception could occur if it was enabled<br>- can only be cleared by software           |
| <b>Exception Type Resolution</b> |            |          |   |
| 7                                | <b>X1T</b> | cvt      | - attempt to convert NaN or too large to integer  |
| 6                                | <b>X1T</b> | sqrtn    | - square root of non-zero negative  |
| 5                                | <b>X1T</b> | NaNComp  | - comparison of NaN not using unordered comparison instructions   |
| 4                                | <b>X1T</b> | infzero  | - multiply infinity by zero   |
| 3                                | <b>X1T</b> | zerozero | - division of zero by zero  |
| 2                                | <b>X1T</b> | infdiv   | - division of infinities  |
| 1                                | <b>X1T</b> | subinf   | - subtraction of infinities   |
| 0                                | <b>X1T</b> | snanx    | - signaling NaN   |

### DBAD<sub>x</sub> (CSR 0x018 to 0x01B) Debug Address Register

These registers contain addresses of instruction or data breakpoints.

|               |   |
|---------------|---|
| 79            | 0 |
| Address 79..0 |   |

### DBCR (CSR 0x01C) 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)   |       |  |

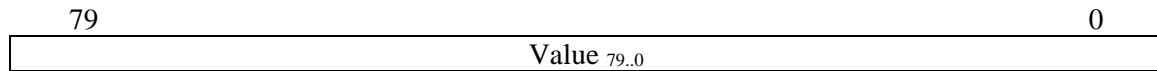
### DBSR (CSR 0x01D) - Debug Status Register

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 |
| 79 to 4 | not used, reserved             |

### CAS (CSR 0x02C) Compare and Swap

This register is to support the compare and swap (CAS) instruction. If the value in the addressed memory location identified by the CAS instruction is equal to the value in the CAS register, then the source register is written to the memory location, and the source register is loaded with the value 1. Otherwise if the value in the addressed memory location doesn't match the value in this register, then value at the memory location is loaded into the CAS register, and the source register is set to zero. No write to memory occurs if the match fails.



### TVEC (0x030 to 0x033)

These registers contain the address of the exception handling routine for a given operating level. TVEC[0] (0x030) is used directly by hardware to form an address of the interrupt routine. The lower eight bits of TVEC[0] are not used. The lower bits of the interrupt address are determined from the operating level. TVEC[1] to TVEC[3] are used by the REX instruction.

### CMDPARM (0x038 to 0x03F)

These registers are used to pass command parameters to the operating system.

### IM\_STACK (0x040)

This register contains the interrupt mask stack. When an exception or interrupt occurs, this register is shifted to the left and the low order bits are set to all ones, when an RTI instruction is executed this register is shifted to the right. On RTI the last stack entry is set to fifteen masking all interrupts on stack underflow. The low order four bits represent the current interrupt mask level. Only the low order 40 bits of the register are implemented.

### OL\_STACK (0x041)

This register contains the operating and data level stack. When an exception or interrupt occurs, this register is shifted to the left, when an RTI instruction is executed this register is shifted to the right. On RTI the last stack entry is set to zero which will select the machine operating level on stack underflow. The low order 20 bits are the code/stack operating level, bits 32 to 51 are the data operating level. Note there is extra unused space in this register to accommodate a change to more threads or greater stack depth. Bit 0,1 represent the current operating level. Bits 32,33 represent the current data level.

### PL\_STACK (0x042)

This register contains the privilege level stack. When an exception or interrupt occurs, this register is shifted to the left, when an RTI instruction is executed this register is shifted to the right. On RTI the last stack entry will be set to zero which will select privilege level zero on stack underflow. The low order eight bits of the register represent the current privilege level.

### RS\_STACK (0x043)

This register contains the register set selection stack. When an exception or interrupt occurs, this register is shifted to the left and register set #0 is selected, when an RTI instruction is executed this register is shifted to the right. On RTI the last stack entry will be set to zero which will select register set #0 on stack underflow.

**STATUS (0x044)**

This register contains the interrupt mask, operating level, and privilege level.

| Bitno    | Field | Description  |
|----------|-------|--|
| 0 to 3   | IM    | active interrupt mask level  |
| 4 to 5   | ~     | reserved   |
| 6 to 13  | PL    | privilege level  |
| 14 to 19 | RS    | register set selection – general purpose registers, this also controls which bounds register set is viewable in the CSRs.  |
| 20       | SX    | software exception – typically set by a throw() operation and cleared in a catch() handler.  |
| 20 to 21 | ~     | reserved   |
| 24 to 27 | Thrd  | active thread  |
| 28 to 31 | IRQ   | The level of interrupt that caused the hardware BRK.   |
| 32       | VCA   | indicates that vector chaining was active prior to an exception  |
|          |       |  |
| 40 to 47 | ASID  | active address space identifier  |
| 48 to 49 | FS    | floating point state   |
| 50 to 51 | XS    | additional core extension state  |
| 55       | MPRV  | memory privilege: This bit when true (1) causes memory operations to use the first stack privilege level when evaluating privilege and protection rules. (Bits 0 to 13 in the status reg). |
| 56 to 60 | VM    | These bits control virtual memory options. Note that multiple options may be present at the same time. At reset all the bits are set to zero.  |
| 63       | SD    |  |

**VM<sub>5</sub>**

| Bit | Indicates                            |  |
|-----|--------------------------------------|--|
| 0   | 1 = single bound                     |  |
| 1   | 1 = separate program and data bounds |  |
| 2   | 1 = lot protection system            |  |
| 3   | 1 = simplified paged unit            |  |
| 4   | 1 = paging unit                      |  |

**BRS\_STACK (0x046)**

This register contains the register set selection stack for base and bounds registers. When an exception or interrupt occurs, this register is shifted to the left and the current status copied to the low order bits, when an RTI instruction is executed this register is shifted to the right and the status bits copied from the low order bits of the register. On RTI the last stack entry will be set to eight which will select register set #8 on stack underflow.

Normally the brs\_stack matches the rs\_stack, but it's convenient to have a different register for interrupt processing so that base and bounds register may be modified without switching the general-purpose register set.

**EIP (0x048 to 0x051)**

This sets of registers contains the interrupt or exception stack of the instruction pointer register. The top of the stack is register 0x48. When an interrupt or exception occurs register 0x48 to 0x50 are copied to the next register and the instruction pointer is placed into register 0x48. When an



RTI instruction is executed the instruction pointer is loaded from register 0x048 and registers 0x048 to 0x050 are loaded with the next register. Register 0x051 is loaded with the address of the break handler so that in the event of an underflow the break handler will be executed.

#### CODEBUF (0x080 to 0x0BF)

This register range is for access to 64 adaptable code buffers. The code buffers are used by the EXEC instruction to execute code which may change at run-time. Bits 0 to 39 of the code buffer are used to encode the instruction. Bits 40 to 46 of the code buffer are used to encode the template associated with the instruction. See the EXEC instruction for more details.

#### IQ\_CTR (0xFC0)

This register contains a 40-bit count of the number of instructions queued since the last reset.

#### BM\_CTR (0xFC1)

This register contains a 40-bit counter of the number of branch misses since the last reset.

#### IRQ\_CTR (0xFC3)

This register is reserved to contain a 40-bit count of the number of interrupt requests.

#### BR\_CTR (0xFC4)

This register contains a 40-bit counter of the number of branches committed since the last reset.

#### TIME (0xFE0)

The TIME register corresponds to the wall clock real time. This register can be used to compute the current time based on a known reference point. The register value will typically be a fixed number of seconds offset from the real wall clock time. The lower 40 bits of the register are driven by the tm\_clk\_i clock time base input which is independent of the cpu clock. The tm\_clk\_i input is a fixed frequency used for timing that cannot be less than 10MHz. The low order 40 bits represent the fraction of one second. The upper 40 bits represent seconds passed. For example, if the tm\_clk\_i frequency is 100MHz the low order 40 bits should count from 0 to 99,999,999 then cycle back to 0 again. When the low order 40 bits cycle back to 0 again, the upper 40 bits of the register is incremented. The upper 40 bits of the register represent the number of seconds passed since an arbitrary point in the past.

Note that this register has a fixed time basis, unlike the TICK register whose frequency may vary with the cpu clock. The cpu clock input may vary in frequency to allow for performance and power adjustments.

#### INSTRET (0xFE1)

This register contains a count of the number of instructions retired (successfully completed) by the core.

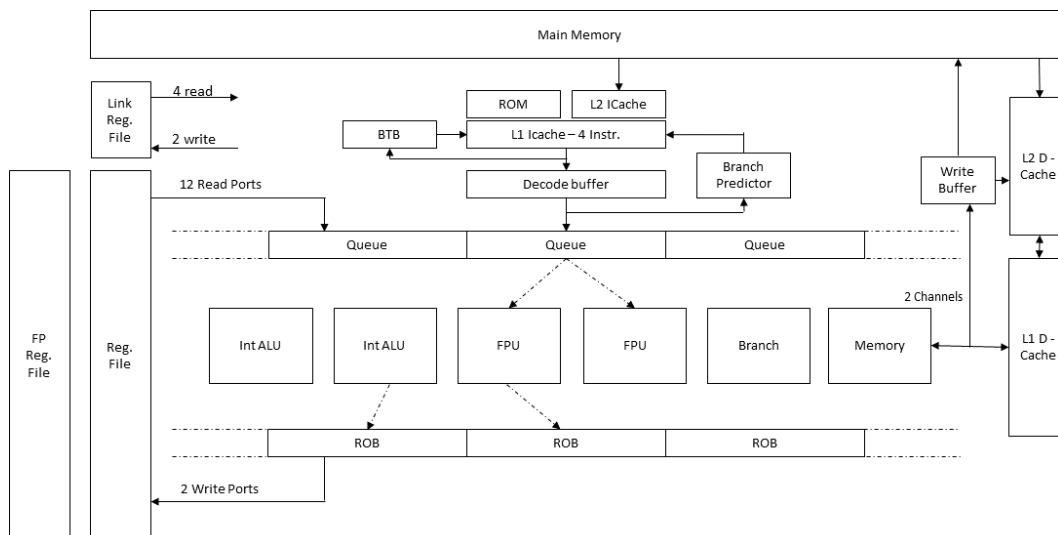
#### INFO (0xFF0 to 0xFFF)

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

## Pipeline

The pipeline is a typical superscalar pipeline as can be seen in the diagram below. A ROM with a two cycle latency feeds directly into the L1 cache. Up to four instructions are decoded and queued in a single cycle. After the instruction is queued it is assigned a re-order buffer entry. Once an instruction is finished results are transferred to the re-order buffer. Re-order buffer results are transferred to the register file in program order.

## Pipeline Diagram



## Caches

### Overview

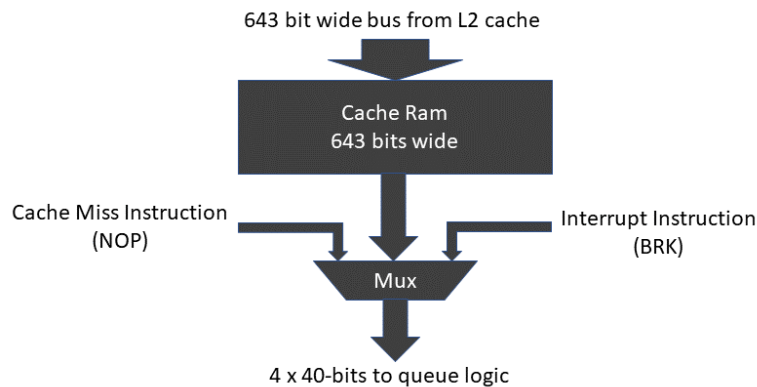
The core has both instruction and data caches to improve performance. Both caches are a two-level caches (L1, L2) allowing better performance. The first level cache is four-way set associative, the second level cache is also four-way set associative. The author initially had the first level cache fully associative based on a cam memory but found the resource requirements for the cam memory to be too large. The cache sizes of the instruction and data cache are available for reference from one of the INFO CSR registers.

### Instructions

Since the instruction format affects the cache design it is mentioned here. For this design instructions are a single size. Specific formats are listed under the instruction set description section of this book.

### L1 Instruction Cache

L1 is 5kB in size and made from distributed ram to get single cycle read performance. L1 is organized as 64 lines of 80-bytes. The following illustration shows the L1 cache organization for NVIO.



A 64-line cache was chosen as that matches the inherent size of single distributed ram component in the FPGA. It is the author's opinion that it would be better if the L1 cache were larger because it often misses due to its small size. However, using a larger distributed ram means going outside of the single lookup-table (LUT) and may then affect the clock cycle time. In short, the current design is an attempt to make it easy for the tools to create a fast implementation.

Note that supporting interrupts and cache misses, a requirement for a realistic processor design, adds complexity to the instruction stream. Reading the cache ram, selecting the correct instruction word and accounting for interrupts and cache misses must all be done in a single clock cycle.

While the L1 cache has single cycle reads it requires two clock cycles to update (write) the cache. The cache line to update needs to be provided by the tag memory which is unknown until after the tag updates.

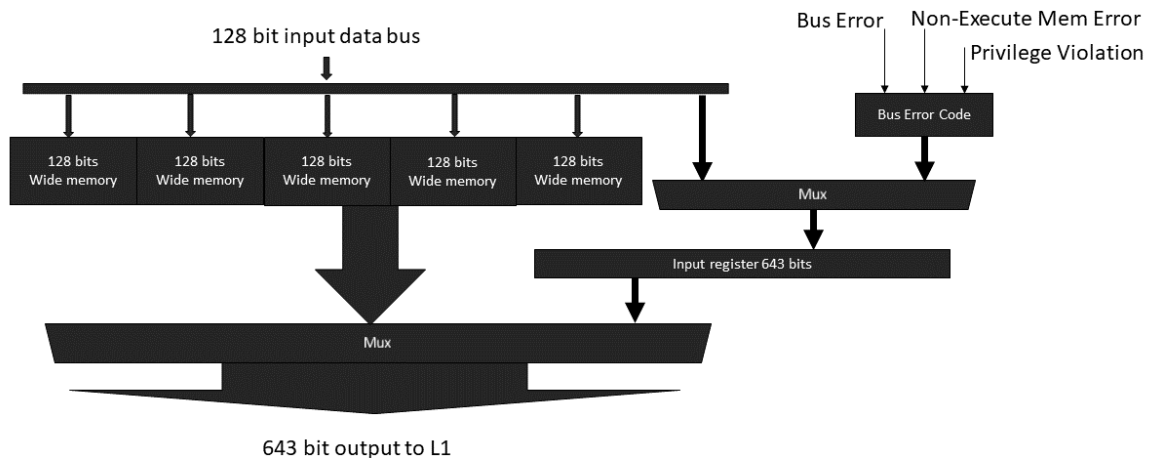
One may have noticed the cache line size of 643 bits, the line size provides for sixteen instructions plus three bits which record error (tlb miss) state. One may also notice the lack of an instruction aligner, which is not required since instructions are fixed in size.

## L2 Instruction Cache

L2 is 40kB in size implemented with block ram. L2 is organized as 512 lines of 80 bytes. There is more flexibility in the design of the L2 cache since it's made up of block ram components. Once again, the cache is too small in the author's opinion, but it represents a trade-off in use of block ram resources for the cpu core versus using the block ram for other purposes such as memory management or data cache. There are only so many block rams in the FPGA.

The L2 cache has a read latency of three clock cycles to try and get the best clock cycle time out of the cache. It feeds the L1 cache with a cache-line wide bus so that only a single transfer cycle is required to update the L1 cache.

### L2 Cache Organization



A cache load reads five 128-bit words. Cache line size is 80 bytes or 640 bits.

An input register is used to feed the L1 cache directly on a load so that it doesn't have to wait for a read of the L2 cache before proceeding, otherwise there would be an additional three cycle delay during a cache miss.

If there is an error during the fetch, the L1 cache line is loaded with a break instruction corresponding to the error. A subsequent instruction fetch will cause the processor to execute the error routine.

The L2 cache is 40kB (512 rows \* 80 bytes) composed of block ram which has a three cycle read latency. The L2 cache is four-way set associative.

The L2 cache reads two words from memory on a cache line load. While data is loading into the L2 cache an input register is also loaded with the data, the input register is transferred to the L1 cache, this is done so that the L1 cache update doesn't have to wait for the three cycle read latency of the L2 cache. Also fed into the input register are instructions for error processing should an error occur during the cache load.



## Data Cache

The data cache organization is similar to the instruction cache with the exception that the data cache needs to be able to accept data written by store instructions as well as providing read data. The data cache also supports unaligned data access. These features make the data cache more complex than the instruction cache. Some of the latency of the data cache can be hidden by the presence of non-memory operating instructions in the instruction queue and the ability of the core to overlap data fetches.

The data cache has two read ports allowing two load operations to be in progress at the same time. The policy for stores is write-through. Stores always write through to memory. Since stores follow a write-through policy the latency of the store operation depends on the external memory system.

### L1 Data Cache

Like the L1 instruction cache, the L1 data cache is 2kB in size, making use of 64 deep LUT rams. The data cache line size is 333 bits. 328 bits are required to support unaligned data access where the maximum overflow from a 256-bit region is 72 bits. Three bits are used to record error state. Like the I-cache the D-cache is four way set associative.

### L2 Data Cache

The L2 data cache is organized as 512 lines of 32 bytes (16kB) and implemented with block ram. Access to the L2 data cache is multicycle. It isn't critical that the cache be able to update in single cycle as external memory access is bound to take many more cycles than a cache update. There is only a single write port on the data cache.

## Cache Enables

The instruction cache is always enabled to keep hardware simpler and faster. Otherwise an additional multiplexor and control logic would be required in the instruction stream to read from external memory.

For some operations it may be desirable to disable the data cache so there is a data cache enable bit in control register #0. This bit may be set or cleared with one of the CSR instructions.

## Cache Validation

A cache line is automatically marked as valid when loaded. The entire cache may be invalidated using the CACHE instruction. Individual cache lines may also be invalidated using the CACHE instruction.

## Uncached Data Area

The address range \$F...FDxxxxx is an uncached 1MB data area. This area is reserved for I/O devices. The data cache may also be disabled in control register zero.

## Write Buffering

The core has a configurable seven entry write buffer to enhance performance of store operations. The core always writes-through to memory and at the same time the cache is updated if a cache hit occurs. Loads will not occur until after the write buffer is emptied to ensure that data loaded isn't stale.

## Load Bypassing

The write buffer does not currently support load bypassing of values contained in the write buffer to the load unit. Load bypassing has a higher security risk.

## Branch Predictor

The branch predictor is a (2, 2) correlating predictor. The branch history is maintained in a 512-entry history table. It has four read ports for predicting branch outcomes, one port for each instruction in the fetch buffer. The branch predictor may be disabled by a bit in control register zero. When disabled all branches are predicted as not taken.

To conserve hardware the branch predictor uses a fifo that can queue up to two branch outcomes at the same time. Outcomes are removed from the fifo one at a time and used to update the branch history table which has only a single write port. In an earlier implementation of the branch predictor, two write ports were provided on the history table. This turned out to be relatively large compared to its usefulness.

Correctly predicting a branch turns the branch into a single-cycle operation. Branches are detected in the instruction fetch stage. During execution of the branch instruction the branch status is checked against the predicted status and if the two differ then a branch miss occurs. The miss address may be the branch's target address if the branch was supposed to be taken, otherwise it will be the address of the next instruction. If there was a branch miss, then the queue will be flushed, and new instructions loaded from the correct program path.

## Branches

### Target Address

Branches use absolute addressing. The branch range is approximately 128MB.

### Static Predictions

Branches may be statically predicted to be taken or not taken. A branch that is statically predicted does not consume space in the branch prediction tables.

### Performance

Branches should target the first slot (slot0) of a bundle for maximum performance. The core always fetches instructions at bundle aligned addresses. Branching into the middle of a bundle lowers performance because effectively fewer instructions are available for queueing. Branches should also be placed in the last slot (slot3) for maximum performance.

### Clock Cycles

Under absolutely ideal conditions, 0.25 clocks are required to perform a branch. Ideal conditions being the branch is located in instruction slot #3 (so that the prior three slots may be executing instructions). Also register values must already be valid and the branch prediction is correct. Branches are predicted and taken in the fetch stage of the processor.

The predictor is about 90% accurate. So, with an eight-entry instruction queue on average about ½ the queue has to be flushed or 4 entries. That occurs 10% of the time. Also, assuming the branch is not in the last slot, about 2 slots are wasted on average. So, an average number of clock cycles for a branch would be about  $(0.10 * 4 + 0.25) * 2 = 0.85$  clock cycles (or 1 if rounding).

## Sequence Numbers



Sequence numbers are mentioned here because they are used by branch instructions to determine what to invalidate. There are a few different ways to handle invalidating instructions in the queue that follow a branch miss. Perhaps the lowest cost and fastest means is to use branch tags. Two other means are using a set of branch shadow bitmasks and using instruction sequence numbers.

### Branch Tags

As branch instructions enter the queue, they are assigned a branch tag usually two bits. Non-branch instructions inherit the branch tag from the preceding branch. When a branch miss occurs all the instructions with the branch tag matching the missed branch are invalidated. When the branch commits to the state of the machine, the branch tag is freed up. Since there are only two bits used only four different tags are possible, meaning the processor can only speculate past four branches. In practical terms this isn't really a severe limitation.

### Branch Shadow Masks

A means which the author came up. A shadow bit is allocated in the instruction queue for each branch. For instance, to support speculating past four branches four bits are required in the instruction queue. As instructions are entered into the queue the bit is set for the preceding branch instruction. When a branch miss occurs all the instructions with the shadow bit set for that branch are invalidated. The shadow bits are available to be reassigned to new branches when the branch commits to the machine state.

### Sequence Numbers

Sequence numbers are probably conceptually the simplest means to understand. The order of instructions in the queue is associated with a number. The core assigns a sequence number to each instruction as it enters the instruction queue. One purpose of the sequence number is to allow the core to determine which instructions should be invalidated because of a branch miss. All the instructions in the queue with a sequence number coming after the branch instruction's sequence number will be invalidated. The sequence number assigned is the next highest number above that which is already in the queue. As instructions commit to the machine state, the sequence numbers of following instructions are decremented by the value of the sequence number of the committing instruction. This keeps the sequence numbers within range of the number of queue entries while maintaining the ordering relationship between the numbers. The sequence number mechanism allows the core to speculate across any number of branches that might be in the instruction queue. However, sequence numbers require more bits in the instruction queue than branch tags and even more hardware. Since every instruction in the queue needs a number the number of bits required for the sequence number depends on the queue size. For a fifteen-entry queue a five-bit sequence number is required. For debugging purposes, it can be handy to see the sequence number assigned to the instruction.

**Branch Target Buffer (BTB)**

The core has a 1k entry branch target buffer for predicting the target address of flow control instructions where the address is calculated and potentially unknown at time of fetch. Instructions covered by the BTB include jump-and-link, interrupt return and breakpoint instructions and branches to targets contained in a register.

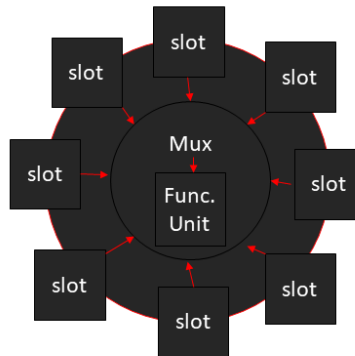
**Return Address Stack Predictor (RSB)**

There is an address predictor for return addresses which can in some cases eliminate the flushing of the instruction queue when a return instruction is executed. The RET instruction is detected in the fetch stage of the core and a predicted return address used to fetch instructions following the return. JAL instructions using the link register as the source are also treated as return instructions. The return address stack predictor has a stack depth of 32 entries. On stack overflow or underflow, the prediction will be wrong, however performance will be no worse than not having a predictor. The return address stack predictor checks the address of the instruction queued following the RET against the address fetched for the RET instruction to make sure that the address corresponds.

## Instruction Queue (ROB)

The instruction queue is a multi-entry re-ordering buffer (ROB). The size of the instruction queue is configurable between 4 and 16 entries. The instruction queue tracks an instructions progress and provides a holding place for operands and results. Each instruction in queue may be in one of a number of different states. The instruction queue is a circular buffer with head and tail pointers. Instructions are queued onto the tail and committed to the machine state at the head.

## Instruction Queue – Re-order Buffer



The instruction queue is circular with eight slots. Each slot feeds a multiplexor which in turn feeds a functional unit. Providing arguments to the functional unit is done under the use of issue logic. Output from the functional unit is fed back to the same queue slot that issued to the functional unit. The queue slots are fed from the fetch buffers.

## Queue Rate

Up to three instructions may queue during the same clock cycle depending on the availability of queue slots.

## Sequence Numbers

The queue maintains a five-bit instruction sequence number which gives other operations in the core a clue as to the order of instructions. The sequence number is assigned when an instruction queues. The number assigned is one more than the maximum sequence number found in the queue. As instructions commit, the sequence numbers of all the remaining queue entries are decremented. This keeps the sequence number within five bits. Branch instructions need to know when the next instruction has queued in order to detect branch misses. The instruction pointer cannot be used to determine the instruction sequence because there may be a software loop at work which causes the instruction pointer to cycle backwards even though it's really the next instruction executing.

## Queueing of Flow Control Operations

Flow control operations are not done until sometime after the next instruction queues. This is necessary to determine address miss-predicts during the flow control operation. Waiting until the next instruction queues avoids the problem of false mis-predictions. A consequence of waiting for the next instruction to queue is that flow control operations may only issue from one of the first

seven queue slots relative to the head of the queue. Note however that if the instruction queue is full the flow control operation will issue anyway otherwise the core could become deadlocked. When the core issues a flow control operation because the queue is full it will most likely cause a branch-miss state, which may reduce performance.

## Issue Logic

Issue logic is responsible for assigning instructions to functional units. Instructions cannot be issued unless all operands are available, and the functional unit is also available.

The amount of issue logic required grows at a more than linear rate corresponding to the number of queue entries in the re-order buffer. This is in part due to the need for the issue logic to be synchronous in nature for an FPGA. There are more elegant ways to implement the issue logic using asynchronous loops, however these are not possible with an FPGA implementation. The amount of issue logic may be reduced by a core configuration define at some loss of performance. Since instructions that have just queued at the tail of the queue are unlikely to be ready to be processed the issue logic for those queue entries can be omitted. Instructions more towards the head of the queue will be more likely to be ready to issue.

## Issue Rate

The functional units of NVIO include two alu's, two floating-point units, a memory unit, and a flow control unit (branch unit). Instructions may be issued to any and all functional units during a single clock cycle. The memory unit can handle two requests at the same time. As a result, up to seven instructions may be issued in a single clock cycle. In practice fewer instructions will be ready to be issued. The author has noted, with limited testing, that issuing a third memory operation in the same clock cycle is rarely done. The memory unit is typically 2/3 occupied or less.

## Execute Logic

Instructions are executed on functional units after they have been issued to the unit. The execution logic for NVIO consists of two alu's, two address generators, two floating-point units, a flow control unit and a memory unit.

### ALU's

Most instructions execute on one of two alu's. The alu's are asymmetrical. The first alu supports all operations including rarely performed operations, the second alu supports a subset of the operations which represents the most commonly performed operations. Splitting the functionality like this allows the core to support a wide variety of instructions while at the same time not using too many resources for rarely used operations. The issue logic knows about the difference in the ALU's and will issue what it can to the second alu if the first is busy. It's best to intermix commonly used instructions with rarely used ones to keep both alu's busy.

### *Shift Operations*

The immediate constant for a shift is limited to six bits or the values 0 to 63. If it's required to shift by more than 63 bits two shift operations will be needed, or the value of the shift may come from a register.

### *Fixed Point Arithmetic*

The core features the capability to perform fixed point arithmetic including addition, subtraction, multiplication and division. The point position is at  $\frac{1}{2}$  the machine width or 40 binary point positions.

### AGEN's

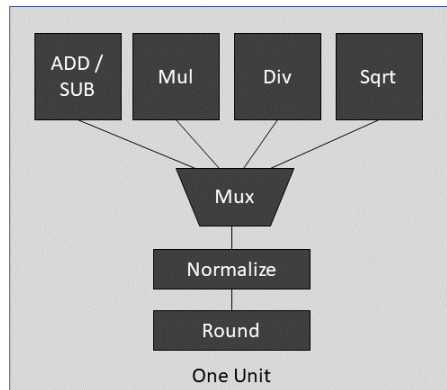
The core supports dual address generators which compute memory addresses. The address generators handle both register indirect with displacement and indexed register indirect addressing modes. The stack address for push operations and the effective address for the LEA instruction are also computed by the agen.

### Floating Point Unit

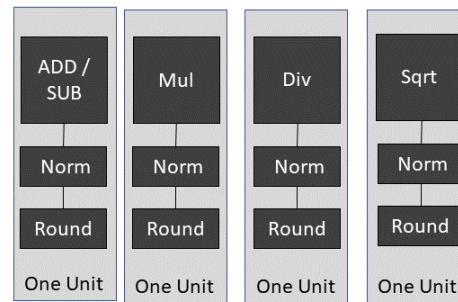
The floating-point unit is used to execute almost all floating-point operations. The exception is floating point branches which are executed on the flow control unit. The length of time required to complete a floating-point operation varies depending on the instruction.

It would be better for performance if the floating-point unit were broken apart into several separate units. Each unit would have associated issue logic. That way the pipelining of the individual operations could be put to better use. Some machines have the fp multiplier separate from fp addition/subtraction and other units. This could be done but would require more resources from the FPGA. Having multiple floating-point units would also help.

Floating-Point Unit Organization



Alternate, Faster Floating-Point Unit Organization



The current fp organization doesn't make good use of pipelining; one operation must complete before the next one can start. A faster fp organization allowing the use of pipelining is shown to the right. Other organizations with good performance are possible. The current unit focuses on small size.

Note that the ISA isn't closely related to the implementation of the floating-point unit. There is no reason why faster floating point wouldn't be possible from an ISA perspective.

### Flow Control Unit / Branch Unit

A single flow control (or branch) unit takes care of all the flow control instructions the core supports. The author prefers to call the unit a flow control unit rather than a branch unit, because the unit takes care of additional instructions besides simple branches. A branch implies multiple paths of execution. For some instructions for instance a jump or a call there is only one path of execution, calling them a branch is a bit of a misnomer. In an initial version of the core flow control operations were performed by the alu's. This led to problems of prioritization of flow control operations when two flow control operations were taking place at the same time. Especially given that the operations may have been assigned out of order to the alu's. Moving from the alu pair to a single flow control unit resolved those problems.

The flow control unit is responsible for determining whether a branch should be taken. However, by the time the branch reaches the flow control unit, it has already taken a predicted path of execution. So, the flow control unit's real job is to verify that the correct path was taken.

The flow control unit has a small alu to calculate register increment or decrement, stack pointer adjustment, and return addresses. The fcu's alu is called NVIO\_fcu\_calc.v to avoid confusion with the master alu.

### Memory Unit

The memory unit can handle up to two requests at a time. If the data cache is enabled loads check for data in the data cache, which is loaded with data if the data is not present in the cache. Up to two load operations may be taking place at the same time, each one making use of a different read port of the data cache.

Un-cached loads and stores access external memory and hence are serialized. Since there is only a single port to external memory access takes place one request at a time.

### Stores

The write buffer is dual ported and will accept two store operations at the same time. They will be placed in the write queue in program order.

## Operating Levels

The core has four operating levels. The highest operating level is operating level zero which is called the machine operating level. Operating level zero has complete access to the machine. Other operating levels may have more restricted access. When an interrupt occurs, the operating level is set to the machine level. The core vectors to an address depending on the current operating level. When operating at level zero addresses are not subjected to translation and the virtual address and physical address are the same.

| Operating Level | Privilege Level | Moniker    |
|-----------------|-----------------|------------|
| 3               | 7 to 255        | user       |
| 2               | 2 to 6          | supervisor |
| 1               | 1               | hypervisor |
| 0               | 0               | machine    |

### Switching Operating Levels

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

## Privilege Levels

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

## **Debug Mode**

In debug mode the core single steps one instruction at a time.



## TLB – The Translation Lookaside Buffer

### Overview

The TLB (translation look-aside buffer) offers a means of address virtualization and memory protection. A TLB works by caching address mappings between a real physical address and a virtual address used by software. The TLB is managed by software triggered when a TLB miss occurs. The TLB deals with memory organized as pages. Typically, software manages a paging table whose entries are loaded into the TLB as translations are required.

The TLB keeps a reference count for each map entry stored in the TLB. The upper 24-bits of the reference count, which is a 32-bit saturating counter, are automatically incremented with each memory access to the page. Reference counts are subject to aging under control of the AFC register. The reference counts may be read or written with the TBLRDAGE or TLBWRAGE commands.

The TLB is manipulated with the [TLB](#) instruction.

### Size / Organization

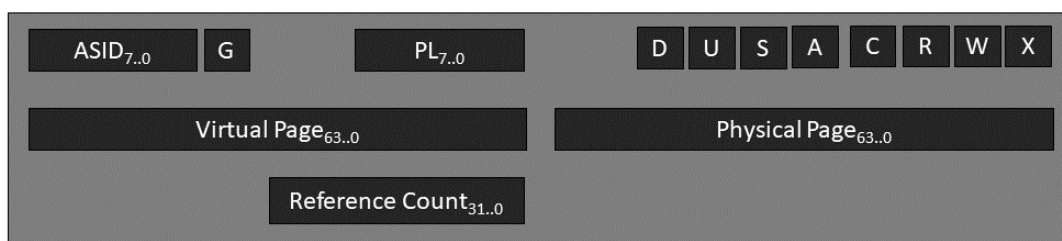
The core uses a 256 entry TLB (translation look-aside buffer) to support virtual memory. The TLB supports variable page sizes from 8kB to 2MB. The TLB is organized as a sixteen-way sixteen-set cache. The TLB processes all addresses leaving the core including both code and data addresses.

### Updating the TLB

The TLB is updated by first placing values into the TLB holding registers using the TLB instruction, then issuing a TLB write command using the TLB command instruction.

Address translations will not take place until the TLB is enabled. An enable TLB command must be issued using the TLB command instruction.

## TLB Entries



G = Global

The global bit marks the TLB entry as a global address translation where the ASID field is not used to match addresses.

ASID = address space identifier

The ASID field in the TLB entry must match the processor's current ASID value for the translation to be considered valid, unless the G bit is set. If the G bit is set in the TLB

entry, then the ASID field is ignored during the address comparison. The processor's current ASID is located in the machine status register.

C = cache-ability bits

If the cache-ability bits are set to 001<sub>b</sub> then the page is un-cached, otherwise the page is cached.

D = dirty bit

The dirty bit is set by hardware when a write occurs to the virtual memory page identified by the TLB entry.

A = accessed bit

This bit is set when the page is accessed.

U = undefined usage

This bit is available for OS usage

S = address shortcut

R = read bit

This bit indicates that the page is readable.

W = write bit

This bit indicates that the page is writeable

X = execute bit

This bit indicates that the page contains executable code

R,W,X = valid bit

One of these bits must be set for the address translation to be considered valid. The entire TLB may be invalidated using the invalidate all command.

## Page Table Entry

The following layout shows the page table entry structure as stored in memory. Although the page table is managed by software, this layout should be followed.

|                                 |  |                |    |                              |  |   |   |   |   |   |   |   |   |   |
|---------------------------------|--|----------------|----|------------------------------|--|---|---|---|---|---|---|---|---|---|
| 31                              |  |                | 16 | 15                           |  | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Share Count <sub>16</sub>       |  |                |    | PrivLevel                    |  | D | U | S | A | C | R | W | X |   |
| ASID <sub>8</sub>               |  | ~ <sub>4</sub> |    | Protection Key <sub>20</sub> |  |   |   |   |   |   |   |   |   |   |
| Reference Counter <sub>32</sub> |  |                |    |                              |  |   |   |   |   |   |   |   |   |   |
| Page Number <sub>32</sub>       |  |                |    |                              |  |   |   |   |   |   |   |   |   |   |

| Bit       |      |                               |  |
|-----------|------|-------------------------------|--|
| 0         | X    | 1 = executable                | Together these three fields combined indicate if the page is present. It must be at least one of readable, writeable, or executable. |
| 1         | W    | 1 = page writeable            |  |
| 2         | R    | 1 = readable                  |  |
| 3         | C    | 1 = cachable                  | Ignored for executable pages which are always cached   |
| 4         | A    | 1 = accessed                  |  |
| 5         | S    | 1 = shortcut translation      | Translation shortcut bit eg (8MiB pages)   |
| 6         | U    | undefined usage               | available to be used by OS   |
| 7         | D    | 1=dirty                       | set if the page is written to  |
| 8 to 15   | PL   | Privilege level               |  |
| 16 to 31  | SC   | Shared memory reference count | Number of times the memory page is shared.   |
| 32 to 51  | KY   | Protection key                | 20-bit key used to protect memory page   |
| 52 to 55  | Pad1 | reserved area                 |  |
| 56 to 63  | ASID | Address Space IDentifier      |  |
| 64 to 95  | RC   | Reference Count               | 32-bit page reference count  |
| 96 to 127 | PN   | Memory page number            | virtual page number associated with physical page  |

## Page Directory Entry

Page directory entries have the same format as page table entries.

## TLB Registers

### TLBWired (#0h)

This register limits random updates to the TLB to a subset of the available number of ways. TLB ways below the value specified in the Wired register will not be updated randomly. Setting this register provides a means to create fixed translation settings. For instance, if the wired register is set to two, the thirty-two fixed entries will be available.

### TLBIndex (#1h)

This register contains the entry number of the TLB entry to be read from or written to.

### TLBRandom (#2h)

This register contains a random four-bit value used to update a random TLB entry during a TLB write operation.

### TLBPageSize (#3h)

The TLBPageSize register controls which address bits are significant during a TLB lookup.

| N | Page Size |  |
|---|-----------|--|
| 0 | 8KiB      |  |
| 1 | 32kiB     |  |
| 2 | 128kiB    |  |
| 3 | 512kiB    |  |
| 4 | 2MiB      |  |
| 5 | 8MiB      |  |
|   |           |  |

### TLBPhysPage (#5h)

The TLBPhysPage register is a holding register that contains the page number for an associated virtual address. This register is transferred to or from the TLB by TLB instructions.

|                      |   |
|----------------------|---|
| 79                   | 0 |
| Physical Page Number |   |

### TLBVirtPage (#4h)

The TLBVirtPage register is a holding register that contains the page number for an associated physical address. This register is transferred to or from the TLB by TLB instructions.

|                     |   |
|---------------------|---|
| 79                  | 0 |
| Virtual Page Number |   |

### TLBASID (#7h)

The TLBASID register is a holding register that contains the address space identifier (ASID) , valid, dirty, global, and cache-ability bits associated with a TLB entry. This register is transferred to or from the TLB by TLB instructions.

|       |    |    |    |      |    |    |      |    |   |   |   |   |   |   |   |   |   |
|-------|----|----|----|------|----|----|------|----|---|---|---|---|---|---|---|---|---|
| 79    | 32 | 31 | 24 | 23   | 16 | 13 | 11   | 10 | 9 | 8 | 7 | 6 | 5 | 3 | 2 | 1 | 0 |
| ----- | PL |    |    | ASID |    |    | PgSz | G  | D | U | S | A | C | R | W | X |   |

### TLBAFC (#12) – Aging Frequency Control

This 24-bit register controls the frequency of aging applied to page reference counters. The aging counter is decremented at the core's clock frequency. When the counter underflows it triggers an aging cycle and is reloaded with the value in the AFC register. The AFC register is defaulted to 20000. This gives an aging frequency of 1000Hz with a 20MHz core clock.

## Exceptions

### External Interrupts

There is very little difference between an externally generated exception and an internally generated one. An externally caused exception will force a BRK instruction into the instruction stream. The BRK instruction contains a cause code identifying the external interrupt source.

### Polling for Interrupts

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

### Effect on Machine Status

The operating mode is always switched to the machine mode on exception. It's up to the machine mode code to redirect the exception to a lower operating mode when desired. Further exceptions at the same or lower interrupt level are disabled automatically. Machine mode code must enable interrupts at some point. This can be done automatically when the exception is redirected to a lower level by the REX instruction. The RTI instruction will also automatically enable further machine level exceptions.

### Exception Stack

The instruction pointer and status bits are pushed onto an internal stack when an exception occurs. This stack is only eight entries deep as that is the maximum amount of nesting that can occur. Further nesting of exceptions can be achieved by saving the state contained in the exception registers.

### Exception Vectoring

Exceptions are handled through a vector table. The vector table has four entries, one for each operating level the core may be running at. The location of the vector table is determined by TVEC[0]. If the core is operating at level three for instance and an interrupt occurs vector table address number three is used for the interrupt handler. Note that the interrupt automatically switches the core to operating level zero, privilege level zero. An exception handler at the machine level may redirect exceptions to a lower level handler identified in one of the vector registers. More specific exception information is supplied in the cause register.

| Operating Level | Address (If TVEC[0] contains \$FFFC0000) |                                  |
|-----------------|--|----------------------------------|
| 0               | \$FFFC0000                               | Handler for operating level zero |
| 1               | \$FFFC0020                               |                                  |
| 2               | \$FFFC0040                               |                                  |
| 3               | \$FFFC0060                               |                                  |

**Reset**

The core begins executing instructions at address \$FFFFFFFFFFFFFFFC0100. All registers are in an undefined state. Register set #0 is selected.

**Precision**

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

## Exception Cause Codes

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

| Cause Code |       | HW | Description               |  |
|------------|-------|----|---------------------------|--|
|            |       |    |                           |  |
| 1          | IBE   | x  | instruction bus error     |  |
| 2          | EXF   | x  | Executable fault          |  |
| 4          | TLB   | x  | tlb miss                  |  |
|            |       |    | FMTK Scheduler            |  |
| 128        |       | e  |                           |  |
| 129        | KRST  | e  | Keyboard reset interrupt  |  |
| 130        | MSI   | e  | Millisecond Interrupt     |  |
| 131        | TICK  | e  |                           |  |
| 156        | KBD   | e  | Keyboard interrupt        |  |
| 157        | GCS   | e  | Garbage collect stop      |  |
| 158        | GC    | e  | Garbage collect           |  |
| 159        | TSI   | e  | FMTK Time Slice Interrupt |  |
| 3          |       |    | Control-C pressed         |  |
| 20         |       |    | Control-T pressed         |  |
| 26         |       |    | Control-Z pressed         |  |
|            |       |    |                           |  |
| 32         | SSM   | x  | single step               |  |
| 33         | DBG   | x  | debug exception           |  |
| 34         | TGT   | x  | call target exception     |  |
| 35         | MEM   | x  | memory fault              |  |
| 36         | IADR  | x  | bad instruction address   |  |
| 37         | UNIMP | x  | unimplemented instruction |  |
| 38         | FLT   | x  | floating point exception  |  |
| 39         | CHK   | x  | bounds check exception    |  |
| 40         | DBZ   | x  | divide by zero            |  |
| 41         | OFL   | x  | overflow                  |  |
|            |       |    |                           |  |
| 47         |       |    |                           |  |
| 48         | ALN   | x  | data alignment            |  |
| 49         |       |    |                           |  |
| 50         | DWF   | x  | Data write fault          |  |
| 51         | DRF   | x  | data read fault           |  |
| 52         | SGB   | x  | segment bounds violation  |  |
| 53         | PRIV  | x  | privilege level violation |  |
| 54         | CMT   | x  | commit timeout            |  |
| 55         | BD    | x  | branch displacement       |  |
| 56         | STK   | x  | stack fault               |  |
| 57         | CPF   | x  | code page fault           |  |
| 58         | DPF   | x  | data page fault           |  |



|     |             |   |   |  |
|-----|-------------|---|---|--|
| 60  | DBE         | x | data bus error                              |  |
| 61  |             |   |   |  |
| 62  | NMI         | x | Non-maskable interrupt                      |  |
|     |             |   |   |  |
| 225 | FPX_IOP     | x | Floating point invalid operation            |  |
| 226 | FPX_DBZ     | x | Floating point divide by zero               |  |
| 227 | FPX_OVER    | x | floating point overflow                     |  |
| 228 | FPX_UNDER   | x | floating point underflow                    |  |
| 229 | FPX_INEXACT | x | floating point inexact                      |  |
| 231 | FPX_SWT     | x | floating point software triggered           |  |
|     |             |   |   |  |
| 240 | SYS         |   | Call operating system (FMTK)                |  |
| 241 |             |   | FMTK Schedule interrupt                     |  |
|     |             |   |   |  |
| 255 | PFI         |   | reserved for poll-for-interrupt instruction |  |

## DBG

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

## IADR

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

## UNIMP

This exception occurs if an instruction is encountered that is not supported by the processor. It is not currently implemented.

## OFL

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

## FLT

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

## DRF, DWF, EXF

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

## CPF, DPF

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

## PRIV

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

## STK

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

## DBE

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

## IBE

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

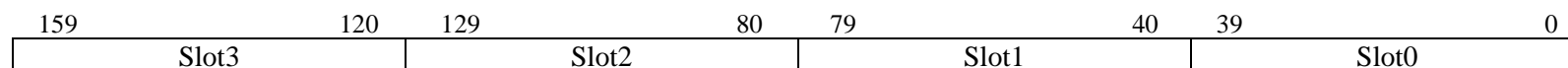
## NMI

The core does not currently support non-maskable interrupts. However, this cause value is reserved for that purpose.

## BD

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

## Bundle Format:



Slot0 This is the instruction slot for the first executed instruction in the bundle.

Slot1 This is the instruction slot for the second executed instruction in the bundle.

Slot2 This is the instruction slot for the third executed instruction in the bundle.

Slot3 This is the instruction slot for the fourth executed instruction in the bundle.

## Instruction Formats:

### Operation Sizes

Many instructions have an option to process data in sub-word data sizes including bytes, wydes, and penta-bytes. Typically, sized operations are supported only with register-register instructions. Instructions using immediate values always operate on whole words.

### SIMD

Single instruction multiple data operations treat the 256-bit operands as multiple independent lanes of data depending on the size selected. For an octa-byte size, the operands are treated as four independent 64-bit operands. For a wyde size the operands are treated as sixteen independent 16-bit operands. SIMD operations are selected by setting the parallel operation bit in the instruction (the most significant bit of the format field).

| Fmt <sub>4</sub>  | Operation Size | Number of Elements |
|-------------------|----------------|--------------------|
| Scalar Operations |                |                    |
| 0                 | byte           | 1                  |
| 1                 | wyde           | 1                  |
| 2                 | tetra-byte     | 1                  |
| 3                 | octa-byte      | 1                  |
| 4                 | hexi-byte      | 1                  |
| 5                 | 32 byte        | 1                  |
| 6, 7              | reserved       |                    |
| Vector Operations |                |                    |
| 8                 | byte parallel  | 32                 |

|    |                     |    |
|----|---------------------|----|
| 9  | wyde parallel       | 16 |
| 10 | tetra-byte parallel | 8  |
| 11 | octa-byte parallel  | 4  |
| 12 | hexi-byte parallel  | 2  |
| 13 | reserved            |    |

## Arithmetic Operations

Arithmetic operations include addition, subtraction, comparison, multiplication and division.

## Relational Operations

There are handful of instructions for determining the relationship between values. These include an assortment of set instructions.

## Logical Operations

Logical operations include bitwise and, or, and exclusive or. Inverted logical ops are also available for register instruction forms (nand, nor, and exnor).

## Shift Operations

There is a full complement of shift operations including left and right unsigned and signed shifts and left and right rotate instructions.

## Bitfield Operations

The CC80 compiler has direct support for bitfields and bitfield instructions support these operations. Bitfield operations include insert, extract, set, change and clear operations.

## Memory Operations

Memory operations include loads and stores of bytes, wydes or half-words. The core can perform loads and stores using indexed addressing.

|                 |                      |                  |                  |                  |                     |                    |                  |                     |                  |                     |                     |          |
|-----------------|----------------------|------------------|------------------|------------------|---------------------|--------------------|------------------|---------------------|------------------|---------------------|---------------------|----------|
| p               | Imm <sub>20..0</sub> |                  |                  |                  |                     |                    | Rs1 <sub>5</sub> | Rd <sub>5</sub>     |                  | Opcode <sub>8</sub> | ML                  |          |
| p               | Imm <sub>20..0</sub> |                  |                  |                  |                     |                    | Rs1 <sub>5</sub> | 0 <sub>2</sub>      | L <sub>3</sub>   | 07h <sub>8</sub>    | LDL                 |          |
| p               | Imm <sub>20..0</sub> |                  |                  |                  |                     |                    | Rs1 <sub>5</sub> | 1 <sub>2</sub>      | M <sub>3</sub>   | 07h <sub>8</sub>    | LDM                 |          |
| p               | Imm <sub>20..5</sub> |                  |                  |                  | Rs2 <sub>5</sub>    |                    | Rs1 <sub>5</sub> | Imm <sub>4..0</sub> |                  | Opcode <sub>8</sub> | MS                  |          |
| p               | Imm <sub>20..5</sub> |                  |                  |                  | 0 <sub>2</sub>      | L <sub>3</sub>     | Rs1 <sub>5</sub> | Imm <sub>4..0</sub> |                  | 27h <sub>8</sub>    | STL                 |          |
| p               | Imm <sub>20..5</sub> |                  |                  |                  | 1 <sub>2</sub>      | M <sub>3</sub>     | Rs1 <sub>5</sub> | Imm <sub>4..0</sub> |                  | 27h <sub>8</sub>    | STM                 |          |
| p               | Imm <sub>20..0</sub> |                  |                  |                  |                     |                    | 31 <sub>5</sub>  | 31 <sub>5</sub>     |                  | 2Ah <sub>8</sub>    | PUSHC               |          |
|                 | ~17                  |                  |                  |                  | Rs2 <sub>5</sub>    |                    | 31 <sub>5</sub>  | 31 <sub>5</sub>     |                  | 29h <sub>8</sub>    | PUSH                |          |
| p               | Imm <sub>20..5</sub> |                  |                  |                  | DC <sub>3</sub>     | IC <sub>2</sub>    | Rs1 <sub>5</sub> | Imm <sub>4..0</sub> |                  | 2Eh <sub>8</sub>    | CACHE               |          |
| p               | Imm <sub>20..0</sub> |                  |                  |                  |                     |                    | Rs1 <sub>5</sub> | VRd <sub>5</sub>    |                  | Opcode <sub>8</sub> | VML                 |          |
| p               | Imm <sub>20..5</sub> |                  |                  |                  | VRs2 <sub>5</sub>   |                    | Rs1 <sub>5</sub> | Imm <sub>4..0</sub> |                  | Opcode <sub>8</sub> | VMS                 |          |
| ~ <sub>9</sub>  |                      | Sc <sub>3</sub>  |                  | Rs3 <sub>5</sub> |                     | Funct <sub>5</sub> | Rs1 <sub>5</sub> | Rd <sub>5</sub>     |                  | 0Fh <sub>8</sub>    | MLX                 |          |
| ~ <sub>9</sub>  |                      | Sc <sub>3</sub>  |                  | Rs3 <sub>5</sub> |                     | Rs2 <sub>5</sub>   | Rs1 <sub>5</sub> | Funct <sub>5</sub>  |                  | 2Fh <sub>8</sub>    | MSX                 |          |
| ~ <sub>6</sub>  |                      | AR               | Sz <sub>4</sub>  |                  | Funct <sub>5</sub>  | Rs2 <sub>5</sub>   | Rs1 <sub>5</sub> | Rd <sub>5</sub>     |                  | 0Eh <sub>8</sub>    | AMO                 |          |
| ~ <sub>6</sub>  |                      | AR               | Sz <sub>4</sub>  |                  | Funct <sub>5</sub>  | Imm <sub>5</sub>   | Rs1 <sub>6</sub> | Rd <sub>6</sub>     |                  | 0Eh <sub>8</sub>    | AMOI                |          |
| ~ <sub>9</sub>  |                      |                  | Sc <sub>3</sub>  |                  | Rs3 <sub>5</sub>    | DC <sub>3</sub>    | IC <sub>2</sub>  | Rs1 <sub>5</sub>    | 0Eh <sub>5</sub> |                     | CACHEX              |          |
| ~27             |                      |                  |                  |                  |                     |                    |                  | 18h <sub>5</sub>    |                  | 2Fh <sub>8</sub>    | MEMDB               |          |
| ~27             |                      |                  |                  |                  |                     |                    |                  | 19h <sub>5</sub>    |                  | 2Fh <sub>8</sub>    | MEMSB               |          |
| ~ <sub>9</sub>  |                      |                  | Cmd <sub>5</sub> |                  | ~ <sub>4</sub>      |                    | Tn <sub>4</sub>  | Rs1 <sub>5</sub>    | Rd <sub>5</sub>  | 2Dh <sub>8</sub>    | TLB                 |          |
| p               | Imm <sub>31</sub>    |                  |                  |                  |                     |                    |                  |                     |                  | BEh <sub>8</sub>    | PFX                 |          |
| p               | Imm <sub>20..0</sub> |                  |                  |                  |                     |                    | Rs1 <sub>5</sub> | Rd <sub>5</sub>     |                  | Opcode <sub>8</sub> | RI                  |          |
| p               | Imm <sub>20..5</sub> |                  |                  |                  | Rs2 <sub>5</sub>    |                    | Rs1 <sub>5</sub> | Imm <sub>4..0</sub> |                  | 9Ah <sub>8</sub>    | CHKI                |          |
| p               | Imm <sub>20..0</sub> |                  |                  |                  |                     |                    | Rs1 <sub>5</sub> | 2 <sub>2</sub>      | Cr <sub>3</sub>  | Opcode <sub>8</sub> | CMP                 |          |
| M <sub>3</sub>  | z                    | ~ <sub>2</sub>   | r                | Fmt <sub>4</sub> | Funct <sub>6</sub>  | ~ <sub>5</sub>     |                  | Rs1 <sub>5</sub>    | Rd <sub>5</sub>  |                     | 8Bh <sub>8</sub>    | R1       |
| M <sub>3</sub>  | z                    | ~ <sub>2</sub>   | r                | Fmt <sub>4</sub> | Funct <sub>6</sub>  | Rs2 <sub>5</sub>   |                  | Rs1 <sub>5</sub>    | Rd <sub>5</sub>  |                     | 8Ch <sub>8</sub>    | R2 / R2S |
| M <sub>3</sub>  | z                    | S <sub>2</sub>   | r                | Fmt <sub>4</sub> | Funct <sub>6</sub>  | Rs2 <sub>5</sub>   |                  | Rs1 <sub>5</sub>    | Rd <sub>5</sub>  |                     | 8Ch <sub>8</sub>    | PTRDIF   |
| M <sub>3</sub>  | z                    | ~ <sub>2</sub>   | r                | Fmt <sub>4</sub> | Funct <sub>6</sub>  | Rs2 <sub>5</sub>   |                  | Rs1 <sub>5</sub>    | 2 <sub>2</sub>   | Cr <sub>3</sub>     | 8Ch <sub>8</sub>    | CMP      |
| M <sub>3</sub>  | z                    | ~ <sub>2</sub>   | r                | Fmt <sub>4</sub> | Funct <sub>6</sub>  | Rs2 <sub>5</sub>   |                  | Rs1 <sub>5</sub>    | 1 <sub>2</sub>   | M <sub>3</sub>      | 8Ch <sub>8</sub>    | Sec      |
| M <sub>3</sub>  | z                    | ~ <sub>2</sub>   | r                | Fmt <sub>4</sub> | ~                   | Rs3 <sub>5</sub>   | Rs2 <sub>5</sub> | Rs1 <sub>5</sub>    | Rd <sub>5</sub>  |                     | Opcode <sub>8</sub> | R3       |
| M <sub>3</sub>  | z                    | ~ <sub>2</sub>   | r                | Fmt <sub>4</sub> | 3Eh <sub>6</sub>    |                    | Rs2 <sub>5</sub> | Rs1 <sub>5</sub>    | Rd <sub>5</sub>  |                     | 8Ch <sub>8</sub>    | BMM      |
| M <sub>3</sub>  | z                    | ~ <sub>2</sub>   | r                | Fmt <sub>4</sub> | Funct <sub>6</sub>  | Rs2 <sub>5</sub>   |                  | Rs1 <sub>5</sub>    | Rd <sub>5</sub>  |                     | 8Ch <sub>8</sub>    | SHIFT    |
| M <sub>3</sub>  | z                    | I <sub>2</sub>   | r                | Fmt <sub>4</sub> | Funct <sub>6</sub>  | Imm <sub>5</sub>   |                  | Rs1 <sub>5</sub>    | Rd <sub>5</sub>  |                     | 8Ch <sub>8</sub>    | SHIFTI   |
| Op <sub>3</sub> | OL <sub>2</sub>      | ~ <sub>5</sub>   |                  |                  | Regno <sub>12</sub> |                    |                  | Rs1 <sub>5</sub>    | Rd <sub>5</sub>  |                     | A5h <sub>8</sub>    | CSR      |
| Op <sub>3</sub> | OL <sub>2</sub>      | Imm <sub>5</sub> |                  |                  | Regno <sub>12</sub> |                    |                  | Imm <sub>5</sub>    | Rd <sub>5</sub>  |                     | 95h <sub>8</sub>    | CSRI     |

|                               |              |          |         |          |        |        |       |      |         |         |      |         |       |      |
|-------------------------------|--------------|----------|---------|----------|--------|--------|-------|------|---------|---------|------|---------|-------|------|
| ~32                           |              |          |         |          |        |        |       |      |         | D2h8    | NOP  |         |       |      |
| P2                            | Address26..0 |          |         |          |        |        |       |      | Cr3     | Opcode8 | Jcc  |         |       |      |
| Imm18..0                      |              |          |         |          | L3     | Rs15   | 02    | L3   | Opcode8 | JRL     |      |         |       |      |
| Address28..0                  |              |          |         |          |        |        |       | L3   | D0h8    | JSR     |      |         |       |      |
| Address28..0                  |              |          |         |          |        |        |       | 03   | D0h8    | JMP     |      |         |       |      |
| p                             | Imm20..4     |          |         |          | ~      | L3     | 315   | 315  | D1h8    | RTS     |      |         |       |      |
| H                             | 03           | Cause8   |         | ~6       | Imask4 |        | Rs15  | ~5   | DCh8    | BRK     |      |         |       |      |
| 0                             | 13           | 2558     |         | ~6       | 04     |        | 05    | ~5   | DCh8    | PFI     |      |         |       |      |
| 04                            |              | ~        | Sema7   |          | ~10    |        | Rs15  | ~5   | DDh8    | RTI     |      |         |       |      |
| 14                            |              | PrivLvl8 |         | ~4       | T2     | Imask4 | Rs15  | ~5   | DDh8    | REX     |      |         |       |      |
| 24                            |              | ~2       | ~6      |          | ~6     | ~4     | ~5    | ~5   | DDh8    | SYNC    |      |         |       |      |
| 34                            |              | ~8       |         | ~6       |        | Imask4 | Rs15  | Rd5  | DDh8    | SEI     |      |         |       |      |
| 44                            |              | ~7       |         | Imm6     |        | Rs25   | Rs15  | ~5   | DDh8    | WAIT    |      |         |       |      |
| 54                            |              | ~7       |         | Codebuf6 |        | ~5     | ~5    | ~5   | DDh8    | EXEC    |      |         |       |      |
| 64                            |              | ~7       |         | ~6       |        | 02     | L3    | Rs15 | 02      | L3      | DDh8 | MTL     |       |      |
| 74                            |              | ~7       |         | ~6       |        | 02     | L3    | ~5   | Rd5     |         | DDh8 | MFL     |       |      |
| Ah4                           |              | Mask8    |         |          | ~5     | ~5     |       | Rs15 | ~5      | DDh8    | MTCR |         |       |      |
| Bh4                           |              | ~8       |         |          | ~5     | ~5     |       | ~5   | Rd5     | DDh8    | MFCR |         |       |      |
| 84                            |              | ~4       | CRFunc4 | ~2       | Cbs26  |        | Cbs16 |      | Cbd6    |         | DDh8 | CRLOG   |       |      |
| M3                            | Rm3          | r        | Fmt4    | z        | FRs35  |        | FRs25 |      | FRs15   |         | FRd5 | Opcode8 | FLT3  |      |
| M3                            | Rm3          | r        | Fmt4    | z        | Funct5 |        | FRs25 |      | FRs15   |         | FRd5 | E2h8    | FLT2  |      |
| M3                            | Rm3          | r        | Fmt4    | z        | Funct5 |        | FRs25 |      | FRs15   |         | 22   | Cr3     | E2h8  | FCMP |
| M3                            | Rm3          | r        | Fmt4    | z        | Funct5 |        | Cndx5 |      | FRs15   |         | FRd5 | E3h8    | FLT2I |      |
| M3                            | Rm3          | r        | Fmt4    | z        | Funct5 |        | ~5    |      | FRs15   |         | FRd5 | E1h8    | FLT1  |      |
| M3                            | Rm3          | r        | Fmt4    | z        | Funct5 |        | ~5    |      | Rs15    |         | FRd5 | E1h8    | I2F   |      |
| M3                            | Rm3          | r        | Fmt4    | z        | Funct5 |        | ~5    |      | FRs15   |         | Rd5  | E1h8    | F2I   |      |
| VMAND / VMOR / VMXOR / VMXNOR |              |          |         |          |        |        |       |      |         |         |      |         |       |      |
| ~4                            | Imm8         |          |         | Funct5   |        | 12     | M23   | 12   | M13     | 12      | M3   | BCh8    | MOP   |      |
| VMFIRST / VMLAST / VMPOP      |              |          |         |          |        |        |       |      |         |         |      |         |       |      |
| ~6                            |              | ~3       | ~5      | Funct5   |        | ~5     |       | 12   | M3      | Rd5     |      | BCh8    |       |      |
| ~6                            |              | ~3       | ~5      | Funct5   |        | ~5     |       | 12   | M3      | Rd5     |      | BCh8    | MFM   |      |
| ~6                            |              | ~3       | ~5      | Funct5   |        | ~5     |       | Rs15 |         | 12      | M3   | BCh8    | MFILL |      |
| ~6                            |              | ~3       | ~5      | Funct5   |        | ~5     |       | Rs15 |         | 12      | M3   | BCh8    | MTM   |      |







## Instruction Encodings

### Indexed Memory Loads {MLX} {LXFunc5}

| Funct <sub>5</sub> | xx000 | xx001 | xx010 | xx011 | xx100 | xx101  | xx110  | xx111  |
|--------------------|-------|-------|-------|-------|-------|--------|--------|--------|
| 00xxx              | LDBX  | LDWX  | LDTX  | LDOX  | LDHX  | LDHRX  | LDPX   |        |
| 01xxx              | LDBUX | LDWUX | LDTUX | LDOUX |       |        | LDPUX  |        |
| 10xxx              |       | LDFHX | LDFSX | LDFDX | LDFQX | LDFQ2X | LDFQ4X | LDFQ8X |
| 11xxx              |       |       |       |       |       |        |        |        |

### AMO {Func5}

| Funct <sub>5</sub> | xx000 | xx001 | xx010 | xx011 | xx100 | xx101 | xx110 | xx111 |
|--------------------|-------|-------|-------|-------|-------|-------|-------|-------|
| 00xxx              | SWAP  | SWAPI | ADD   | ADDI  | AND   | ANDI  | OR    | ORI   |
| 01xxx              | XOR   | XORI  | SHL   | SHLI  | SHR   | SHRI  | MIN   | MINI  |
| 10xxx              | MAX   | MAXI  | MINU  | MINUI | MAXU  | MAXUI |       |       |
| 11xxx              |       |       |       |       |       |       |       |       |

### Indexed Memory Stores / Miscellaneous {SXFunct5}

| Funct <sub>5</sub> | xx000 | xx001 | xx010 | xx011 | xx100 | xx101  | xx110  | xx111  |
|--------------------|-------|-------|-------|-------|-------|--------|--------|--------|
| 00xxx              | STBX  | STWX  | STTX  | STOX  | STHX  | STHCX  | STPX   |        |
| 01xxx              | CASX  |       |       |       |       |        | CACHE  |        |
| 10xxx              |       | STFHX | STFSX | STFDX | STFQX | STFQ2X | STFQ4X | STFQ8X |
| 11xxx              | MEMSB | MEMDB |       |       |       |        |        |        |

## Flow Control

### Branch Predictor Bits

| P <sub>2</sub> | Prediction                           |
|----------------|--------------------------------------|
| 0              | No prediction – use branch predictor |
| 1              | reserved                             |
| 2              | predict as not taken                 |
| 3              | predict as taken                     |

### Condition Register Logic Functions (CRFunc<sub>4</sub>)

|    | x0  | x1  | x2 | x3 | x4 | x5 | x6 | x7 | x8  | x9 | xA  | xB   | xC   | xD  | xE   | xF  |
|----|-----|-----|----|----|----|----|----|----|-----|----|-----|------|------|-----|------|-----|
| 0x | CLR | SET |    |    |    |    |    |    | AND | OR | EOR | ANDC | NAND | NOR | ENOR | ORC |

## Floating Point

### Dyadic Ops – {FLT2, FLT2I} Funct<sub>5</sub>

|    | x0     | x1     | x2   | x3   | x4   | x5   | x6   | x7    | x8     | x9     | xA     | xB    | xC     | xD      | xE | xF |
|----|--------|--------|------|------|------|------|------|-------|--------|--------|--------|-------|--------|---------|----|----|
| 0x | SCALEB | {FLT1} | FMAX | FMIN | FADD | FUB  | FCMP | FCMPM | FMUL   | FDIV   | FREM   | FNXT  | FAND   | FOR     |    |    |
| 1x | FSLT   | FSGE   | FSLE | SFGT | FSEQ | FSNE | FSUN |       | CPYSGN | SGNINV | SGNAND | SGNOR | SGNXOR | SGNXNOR |    |    |
| 2x |        |        |      |      |      |      |      |       |        |        |        |       |        |         |    |    |
| 3x |        |        |      |      |      |      |      |       |        |        |        |       |        |         |    |    |

### Monadic Ops – {FLT1} Funct<sub>6</sub>

|    | x0   | x1   | x2     | x3   | x4  | x5    | x6    | x7   | x8 | x9 | xA   | xB   | xC    | xD    | xE     | xF     |
|----|------|------|--------|------|-----|-------|-------|------|----|----|------|------|-------|-------|--------|--------|
| 0x | FMOV |      | FTOI   | ITOF |     |       | FSIGN | FMAN |    |    | FS2Q | FD2Q | FSTAT | FSQRT | ISNAN  | FINITE |
| 1x | FTX  | FCX  | FEX    | FDX  | FRM | TRUNC | FSYNC |      |    |    | FQ2S | FQ2D |       |       | FCLASS | UNORD  |
| 2x |      |      |        |      |     |       |       |      |    |    |      |      |       |       |        |        |
| 3x | FRES | FSIG | FRSQRT |      |     |       |       |      |    |    |      |      |       |       |        |        |

### Constants ROM – VRs<sub>2</sub>

|    | x0   | x1   | x2 | x3 | x4                 | x5                  | x6                  | x7                 | x8 | x9 | xA | xB | xC | xD    | xE    | xF     |
|----|------|------|----|----|--------------------|---------------------|---------------------|--------------------|----|----|----|----|----|-------|-------|--------|
| 0x | +0.0 | +1.0 | e  | PI | log <sub>e</sub> e | log <sub>2</sub> 10 | log <sub>10</sub> 2 | log <sub>e</sub> 2 |    |    |    |    |    |       |       |        |
| 1x |      |      |    |    |                    |                     |                     |                    |    |    |    |    |    |       |       |        |
| 2x |      |      |    |    |                    |                     |                     |                    |    |    |    |    |    |       |       |        |
| 3x |      |      |    |    |                    |                     |                     |                    |    |    |    |    |    | Imm32 | Imm64 | Imm128 |

| Unit |                |  |
|------|----------------|--|
| 0    |                |  |
| 1    | Branch         |  |
| 2    | Integer        |  |
| 3    | Floating Point |  |
| 4    | Memory         |  |
| 5    |                |  |
| 6    |                |  |
| 7    |                |  |

[illegible]

|    | x0    | x1    | x2   | x3   | x4   | x5   | x6   | x7   | x8    | x9    | xA     | xB     | xC    | xD    | xE     | xF     |
|----|-------|-------|------|------|------|------|------|------|-------|-------|--------|--------|-------|-------|--------|--------|
| 0x | MULH  | MULUH | ADDV | SUBV | ADD  | SUB  | CMP  | CMPU | AND   | OR    | XOR    |        | NAND  | NOR   | XNOR   | MULF   |
| 1x | SLT   | SGE   | SLE  | SGT  | SLTU | SGEU | SLEU | SGTU | SEQ   | SNE   |        |        |       |       |        |        |
| 2x | MUL   | MULU  | DIV  | DIVU | MOD  | MODU |      |      | MADF  |       | BYTNDX | WYDNDX | DIF   | AVG   | PTRDIF | PTRDIF |
| 3x | FXMUL | FXDIV | SHL  | ASL  | SHR  | ASR  | ROL  | ROR  | SHL # | ASL # | SHR #  | ASR #  | ROL # | ROR # | BMM    | LEAX   |

[illegible]

## Instruction Descriptions

**Execution Units:** this is the functional unit that the instruction is executed by.

**Clock cycles:**

This is a highly optimistic estimate of the number of clock cycles required to complete the operation. This estimate assumes all values required for the operation are available without delay. Under ideal circumstances many operations can complete in 1/3 of a clock cycle because there are enough functional units to service that many instructions at once.

### Branch Unit

The branch unit executes instructions that control the flow of program execution. These instructions include things like SEI which enables interrupts in addition to regular branch, jump and call instructions.

### Branch Targets

Branch targets are absolute addresses which modify the lower 30 bits of the instruction pointer. This gives the branch a 1GB range.

### Branch Predictions

Normally the branch predictor determines the branch prediction, however, a branch instruction may be statically predicted to be taken or not taken. This is controlled by the P<sub>2</sub> field of the branch instruction.

## BRK – Hardware / Software Breakpoint

### Description:

Invoke the break handler routine. The break handler routine handles all the hardware and software exceptions in the core. A cause code is loaded into the CAUSE CSR register. The break handler should read the CAUSE code to determine what to do. The break handler is located by TVEC[0]. This address should contain a jump to the break handler. Note the reset address is \$F[...]FFC0100. An exception will automatically switch the processor to the machine level operating mode. The break handler routine may redirect the exception to a lower level using the [REX](#) instruction.

The core maintains an internal eight level interrupt stack for each of the following:

| Item Stacked        | CSR reg  |                           |
|---------------------|----------|---------------------------|
| instruction pointer | ip_stack |                           |
| operating level     | ol_stack | available as a single CSR |
| privilege level     | pl_stack | available as a single CSR |
| interrupt mask      | im_stack | available as a single CSR |

If further nesting of interrupts is required, the stacks may be copied to memory as they are available from CSR's.

On stack underflow a break exception is triggered.

Hardware interrupts will cause a BRK instruction to be inserted into the instruction stream.

Because instructions are fetched in bundles a hardware interrupt always intercepts at a bundle address, and always returns to a bundle address.

### Instruction Format: BRK

H = 1 = software interrupt – return address is next instruction

H = 0 = hardware interrupt – return address is current instruction

IMask<sub>4</sub> = the priority level of the hardware interrupt, the priority level at time of interrupt is recorded in the instruction, the interrupt mask will be set to this level when the instruction commits. This field is not used for software interrupts and should be zero.

Cause Code = numeric code associated with the cause of the interrupt. The cause code is bitwise 'or'd with the value in register Rs1 to set the the cause CSR. Usually either one of the cause code field or Rs1 will be zero.

The empty instruction fields may be used to pass constant data to the break handler.

| Operating Level | Address (If TVEC[0] contains \$FFFC0000) |                                  |
|-----------------|--|----------------------------------|
| 0               | \$FFFC0000                               | Handler for operating level zero |
| 1               | \$FFFC0020                               |                                  |
| 2               | \$FFFC0040                               |                                  |
| 3               | \$FFFC0060                               |                                  |



## CRAND – Condition Register Bit And

### Description:

Perform an ‘and’ operation between two bits (Cbs1 and Cbs2) of the condition register and place the result in condition register bit Cbd.

**Instruction Format:** CRLOG

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none

## CRANDC – Condition Register Bit And Complement

### Description:

Perform an ‘and complement’ operation between two bits (Cbs1 and Cbs2) of the condition register and place the result in condition register bit Cbd.

**Instruction Format:** CRLOG

### Operation:

$$Cr_{[bit\ Cbd]} = Cr_{[bit\ Cbs1]} \& \sim Cr_{[bit\ Cbs2]}$$

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none

## CRCLR – Condition Register Clear Bit

### Description:

This is an extended mnemonic for the CREOR instruction where the same bit is specified for Cbs1 and Cbs2 fields of the instruction.

**Instruction Format:** CRLOG

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none

## CRENOR – Condition Register Bit Exclusive Nor

**Description:**

Perform an ‘exclusive or’ operation between two bits (Cbs1 and Cbs2) of the condition register and place the inverted result in condition register bit Cbd.

**Instruction Format:** CRLOG**Clock Cycles:** 0.25**Execution Units:** Branch**Exceptions:** none

## CREOR – Condition Register Bit Exclusive Or

**Description:**

Perform an ‘exclusive or’ operation between two bits (Cbs1 and Cbs2) of the condition register and place the result in condition register bit Cbd.

**Instruction Format:** CRLOG**Clock Cycles:** 0.25**Execution Units:** Branch**Exceptions:** none

## CRNAND – Condition Register Bit Nand

### Description:

Perform an ‘and’ operation between two bits (Cbs1 and Cbs2) of the condition register and place the inverted result in condition register bit Cbd.

**Instruction Format:** CRLOG

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none

## CRNOR – Condition Register Bit Nor

### Description:

Perform an ‘or’ operation between two bits (Cbs1 and Cbs2) of the condition register and place the inverted result in condition register bit Cbd.

**Instruction Format:** CRLOG

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none

## CROR – Condition Register Bit Or

### Description:

Perform an ‘or’ operation between two bits (Cbs1 and Cbs2) of the condition register and place the result in condition register bit Cbd.

**Instruction Format:** CRLOG

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none

## CRORC – Condition Register Bit Or Complement

### Description:

Perform an ‘or complement’ operation between two bits (Cbs1 and Cbs2) of the condition register and place the result in condition register bit Cbd.

### Instruction Format: CRLOG

### Operation:

$$Cr_{[bit\ Cbd]} = Cr_{[bit\ Cbs1]} \mid \sim Cr_{[bit\ Cbs2]}$$

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none

## CRSET – Condition Register Set Bit

**Description:**

This is an extended mnemonic for the CRENOR instruction where the same bit is specified for Cbs1 and Cbs2 fields of the instruction.

**Instruction Format:** CRLOG**Clock Cycles:** 0.25**Execution Units:** Branch**Exceptions:** none

## EXEC – Execute Code Buffer

### Description:

Execute code from code buffer. The Codebuf<sub>6</sub> field specifies the code buffer to use. Code buffers allow code to be adapted at run-time. This is useful as an alternative to self-modifying code when code must change at runtime. Some of the benefits of using a code buffer are the cache doesn't have to be invalidated, they are fast and easy to use.

### Instruction Format: EXEC

### Execution Units: Branch

**Clock Cycles:** Minimum 0.25 – depends on the instruction in the code buffer

**Exceptions:** As per instruction in code buffer.

### Note:

The EXEC instruction intercepts the instruction stream at the fetch phase and inserts the instruction from the code buffer rather than the cache output.

## Jcc – Conditional Jump

### Description:

If the condition is true, the target address is loaded into the instruction pointer. If the instruction branches back to itself, a branch target exception will occur. The range of the branch is approximately 640MB (128M instructions). The jump instruction modifies only the low order 27 bits of the instruction pointer.

### Instruction Format: Jcc

| P <sub>2</sub> | Static Prediction                    |
|----------------|--------------------------------------|
| 0              | no prediction (use branch predictor) |
| 1              | {reserved}                           |
| 2              | predict not-taken                    |
| 3              | predict taken                        |

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target



## JCC –Jump if Carry Clear

### Description:

If the condition register carry flag is not set, a target address is loaded into the instruction pointer.  
If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (!Cr.cf)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JCS –Jump if Carry Set

### Description:

If the condition register carry flag is set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (Cr.cf)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JEQ –Jump if Equal

### Description:

If the condition register equals flags is set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (Cr.eq)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JGE –Jump if Greater or Equal

### Description:

If the condition register less than flag is not set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (!Cr.lt)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JGT –Jump if Greater Than

### Description:

If the condition register greater than flag is set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (Cr.gt)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JLE –Jump if Less Than or Equal

### Description:

If the condition register greater than flag is not set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (!Cr.gt)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JLT –Jump if Less Than

### Description:

If the condition register less than flag is set, a target address is loaded into the instruction pointer. The values are treated as signed numbers. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (Cr.lt)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JNE –Jump if Not Equal

### Description:

If the condition register equals flag is not set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (!Cr.eq)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target



## JRL – Jump to Register and Link

### Description:

This instruction allows the target address to be supplied by a register, accommodating computed target addresses. Either a link register or an integer register may be specified. All bits of the instruction pointer are affected. If the instruction branches back to itself, a branch target exception will occur. The address of the next instruction is stored in the target link register. The target link register is assumed to be Lk1 if not specified.

The target address is the sum of the specified link register and integer register Rs1. One of the registers may be zero.

**Instruction Format:** JRL

**Clock Cycles:**

**Execution Units:** Branch

**Exceptions:** branch target

## JSR – Jump to Subroutine

### Description:

This instruction loads the instruction pointer with a constant value specified in the instruction. In addition, the address of the instruction following the JSR is stored in a specified link register. This instruction may be used to implement subroutine calls or method invocations. The target return address register is assumed to be Lk1 by the instruction. In order to use a different register, it must be specified.

### Instruction Format: JSR

This format has an 512M Instr. range. (2.5GB). The format modifies only instruction pointer bits 0 to 28. The high order IP bits are not affected. Note that with the use of a mmu this address range is often sufficient.

If an address range greater than 29 bits is required, then the address must be loaded into a register and a jump to register (JRL) instruction must be used.

### Execution Units: Branch Unit

### Clock Cycles: 0.25

### Notes:

Ideally the call instruction is placed in slot3 and targets an address in slot0. Many routines can be aligned on a bundle address. Since there are four instructions in a bundle the average extra space consumed by aligning routines on a bundle address is only 2 instruction words.

The jsr instruction executes immediately during the fetch stage of the core. This makes it much faster than other options.

## JMP – Jump to Address

### Description:

A jump is made to the address specified in the instruction.

### Instruction Format: JSR

This instruction is an alternate mnemonic for the JSR instruction where the return address register is assumed to be Lk0. The format modifies only instruction pointer bits 0 to 28. The high order IP bits are not affected. This allows accessing code within a 512M Instr. range. (2.5GB) region of memory. Note that with the use of a mmu this address range is often sufficient.

### Execution Units: Branch

### Clock Cycles: 0.25

### Exceptions: Branch target

### Notes:

If an address range larger than 29 bits is required, then the value must be loaded into a register and a branch instruction used.

The jump instruction executes immediately during the fetch stage of the core. This makes it much faster than a branch.

JMP should be placed in slot3 for best performance.

## JUC –Jump if User flag Clear

### Description:

If the condition register user flag is not set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (!Cr.uf)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JUS –Jump if User flag Set

### Description:

If the condition register user flag is set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

if (Cr.uf)  
ip = target

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JVC –Jump if Overflow Clear

### Description:

If the condition register overflow flag is not set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

```
if (!Cr.vf)
    ip = target
```

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## JVS –Jump if Overflow Set

### Description:

If the condition register overflow flag is set, a target address is loaded into the instruction pointer. If the branch branches back to itself a branch exception will be generated. The branch range is approximately 640MB (128M instructions).

### Instruction Format: Jcc

### Operation:

```
if (Cr.vf)
    ip = target
```

**Clock Cycles:** 0.25 (Ideal) 1 (average)

**Execution Units:** Branch

**Exceptions:** branch target

## MFCR – Move From Condition Registers

**Description:**

Move all the condition registers as a single group to the specified general-purpose register. Cr0 is placed in the low order eight bits of the target register. Cr1 is placed in the next eight bits (bits 8 to 15) and so on up to Cr7. The high order 64 bits of the target register are set to zero.

**Instruction Format:** MFCR**Clock Cycles:** 0.25**Execution Units:** Branch**Exceptions:** none

## MFL – Move From Link Register

**Description:**

Move from a link register to the specified general-purpose register.

**Instruction Format:** MFL**Clock Cycles:** 0.25**Execution Units:** Branch**Exceptions:** none

## MTCR – Move To Condition Registers

### Description:

Move a general-purpose register into all the condition registers under the restriction of an immediate mask. Cr0 is loaded from the low order eight bits of the source register, if mask bit zero is set. Cr1 is loaded from the next eight bits (bits 8 to 15) if mask bit one is set, and so on up to Cr7.

**Instruction Format:** MTCR

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none

## MTL – Move To Link Register

### Description:

This instruction allows moving a link register to a link register, or a general-purpose register to a link register. The source is the sum of the specified source link register and source general purpose register. One or the other of these registers should be zero.

**Instruction Format:** MTL

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none

## NOP – No Operation

### Description:

The NOP instruction doesn't perform any operation. NOP's are detected in the instruction fetch stage of the core and are not enqueued by the core. They do not occupy queue slots. Because NOPs don't occupy queue slots they may not be used to synchronize operations between instructions.

**Instruction Format:** NOP

**Clock Cycles:** 0.25

**Execution Units:** Branch

**Exceptions:** none



## PFI – Poll for Interrupt

### Description:

The poll for interrupt instruction polls the interrupt status lines and performs an interrupt service if an interrupt is present. Otherwise the PFI instruction is treated as a NOP operation. Polling for interrupts is performed by managed code. PFI provides a means to process interrupts at specific points in running software.

Note that the core records the address including the slot number of the PFI instruction as the exception address. This may be in the middle of an instruction bundle. In the case where an exception is present instructions in the bundle following the PFI instruction are not executed. After completing the exception service routine, the core will return to the instruction following the PFI instruction, this may be in the middle of an instruction bundle.

### Instruction Format: PFI

**Clock Cycles:** 0.25 (if no exception is present)

**Execution Units:** Branch

**Exceptions:** any outstanding hardware interrupts

## RTS – Return from Subroutine

**Description:**

This instruction performs a subroutine return by loading the instruction pointer with the contents of a specified return address register (Lk0 to Lk7). Additionally, an amount may be added to the stack pointer.

**Instruction Format:** RET

**Clock Cycles:** 1 (more if predicted incorrectly).

**Execution Units:** Branch

**Exceptions:** none

**Notes:**

The RTS instruction is detected and used at the fetch stage of the processor to update the RSB.

## REX – Redirect Exception

### Description:

This instruction redirects an exception from an operating level to a lower operating level and privilege level. If the target operating level is hypervisor then the hypervisor privilege level (1) is set. If the target operating level is supervisor, then one of the supervisor privilege levels must be chosen (2 to 6). 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.

When redirecting the target privilege level is set to the bitwise 'or' of an immediate constant specified in the instruction and register Rs1. One of these two values should be zero. The result should be a value in the range 2 to 255. The instruction will not allow setting the privilege level numerically less than the operating level.

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

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

The REX instruction also specifies the interrupt mask level to set for further processing.

Attempting to redirect the operating level to the machine level (0) will be ignored. The instruction will be treated as a NOP with the exception of setting the interrupt mask register.

### Instruction Format: REX

| Tgt2 |                              |
|------|------------------------------|
| 0    | not used                     |
| 1    | redirect to hypervisor level |
| 2    | redirect to supervisor level |
| 3    | not used                     |

**Clock Cycles:** 4

**Execution Units:** Branch

Example:

```

REX 5,12,r0    ; redirect to supervisor handler, privilege level two
; If the redirection failed, exceptions were likely disabled at the target level.
; Continue processing so the target level may complete it's operation.
RTI            ; redirection failed (exceptions disabled ?)
  
```

### Notes:

Since all exceptions are initially handled at the machine level the machine level handler must check for disabled lower level exceptions.

## RTI – Return from Interrupt

### Description:

Return from an interrupt or exception processing subroutine. The interrupted instruction pointer (Lk7) is loaded into the instruction pointer register. The internal interrupt stack is popped and the operating level, privilege level, interrupt mask level, and register set are reset to values before the exception occurred. Optionally a semaphore bit in the semaphore register is cleared. The least significant bit of the semaphore register (the reservation status bit) is always cleared by this instruction.

### Instruction Format: RTI

$\text{Semaphore}[\text{Sema}_6[\text{Rs1}]] = 0$   
 $\text{Semaphore}[0] = 0$

**Clock Cycles:** 8 minimum

**Execution Units:** Branch

## SEI – Set Interrupt Mask

SEI #3

SEI \$v0,#7

### Description:

The interrupt level mask is set to the value specified by the instruction. The value used is the bitwise or of the contents of register Rs1 and an immediate (M<sub>4</sub>) supplied in the instruction. The assembler assumes a mask value of fifteen, masking all interrupts, if no mask value is specified. Usually either M<sub>4</sub> or Rs1 should be zero. The previous setting of the interrupt mask is stored in Rd.

### Instruction Format: SEI

### Operation:

Rd = im

im = M<sub>4</sub> | Rs1

# SYNC -Synchronize

**Description:**

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

**Instruction Format:** SYNC

**Clock Cycles:** 1 \*varies depending on queue contents

**Execution Units:** Branch

**Exceptions:** none

**Notes:**

This instruction may be used with CSR register access as the core does not provide bypassing on the CSR registers. Issuing a sync instruction before reading a CSR will ensure that any outstanding updates to the CSR will be completed before the read.

## Integer Unit

## Integer Unit

This unit handles many different types of instructions associated with integer processing.

## Nomenclature

A dot after the instruction mnemonic causes an associated condition register update. `ADD.r1,r2,r3` will perform the addition then update `cr1`. Many integer and floating-point instructions may update condition registers. Those that can are indicated by `[]` after the instruction mnemonic.

## Constant Prefixes

Constant prefixes allow the use of immediate values larger than what can be encoded in a single instruction word. Instructions using a 21-bit constant may indicate that a prefix is present by having the most significant bit of the instruction set. Each constant prefix supplies 31 extra bits to the constant. (A constant prefix may itself have a prefix). The most significant bits of the constant are represented by the first prefix. Subsequent prefixes supply subsequently less significant bits of the constant. The constant prefix is sign extended to the machine width of 128 bits.



## ABS[.] – Absolute Value

**Description:**

This instruction takes the absolute value of a register and places the result in a target register. This instruction may optionally update Cr0 with result flags.

**Instruction Format:** Integer R1**Clock Cycles:** 0.25**Execution Units:** Integer ALU**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv**Operation:**

```
If Rs1 < 0
    Rd = -Rs1
else
    Rd = Rs1
```

**Exceptions:** none

## ADD[.] - Addition

### Description:

Add two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. This instruction may optionally update Cr0 with result flags.

**Instruction Formats:** Integer RI, R2, R2S

**Clock Cycles:** 0.25

**Execution Units:** All ALU's

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

### Scalar Operation

$$Rd = Rs1 + Rs2$$

OR

$$Rd = Rs1 + \text{Immediate}$$

### Vector Operation

for  $x = 0$  to  $VL - 1$

$$\text{if } (Vm[x]) \text{ } Rd[x] = Rs1[x] + Rs2[x]$$

$$\text{else if (zero) } Rd[x] = 0$$

$$\text{else } Rd[x] = Rd[x]$$

### Vector - Scalar Operation (R2S)

for  $x = 0$  to  $VL - 1$

$$\text{if } (Vm[x]) \text{ } Rd[x] = Rs1[x] + Rs2[0]$$

$$\text{else if (zero) } Rd[x] = 0$$

$$\text{else } Rd[x] = Rd[x]$$

**Exceptions:** none

**Notes:**

## AND[.] – Bitwise And

### Description:

Perform a bitwise ‘and’ operation between operands. For the immediate form, the immediate constant is one extended to the left before use. This instruction may optionally update Cr0 with result flags.

**Instruction Format:** RI, R2, R2S

**Clock Cycles:** 0.25

**Execution Units:** Integer

### Scalar Operation

$Rd = Rs1 \ \& \ Rs2$

OR

$Rd = Rs1 \ \& \ \text{Immediate}$

### Vector Operation

for  $x = 0$  to  $VL - 1$

if ( $Vm[x]$ )  $Rd[x] = Rs1[x] \ \& \ Rs2[x]$

else if (zero)  $Rd[x] = 0$

else  $Rd[x] = Rd[x]$

### Vector - Scalar Operation (R2S)

for  $x = 0$  to  $VL - 1$

if ( $Vm[x]$ )  $Rd[x] = Rs1[x] \ \& \ Rs2[0]$

else if (zero)  $Rd[x] = 0$

else  $Rd[x] = Rd[x]$

**Exceptions:** none

## ASL[.] – Arithmetic Shift Left

### Description:

Bits from the source register Rs1 are shifted left by the amount in register Rs2 or an immediate value. A zero is shifted into bit zero. The difference between this instruction and a SHL instruction is that ASL may cause an arithmetic overflow exception. SHL will never cause an exception.

**Instruction Format:** SHIFT, SHIFTI

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

### Exceptions:

An overflow exception may result if the bits shifted out from the MSB are not the same as the resulting sign bit and the exception is enabled in the AEC register. Exceptions are only caused by a word size operation.

## ASR[.] – Arithmetic Shift Right

**Description:**

Bits from the source register Rs1 are shifted right by the amount in register Rs2 or an immediate value. The sign bit is shifted into the most significant bits preserving the sign of the value.

**Instruction Formats:** SHIFT, SHIFTI

**Supported Formats:** .b .w, .t, .o, .h, .bv, .vv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## AVG[.] – Average Value

**Description:**

This instruction averages two signed values in registers Rs1 and Rs2 and places the result in a target Rd register. The average is calculated as the sum of Rs1 and Rs2 plus one for rounding, then arithmetically shift right by one.

**Instruction Format:** R1**Clock Cycles:** 0.25**Execution Units:** Integer**Operation:**

$$Rd = (Rs1 + Rs2 + 1) / 2$$

**Exceptions:** none

## BFCHG – Bitfield Change

### Description:

A bitfield in the source specified by Da is inverted, the result is copied to the target register. Bo specifies the bit offset. Bw specifies the bit width. The bit width and offset may either be contained in a register or an immediate value.

### Instruction Format: BF

| Rg <sub>3</sub> Bit |  |
|---------------------|--|
| 0                   | 1 = Bo is a register spec, 0 = Bo is a six bit immediate |
| 1                   | 1 = Bw is a register spec, 0 = Bw is a six bit immediate |
| 2                   | 1  |

**Clock Cycles:** 0.33

**Execution Units:** Integer

**Exceptions:** none

## BFCLR – Bitfield Clear

### Description:

A bitfield is cleared in the target register. All bits are copied from a source Da (which is zero extended if an immediate value) except for bits identified by the bitfield which are set to zero in the target.

### Instruction Format: BF

| Rg <sub>3</sub> Bit |   |
|---------------------|---|
| 0                   | 1 = Bo is a register spec, 0 = Bo is a six bit immediate      |
| 1                   | 1 = Bw is a register spec, 0 = Bw is a six bit immediate      |
| 2                   | 1 = Da is a register spec, 0 = Da is an sixteen bit immediate |

**Clock Cycles:** 0.33

**Execution Units:** Integer

**Exceptions:** none

### Notes:

Normally Da is a register which is the same as the target register Rt.



## BFEXT – Bitfield Extract

### Description:

A bitfield is extracted from the source register Da by shifting 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.

### Instruction Format: BF

| Rg <sub>3</sub> Bit |   |
|---------------------|---|
| 0                   | 1 = Bo is a register spec, 0 = Bo is a six bit immediate      |
| 1                   | 1 = Bw is a register spec, 0 = Bw is a six bit immediate      |
| 2                   | 1 = Da is a register spec, 0 = Da is an sixteen bit immediate |

**Clock Cycles:** 0.33

**Execution Units:** Integer

**Exceptions:** none

### Notes:

While it is possible for Da to be a constant the instruction would not normally be used this way.

## BFEXTU – Bitfield Extract

**Description:**

A bitfield is extracted from the source register Da by shifting 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.

**Instruction Format:** BF**Clock Cycles:** 0.33**Execution Units:** Integer**Exceptions:** none

## BFFFO – Bitfield Find First One

### Description:

A bitfield contained in Da is searched beginning at the most significant bit to the least significant bit for a bit that is set. The index into the word of the bit that is set is stored in Rd. If no bits are set, then Rd is set equal to -1. To get the index into the bitfield of the set bit, subtract off the bitfield offset.

### Instruction Format: BF

| Rg <sub>3</sub> Bit |   |
|---------------------|---|
| 0                   | 1 = Bo is a register spec, 0 = Bo is a six bit immediate      |
| 1                   | 1 = Bw is a register spec, 0 = Bw is a six bit immediate      |
| 2                   | 1 = Da is a register spec, 0 = Da is an sixteen bit immediate |

**Clock Cycles:** 0.33

**Execution Units:** Integer

**Exceptions:** none

## BFINSI – Bitfield Insert Immediate

### Description:

A bitfield is inserted into the target register Rd by combining a value read from Rs3 with a constant shifted to the left. The bitfield may not be larger than six bits. To accommodate a larger field multiple instructions can be used, or a value loaded into a register and the BFINS instruction used.

### Instruction Format: Integer BFI

| Rg <sub>2</sub> Bit |  |
|---------------------|--|
| 0                   | 1 = Bo is a register spec, 0 = Bo is a six bit immediate |
| 1                   | 1 = Bw is a register spec, 0 = Bw is a six bit immediate |

Rg<sub>[1]</sub> bit should always be clear for this instruction

**Clock Cycles:** 1

**Execution Units:** ALU #0 Only

**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:**  $Rd = (Rs1 * \alpha) + (Rs2 * \sim\alpha)$

**Clock Cycles:** 4

## BMM – Bit Matrix Multiply

BMM Rd, Rs1, Rs2

### Description:

The BMM instruction treats the bits of register Rs1 and register Rs2 as a 16x16 bit matrix, or as an 8x8 matrix 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. Note that only the 256-bit and 64-bit sizes are supported for this operation.

**Instruction Format:** Integer R2

**Supported Formats:** .ov,

| Func <sub>2</sub> | Function               |
|-------------------|------------------------|
| 0                 | MOR                    |
| 1                 | MXOR                   |
| 2                 | MORT (MOR transpose)   |
| 3                 | MXORT (MXOR transpose) |

### Operation:

for I = 0 to 15

for j = 0 to 15

$$Rt.bit[i][j] = (Ra[i][0] \& Rb[0][j]) \mid (Ra[i][1] \& Rb[1][j]) \mid \dots \mid (Ra[i][15] \& Rb[15][j])$$

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Exceptions:** none

### Notes:

The bits are numbered with bit 255 of a register representing I,j = 0,0 and bit 0 of the register representing I,j = 15,15.

## 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 +15. The index of the first found byte is returned (closest to zero).

**Instruction Format:** R3**Clock Cycles:** 0.25**Execution Units:** Integer**Operation:**
$$Rd = \text{Index of (Rs2 in Rs1)}$$
**Exceptions:** none

## CHK – Check Register Against Bounds

### Description:

A register is compared to two values. If the register is outside of the bounds defined by Rs2 and Rs3 or an immediate value, then an exception will occur. Rs1 must be greater than or equal to Rs2 and Rs1 must be less than Rs3 or the immediate.

**Instruction Format:** CHK, CHKI

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer Unit

**Exceptions:** bounds check

### Notes:

The system exception handler will typically transfer processing back to a local exception handler.



## CMP – Compare

**Description:**

Performs a comparison operation between operands. Operands are treated as signed values. The specified condition register is updated with flags indicating the relationship between the operands. The eq flag is set if operands are equal. The lt flag is set if op1 is less than op2, otherwise the gt flag is set.

**Instruction Format:** RI, R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## CMPU – Unsigned Compare

**Description:**

Performs a comparison operation between operands. Operands are treated as unsigned values. The specified condition register is updated with flags indicating the relationship between the operands. The eq flag is set if operands are equal. The lt flag is set if op1 is less than op2, otherwise the gt flag is set.

**Instruction Format:** RI, R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## 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

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**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

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**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

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**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**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv**Clock Cycles:** 0.25**Execution Units:** Integer ALU**Operation:**

$$Rd = \sim Rs1$$

**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

| Op <sub>3</sub> |        | 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 OL<sub>2</sub> field is reserved to specify the operating level. Note that registers cannot be accessed by a lower operating level.

| Regno <sub>12</sub> |         | Access | Description  |
|---------------------|---------|--------|--|
| 001                 | HARTID  | R      | hardware thread identifier (core number)                     |
| 002                 | TICK    | R      | tick count, counts every cycle from reset                    |
| 030-037             | TVEC    | RW     | trap vector handler address                                  |
| 048                 | EPC     | RW     | exceptioned pc, pc value at point of exception               |
| 044                 | STATUSL | RWSC   | status register, contains interrupt mask, operating level    |
| 045                 | STATUSH | RW     | status register bits 64 to 127                               |
| 080-0BF             | CODE    | RW     | code buffers   |
| FF0                 | INFO    | R      | Manufacturer name  |
| FF1                 | “       | R      | “  |
| FF2                 | “       | R      | cpu class  |
| FF3                 | “       | R      | “  |
| FF4                 | “       | R      | cpu name   |
| FF5                 | “       | R      | “  |
| FF6                 | “       | R      | model number   |
| FF7                 | “       | R      | serial number  |
| FF8                 | “       | R      | cache sizes instruction (bits 40 to 79), data (bits 0 to 39) |

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

**Clock Cycles:** 0.25

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



## 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, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Operation:**

$$Rd = \text{Abs}(Rs1 - Rs2)$$

**Exceptions:** none



## DIV[.] – Signed Division

### Description:

Compute the quotient. The first operand must be in a register. The second operand may be in either a register or an immediate value specified in the instruction. The operands are treated as signed values and the result is a signed result.

**Instruction Format:** RI, R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 132 (n + 4) where n is the width

**Execution Units:** Integer

**Exceptions:** A divide by zero exception may occur if enabled in the AEC register.

## DIVU – Unsigned Division

### Description:

Compute the quotient value. The first operand must be in a register. The second operand may be in either a register or an immediate value specified in the instruction. The operands are treated as unsigned values and the result is an unsigned result.

### Comment:

Unsigned division is often used in calculation of the difference between two pointer values.

**Instruction Format:** RI, R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 132 (n + 4) where n is the width

**Execution Units:** Integer

**Exceptions:** none

## FXADD – Fixed Point Addition

### Description:

Add two values assuming operands are signed fixed point numbers. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. This instruction is an alternate mnemonic for the ADD instruction.

**Instruction Format:** RI, R3

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## FXMUL – Fixed Point Multiplication

### Description:

Multiply two values assuming operands are signed fixed point numbers. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. The value is rounded towards plus infinity. The binary point is at the 40<sup>th</sup> bit position.

**Instruction Format:** RI, R3

**Clock Cycles:** 20

**Execution Units:** Integer

**Exceptions:** none

## FXSUB – Fixed Point Subtraction

### Description:

Subtract two values assuming operands are signed fixed point numbers. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. This instruction is an alternate mnemonic for the SUB/ADD instruction.

**Instruction Format:** RI, R3

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## ISPTR – Is Pointer

**Description:**

This instruction detects if the value in a register is a pointer and places the result in a target condition register. A pointer value is indicated if the top 24 bits of the value are equal to \$FFFF01.

**Instruction Format:**

**Supported Formats:** .h, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Operation:**

```
If Rs1[127:104] = $FFFF01
    Crd.eq = 1
else
    Crd.eq = 0
```

**Exceptions:** none

Notes:

## LEA – Load Effective Address

### Description:

This instruction loads an address value into a register. The address value is calculated in the same manner as a load / store instruction. The data type is set to indicate a pointer value is in the register.

**Instruction Format:** ML, MLX

**Supported Formats:** .h, .hv

**Clock Cycles:** 0.25

| Sc <sub>3</sub> | Scale Rc By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | 5           |

**Execution Units:** Memory Load

**Operation:**

**Exceptions:** none

Notes:

## MAJ[.] – Majority Logic

**Description:**

Determines the majority logic bits of three values in registers Rs1, Rs2, and Rs3 and places the result in the target register Rd.

**Instruction Format:** R3

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

**Operation:**

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

## MAX[.] – Maximum Value

### Description:

Determines the maximum of two values in registers Rs1, Rs2 and places the result in the target register Rd. The values are treated as signed values.

MAX may be used to determine the lowest level privilege level. The lowest privilege level will have the highest value in registers.

**Instruction Format:** R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

### Operation:

```
IF Rs1 > Rs2
    Rd = Rs1
else
    Rd = Rs2
```

**Exceptions:** none

## MIN[.] – Minimum Value

### Description:

Determines the minimum of two values in registers Rs1, Rs2 and places the result in the target register Rd. The values are treated as signed values.

MIN may be used to determine the highest level privilege level. The highest privilege level will have the lowest value in registers.

**Instruction Format:** R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

### Operation:

```
IF Rs1 < Rs2
    Rd = Rs1
else
    Rd = Rs2
```

**Exceptions:** none

## MOD[.] – Signed Modulus

### Description:

Compute the modulus (remainder) value. The first operand must be in a register. The second operand may be in either a register or an immediate value specified in the instruction. The operands are treated as signed values and the result is a signed result.

**Instruction Format:** RI, R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 132 (n + 4) where n is the width

**Execution Units:** Integer

**Exceptions:** A divide by zero exception may occur if enabled in the AEC register.



## MODU – Unsigned Modulus

### Description:

Compute the modulus (remainder) value. Both operands must be in registers. The operands are treated as unsigned values and the result is an unsigned result. There is no condition register update form of this instruction.

**Instruction Format:** R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:**  $132(n + 4)$  where n is the width

**Execution Units:** Integer

**Exceptions:** none

## MOV[.] – Move register to register

### Description:

This instruction moves one general purpose register to another. The data type is preserved. To copy a register and alter the data type see the MOVP and MOVI instructions.

**Instruction Format:** R1

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

Notes:

## MOVI – Move register to Integer register

### Description:

This instruction moves one general purpose register to another and sets the target register data type to indicate an integer.

**Instruction Format:** R1

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

Notes:

## MOVP – Move register to Pointer register

### Description:

This instruction moves one general purpose register to another and sets the target register data type to indicate a pointer.

**Instruction Format:** R1

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

Notes:

## MUL[.] – Signed Multiply

### Description:

Multiply two values. The first operand must be in a register Rs1. The second operand may be in a register Rs2 or may be an immediate value specified in the instruction. Both the operands are treated as signed values, the result is a signed result.

**Instruction Format:** RI, R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv, .bvs, .wvs, .tvs, .ovs, .hvs

**Clock Cycles:** 20

**Execution Units:** Integer

**Exceptions:** multiply overflow, if enabled

## MULF – Fast Unsigned Multiply

### Description:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as unsigned values. The result is an unsigned result. The fast multiply multiplies only the low order 24 bits of the first operand times the low order 16 bits of the second. The result is a 40-bit unsigned product. This instruction may be used to calculate array indices for small arrays.

**Instruction Format:** RI, R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## MULH – Signed Multiply

### Description:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as signed values, the result is a signed result. The high order bits of the product are returned.

**Instruction Format:** R3

**Clock Cycles:** 20

**Execution Units:** Integer

**Exceptions:** none

## MULU – Unsigned Multiply

### Description:

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as unsigned values. The result is an unsigned result.

### Comment:

Unsigned multiply is often used in address calculations for instance calculating array indexes.

**Instruction Format:** RI, R3

**Clock Cycles:** 20

**Execution Units:** Integer

**Exceptions:** none

## MULUH – Unsigned Multiply

**Description:**

Multiply two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction. Both the operands are treated as unsigned values. The result is an unsigned result. The high order bits of the result are returned.

**Instruction Format:** R3**Clock Cycles:** 20**Execution Units:** Integer**Exceptions:** none

## MUX[.] – Multiplex

### Description:

The MUX instruction performs a bit-by-bit copy of a bit of Rs2 to the target register Rd if the corresponding bit in Rs1 is set, or a copy of a bit from Rs3 if the corresponding bit in Rs1 is clear.

**Instruction Format:** R3

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## NAND[.] – Bitwise Nand

### Description:

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

**Instruction Format:** R3

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## NEG[.] – Negate

### Description:

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

**Instruction Format:** R1

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

### 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:** R3

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**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

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Operation:**
$$Rd = !Rs1$$

**Exceptions:** none

**Notes:**



## OR[.] – Bitwise Or

### Description:

Perform a bitwise or operation between operands. The immediate constant associated with the RI form of the instruction is zero extended to the size of the register.

**Instruction Format:** RI, R2, R2S

**Clock Cycles:** 0.25

**Execution Units:** Integer

### Scalar Operation

$$Rd = Rs1 \mid Rs2$$

OR

$$Rd = Rs1 \mid \text{Immediate}$$

### Vector Operation

for  $x = 0$  to  $VL - 1$

$$\text{if } (Vm[x]) \text{ } Rd[x] = Rs1[x] \mid Rs2[x]$$

$$\text{else if (zero) } Rd[x] = 0$$

$$\text{else } Rd[x] = Rd[x]$$

### Vector - Scalar Operation (R2S)

for  $x = 0$  to  $VL - 1$

$$\text{if } (Vm[x]) \text{ } Rd[x] = Rs1[x] \mid Rs2[0]$$

$$\text{else if (zero) } Rd[x] = 0$$

$$\text{else } Rd[x] = Rd[x]$$

## PFX – Constant Prefix

**Description:**

This instruction provides 31 bits of a constant. If the prefix present bit is set in this instruction then the constant will be appended to a previous constant. In this way constants up to 128 bits may be encoded directly in the instruction stream.

**Instruction Format:** PFX**Operation:****Clock Cycles:** 1.0**Execution Units:** Integer**Exceptions:** None.

## PTRDIF – Difference Between Pointers

### Description:

Subtract two values then shift the result right. Both operands must be in a register. The top 32 bits of each value are masked off before the subtract. 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: R3

### Operation:

$$Rd = (Rs1 - Rs2) \gg Sc$$

**Clock Cycles:** 0.25

**Execution Units:** Integer

### Exceptions:

None.

## ROL[.] – Rotate Left

### Description:

Bits from the source register Ra are shifted left by the amount in register Rb or an immediate value. The most significant bit is shifted into bit zero.

For the sub-word forms the result is sign extended to 64 bits.

**Instruction Format:** SHIFT, SHIFTI

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## ROR[.] – Rotate Right

### Description:

Bits from the source register Rs1 are shifted right by the amount in register Rs2 or an immediate value. The bit zero is shifted into the most significant bits.

For the sub-word forms the result is sign extended to 64 bits.

**Instruction Format:** SHIFT, SHIFTI

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## SHL[.] – Shift Left

**Description:**

Bits from the source register Rs1 are shifted left by the amount in register Rs2 or an immediate value. Zeros are shifted into the least significant bits.

**Instruction Format:** SHIFT, SHIFTI

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## SUB[.] - Subtraction

### Description:

Subtract Rs2 from Rs1, place the result in register Rd. The immediate form of this instruction is an alternate mnemonic for the ADD instruction.

**Instruction Formats:** Integer RI, R2, R2S

**Supported Formats:** .b .w, .t, .o, .h, .bv, .wv, .tv, .ov, .hv

**Clock Cycles:** 0.25

**Execution Units:** All ALU's

**Exceptions:** none

**Notes:**

## SWAP[.] – Swap Register Half

### Description:

Swap the high order and low order 128-bits of a register, placing the result in register Rd. This operation may be useful to load 256-bits of a register with an immediate value or from memory.

**Instruction Formats:** Integer R1

**Clock Cycles:** 0.25

**Execution Units:** All ALU's

**Exceptions:** none

**Notes:**

## SXB[.] – Sign Extend Byte

### Description:

Sign extend a byte (8 bits) as a signed value to the full width of the machine.

**Instruction Format:** R1

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## SXO[.] – Sign Extend Octa Byte

### Description:

Sign extend an octa byte (64 bits) as a signed value to the full width of the machine.

**Instruction Format:** R1

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## SXT[.] – Sign Extend Tetra Byte

### Description:

Sign extend a tetra byte (32 bits) as a signed value to the full width of the machine.

**Instruction Format:** R1

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## SXW[.] – Sign Extend Wyde

### Description:

Sign extend a wyde (16 bits) as a signed value to the full width of the machine.

**Instruction Format:** R1

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none





## WYDNDX – Wyde Index

**Description:**

This instruction searches Rs1, which is treated as an array of eight 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 +7. The index of the first found wyde is returned (closest to zero).

**Instruction Format:** R3**Clock Cycles:** 0.25**Execution Units:** Integer**Operation:**
$$Rd = \text{Index of } (Rs2 \text{ in } Rs1)$$
**Exceptions:** none

## XNOR[.] – 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, R2S

**Clock Cycles:** 0.25

**Execution Units:** Integer

### Scalar Operation

$$Rd = \sim(Rs1 \wedge Rs2)$$

### Vector Operation

for  $x = 0$  to  $VL - 1$

if ( $Vm[x]$ )  $Rd[x] = \sim(Rs1[x] \wedge Rs2[x])$

else if (zero)  $Rd[x] = 0$

else  $Rd[x] = Rd[x]$

### Vector - Scalar Operation (R2S)

for  $x = 0$  to  $VL - 1$

if ( $Vm[x]$ )  $Rd[x] = \sim(Rs1[x] \wedge Rs2[0])$

else if (zero)  $Rd[x] = 0$

else  $Rd[x] = Rd[x]$

## XOR[.] – Bitwise Exclusive Or

### Description:

Perform a bitwise exclusive or operation between operands. The first operand must be in a register. The second operand may be a register or immediate value.

**Instruction Format:** RI, R2, R2S

**Clock Cycles:** 0.25

**Execution Units:** Integer

### Scalar Operation

$$Rd = Rs1 \wedge Rs2$$

OR

$$Rd = Rs1 \wedge \text{Immediate}$$

### Vector Operation

for  $x = 0$  to  $VL - 1$

if ( $Vm[x]$ )  $Rd[x] = Rs1[x] \wedge Rs2[x]$

else if (zero)  $Rd[x] = 0$

else  $Rd[x] = Rd[x]$

### Vector - Scalar Operation (R2S)

for  $x = 0$  to  $VL - 1$

if ( $Vm[x]$ )  $Rd[x] = Rs1[x] \wedge Rs2[0]$

else if (zero)  $Rd[x] = 0$

else  $Rd[x] = Rd[x]$

## ZXB[.] – Zero Extend Byte

**Description:**

Zero extend a byte (8 bits) as an unsigned value to the full width of the machine.

**Instruction Format:** R1

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## ZXO[.] – Zero Extend Octa Byte

**Description:**

Zero extend an octa byte (64 bits) as an unsigned value to the full width of the machine.

**Instruction Format:** R1**Clock Cycles:** 0.25**Execution Units:** Integer**Exceptions:** none

## ZXT[.] – Zero Extend Tetra

**Description:**

Zero extend a tetra byte (32 bits) as an unsigned value to the full width of the machine.

**Instruction Format:** R1

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## ZXW[.] – Zero Extend Wyde

**Description:**

Zero extend a wyde byte (16 bits) as an unsigned value to the full width of the machine.

**Instruction Format:** R1

**Clock Cycles:** 0.25

**Execution Units:** Integer

**Exceptions:** none

## Memory Unit



## Preload Instructions

Issuing a load instruction with a target register of r0 causes the cache line to be loaded without updating the register file. If an exception occurs during the load, it will be ignored.

## AMO – Atomic Memory Operation

### Description:

The atomic memory operations read from memory addressed by the Rs1 register and store the value in Rd. As a second step the value from memory is combined with the value in register Rs2 or an immediate constant according to one of the available functions then stored back into the memory addressed by Rs1.

### Instruction Format: AMO, AMOI

| Funct <sub>4</sub> | Mnemonic | Operation Performed  |  |
|--------------------|----------|----------------------|--|
| 00                 | amoswap  | swap                 | memory[Ra] = Rb                                |
| 01                 | amoadd   | addition             | memory[Ra] = memory[Ra] + Rb                   |
| 02                 | amoand   | bitwise and          | memory[Ra] = memory[Ra] & Rb                   |
| 03                 | amoor    | bitwise or           | memory[Ra] = memory[Ra]   Rb                   |
| 04                 | amoxor   | bitwise exclusive or | memory[Ra] = memory[Ra] ^ Rb                   |
| 05                 | amoshl   | shift left           | memory[Ra] = memory[Ra] << Rb                  |
| 06                 | amoshr   | shift right          | memory[Ra] = memory[Ra] >> Rb                  |
| 07                 | amomin   | minimum              | memory[Ra] = memory[Ra] < Rb ? memory[Ra] : Rb |
| 08                 | amomax   | maximum              | memory[Ra] = memory[Ra] > Rb ? memory[Ra] : Rb |
| 09                 | amominu  | minimum unsigned     | memory[Ra] = memory[Ra] < Rb ? memory[Ra] : Rb |
| 0A                 | amomaxu  | maximum unsigned     | memory[Ra] = memory[Ra] > Rb ? memory[Ra] : Rb |

| Sz <sub>3</sub> |          |
|-----------------|----------|
| 0               | Byte     |
| 1               | Wyde     |
| 2               | Tetra    |
| 3               | Penta    |
| 4               | Octa     |
| 5               | Deci     |
| 6               | reserved |
| 7               | reserved |

Acquire and release bits determine the ordering of memory operations.

A = acquire – 1 = no following memory operations can take place before this one

R = release – 1 = this memory operation cannot take place before prior ones.

All combinations of A, R are allowed.

## CACHE – Cache Command

CACHE Cmd, d[Rn]

CACHE Cmd, d[Ra + Rc \* scale]

### 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:

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

| DCCmd <sub>3</sub> | 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)   |
|                    |         |  |

### Operation:

Register Indirect with Displacement Form

$$\text{Line} = \text{round}_{32}(\text{sign extend}(\text{memory}[\text{displacement} + \text{Ra}]))$$

Register-Register Form

$$\text{Line} = \text{round}_{32}(\text{sign extend}(\text{memory}[\text{Ra} + \text{Rc} * \text{scale}]))$$

Notes:

| Sc <sub>2</sub> Code | Multiply By |
|----------------------|-------------|
| 0                    | 1           |
| 1                    | 2           |
| 2                    | 4           |
| 3                    | 8           |

## LDB – Load Byte

### Description:

This instruction loads a byte (8 bit) value from memory. The value is sign extended to 64 bits when placed in the target register.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDBU – Load Unsigned Byte

### Description:

This instruction loads a byte (8 bit) value from memory. The value is zero extended to 64 bits when placed in the target register.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDFD – Load Float Double (64 bits)

### Description:

This instruction loads a float double (64 bit) value from memory. The value is converted to a quad precision float when placed in the target register.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDFS – Load Float Single (32 bits)

### Description:

This instruction loads a float single (32 bit) value from memory. The value is converted to a quad precision float when placed in the target register.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDFQ – Load Float Quad (128 bits)

### Description:

This instruction loads a quad precision float (128 bit) value from memory.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDH – Load Hexi Byte (128 bits)

### Description:

This instruction loads a hexi-byte (128 bit) value from memory.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDHR – Load Hexi Byte and Reserve

### Description:

This instruction loads a byte (128 bit) value from memory and places a reservation on the address range associated with the given address. The region of memory which is reserved depends on what is supported by the external memory system hardware. Typically, a 128-bit region or larger would be reserved.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDO – Load Octa Byte (64 bits)

### Description:

This instruction loads a byte (64 bit) value from memory.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDOU – Load Unsigned Octa Byte

### Description:

This instruction loads a byte (64 bit) value from memory.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |



## LDT – Load Tetra Byte (32 bits)

### Description:

This instruction loads a penta-byte (32 bit) value from memory. The value is sign extended to 64 bits when placed in the target register.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDTU – Load Unsigned Tetra Byte

### Description:

This instruction loads a penta-byte (32 bit) value from memory. The value is zero extended to 64 bits when placed in the target register.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDW – Load Wyde (16 bits)

### Description:

This instruction loads a wyde (16 bit) value from memory. The value is sign extended to 64 bits when placed in the target register.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## LDWU – Load Unsigned Wyde

### Description:

This instruction loads a byte (16 bit) value from memory. The value is zero extended to 64 bits when placed in the target register.

**Instruction Formats:** ML, MLX

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | reserved    |

## MEMDB –Memory Data Barrier

**Description:**

All memory instructions before the MEMDB are completed and committed to the architectural state before memory instructions after the MEMDB are issued. This instruction is used to ensure that the memory state is valid before subsequent instructions are executed.

**Instruction Format:** MEMDB

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

**Execution Units:** Memory Store

**Operation:**

**Exceptions:** none

## MEMSB –Memory Synchronization Barrier

**Description:**

This instruction is similar to the SYNC instruction except that it applies only to memory operations. All instructions before the MEMSB are completed and committed to the architectural state before memory instructions after the MEMSB are issued. This instruction is used to ensure that the memory state is valid before subsequent instructions are executed.

**Instruction Format:** MEMSB

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

**Execution Units:** Memory Store

**Operation:**

**Exceptions:** none

## SEQ – Set if Equal

### Synopsis

Vector register set.  $Md = Rs1 == Rs2$

### Description

Two vector registers ( $Rs1$  and  $Rs2$ ) are compared for equality and the comparison result is placed in the target vector mask register  $Md$ .

### Instruction Format: R2, R2S

**Supported Formats:** .wv, .tv, .ov, .hv

### Operation

```

for x = 0 to VL-1
    if (Vm[x])
        Vmt[x] = Rs1[x] == Rs2[x]
```

## SLT – Set if Less Than

### Synopsis

Vector register set.  $Md = Rs1 < Rs2$

### Description

Two vector registers ( $Rs1$  and  $Rs2$ ) are compared for less than and the comparison result is placed in the target vector mask register  $Md$ .

### Instruction Format: R2, R2S

**Supported Formats:** .wv, .tv, .ov, .hv

### Operation

```

for x = 0 to VL-1
    if (Vm[x])
        Vmt[x] = Rs1[x] < Rs2[x]
```

# SNE – Set if Not Equal

## Synopsis

Vector register set.  $Md = Rs1 \lessgtr Rs2$

## Description

Two vector registers ( $Rs1$  and  $Rs2$ ) are compared for inequality and the comparison result is placed in the target vector mask register  $Md$ .

## Instruction Format: R2, R2S

**Supported Formats:** .wv, .tv, .ov, .hv

## Operation

```

for x = 0 to VL-1
    if (Vm[x])
        Vmt[x] = Rs1[x] <> Rs2[x]
```

## STB – Store Byte (8 bits)

### Description:

This instruction stores a byte (8 bit) value to memory.

### Instruction Format: MS, MSX

### Operation:

$$\text{Memory}_8[\text{Rs1} + \text{immediate}] = \text{Rs2}$$

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

The number of clock cycles required by this instruction may be “hidden” because the core will continue to process instructions while the store is being processed.

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | 5           |

### Execution Units: Memory Store

### Exceptions: DBE, TLB, WRV

### Notes:

A store instruction will only be issued by the core if it is known that it will be able to commit without being interrupted by a change of program flow due to a previous instruction. Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.

## STH – Store Hexi (128 bits)

### Description:

This instruction stores a hexi byte (128 bit) value to memory.

**Instruction Format:** MS, MSX

**Operation:**

$$\text{Memory}_{128}[\text{Rs1} + \text{immediate}] = \text{Rs2}$$

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

The number of clock cycles required by this instruction may be “hidden” because the core will continue to process instructions while the store is being processed.

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | 5           |

**Execution Units:** Memory Store

### Notes:

A store instruction will only be issued by the core if it is known that it will be able to commit without being interrupted by a change of program flow due to a previous instruction. Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.

## STO – Store Octa (64 bits)

### Description:

This instruction stores an octa byte (64 bit) value to memory.

### Instruction Format: MS, MSX

### Operation:

$\text{Memory}_{64}[\text{Rs1} + \text{immediate}] = \text{Rs2}$

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

The number of clock cycles required by this instruction may be “hidden” because the core will continue to process instructions while the store is being processed.

| $\text{Sc}_3$ | Scale Rb By |
|---------------|-------------|
| 0             | 1           |
| 1             | 2           |
| 2             | 4           |
| 3             | 8           |
| 4             | 16          |
| 5             | reserved    |
| 6             | reserved    |
| 7             | 5           |

### Execution Units: Memory Store

### Notes:

A store instruction will only be issued by the core if it is known that it will be able to commit without being interrupted by a change of program flow due to a previous instruction. Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.



## STP – Store Penta (40 bits)

### Description:

This instruction stores an octa byte (40 bit) value to memory.

### Instruction Format: MS, MSX

### Operation:

$$\text{Memory}_{40}[\text{Rs1} + \text{immediate}] = \text{Rs2}$$

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

The number of clock cycles required by this instruction may be “hidden” because the core will continue to process instructions while the store is being processed.

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | 5           |

### Execution Units: Memory Store

### Notes:

A store instruction will only be issued by the core if it is known that it will be able to commit without being interrupted by a change of program flow due to a previous instruction. Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.

## STT – Store Tetra (32 bits)

### Description:

This instruction stores a tetra (32 bit) value to memory.

### Instruction Format: MS, MSX

### Operation:

$$\text{Memory}_{32}[\text{Rs1} + \text{immediate}] = \text{Rs2}$$

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

The number of clock cycles required by this instruction may be “hidden” because the core will continue to process instructions while the store is being processed.

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | 5           |

### Execution Units: Memory Store

### Notes:

A store instruction will only be issued by the core if it is known that it will be able to commit without being interrupted by a change of program flow due to a previous instruction. Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.

## STW – Store Wyde (16 bits)

### Description:

This instruction stores a wyde (16 bit) value to memory.

### Instruction Format: MS, MSX

### Operation:

$$\text{Memory}_{16}[\text{Rs1} + \text{immediate}] = \text{Rs2}$$

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

The number of clock cycles required by this instruction may be “hidden” because the core will continue to process instructions while the store is being processed.

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | 5           |

### Execution Units: Memory Store

### Notes:

A store instruction will only be issued by the core if it is known that it will be able to commit without being interrupted by a change of program flow due to a previous instruction. Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.

## TLB – TLB Command

### Description:

The command is executed on the TLB unit. The command results are placed in internal TLB registers which can be read or written using TLB command instruction. If the operation is a read register operation, then the register value is placed into Rd. If the operation is a write register operation, then the value for the register comes from Rs1. Otherwise the Rs1/Rd field in the instruction is ignored.

This instruction is only available at the machine operating level.

### Instruction Format: TLB

### Clock Cycles: 3

It typically takes several cycles for the TLB to process the command.

Tn<sub>4</sub> – This field identifies which TLB register is being read or written.

| Tn <sub>4</sub> |                         | Assembler |
|-----------------|-------------------------|-----------|
| 0               | Wired                   | Wired     |
| 1               | Index                   | Index     |
| 2               | Random                  | Random    |
| 3               | Page Size               | PageSize  |
| 4               | Virtual page            | VirtPage  |
| 5               | Physical page           | PhysPage  |
| 7               | ASID                    | ASID      |
| 8               | Miss address            | MA        |
| 9               | reserved                |           |
| 10              | Page Table Address      | PTA       |
| 11              | Page Table Control      | PTC       |
| 12              | Aging frequency control | AFC       |

### TLB Commands

| Cmd <sub>5</sub> | Description                                      | Assembler |
|------------------|--|-----------|
| 0                | No operation                                     |           |
| 1                | Probe TLB entry                                  | TLBPB     |
| 2                | Read TLB entry                                   | TLBRD     |
| 3                | Write TLB entry corresponding to random register | TLBWR     |
| 4                | Write TLB entry corresponding to index register  | TLBWI     |
| 5                | Enable TLB                                       | TLBEN     |
| 6                | Disable TLB                                      | TLBDIS    |
| 7                | Read register                                    | TLBRDREG  |
| 8                | Write register                                   | TLBWRREG  |
| 9                | Invalidate all entries                           | TLBINV    |
| 10               | Get age count                                    | TLBRDAGE  |
| 11               | Set age count                                    | TLBWRAGE  |

Probe TLB – The TLB will be tested to see if an address translation is present.

Read TLB – The TLB entry specified in the index register will be copied to TLB holding registers.

Write Random TLB – A random TLB entry will be written into from the TLB holding registers.

Write Indexed TLB – The TLB entry specified by the index register will be written from the TLB holding registers.

Disable TLB – TLB address translation is disabled so that the physical address will match the supplied virtual address.

Enable TLB – TLB address translation is enabled. Virtual address will be translated to physical addresses using the TLB lookup tables.

The TLB will automatically update the miss address registers when a TLB miss occurs only if the registers are zero to begin with. System software must reset the registers to zero after a miss is processed. This mechanism ensures the first miss that occurs is the one that is recorded by the TLB.

PageTableAddr – This is a scratchpad register available for use to store the address of the page table.

PageTableCtrl – This is a scratchpad register available for use to store control information associated with the page table.

**Notes:**

The TLB automatically tracks reference counts and ages address mappings.

## VLDB – Vector Load Byte

### Description:

This instruction loads contiguous bytes (8 bit) values from memory into successive elements of a vector register. The values are loaded under the control of mask register zero.

### Instruction Formats: ML

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | 5           |

## VLDBWS – Vector Load Byte with Stride

### Description:

This instruction loads bytes (8 bit) values from memory into successive elements of a vector register. The value is sign extended to 128 bits when placed in the target register. The values are loaded under the control of the mask register specified by the Sc<sub>3</sub> field of the instruction. Each value is located one stride away from the previous value. The stride amount is contained in register Rs3.

### Instruction Formats: MLVS

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

## VLDBX – Vector Load Byte Indexed

### Description:

This instruction loads bytes (8 bit) values from memory into successive elements of a vector register. The value is sign extended to 128 bits when placed in the target register. The values are loaded under the control of the vector mask register specified by the Sc<sub>3</sub> field of the instruction. The address of each value is the sum of integer register Rs1 and an element from vector register Rs3.

### Instruction Formats: MLVX

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

## VLDH – Vector Load Hexi-byte

### Description:

This instruction loads contiguous hexi-bytes (128 bit) values from memory into successive elements of a vector register. The values are loaded under the control of mask register zero.

**Instruction Formats:** ML

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

| Sc <sub>3</sub> | Scale Rb By |
|-----------------|-------------|
| 0               | 1           |
| 1               | 2           |
| 2               | 4           |
| 3               | 8           |
| 4               | 16          |
| 5               | reserved    |
| 6               | reserved    |
| 7               | 5           |

## VSTB – Vector Store Byte (8 bits)

### Description:

This instruction stores bytes (the low order 8 bit) values from a vector register (Rs2) to memory. The bytes are stored in contiguous addresses. The store operation is predicated by mask register #0.

### Instruction Format: MS, MSX

### Operation:

```
for n = 0 to vector length
    if (vector mask[n])
        Memory8[Rs1 + immediate + n] = Rs2.element[n][7..0]
```

**Clock Cycles:** 4 minimum depending on memory access time, vector length and predication.

**Execution Units:** Memory Store

**Exceptions:** DBE, TLB, WRV

### Notes:

A store instruction will only be issued by the core if it is known that it will be able to commit without being interrupted by a change of program flow due to a previous instruction. Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.



## VSTW – Vector Store Wyde (16 bits)

### Description:

This instruction stores wydes (16 bit) values from a vector register (Rs2) to memory. The wydes are stored in contiguous addresses. The store operation is predicated by mask register #0.

### Instruction Format: MS, MSX

### Operation:

```
for n = 0 to vector length
    if (vector mask[n])
        Memory16[Rs1 + immediate + n * 2] = Rs2.element[n][15..0]
```

**Clock Cycles:** 4 minimum depending on memory access time, vector length and predication.

**Execution Units:** Memory Store

**Exceptions:** DBE, TLB, WRV

### Notes:

A store instruction will only be issued by the core if it is known that it will be able to commit without being interrupted by a change of program flow due to a previous instruction. Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.



## Floating Point Instruction Description

### 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 extended double precision floating-point operations are supported.

Unlike some FP implementations, the core allows the use of immediate constants supplied in the instruction stream as an operand. The FLTI format instructions use a large immediate constant as the second operand to the instruction.

The precision field ( $\text{prec}_3$ ) should be set to 3.

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.

Internally the core maintains four extra mantissa bits, these extra bits are truncated on stores and zeroed on loads. However, they are used during calculations, as long as results remain in registers the four extra bits will provide four more bits of precision.

### Representation

The floating-point format is similar to an IEEE-754 representation for quad precision. Briefly,

#### Quad Precision Format:

|       |       |          |          |     |   |
|-------|-------|----------|----------|-----|---|
| 127   | 126   | 125      | 112      | 111 | 0 |
| $S_M$ | $S_E$ | Exponent | Mantissa |     |   |

$S_M$  – sign of mantissa

$S_E$  – sign of exponent

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

|                  |                  |
|------------------|------------------|
| $S_e$ EEEEEEEEEE |                  |
| 1111111111       | Maximum exponent |
| ....             |                  |
| 0111111111       | exponent of zero |
| ....             |                  |
| 0000000000       | Minimum exponent |

The exponent ranges from -16383 to +16384 for quad precision numbers

| $\text{Prec}_3$ | Precision       |
|-----------------|-----------------|
| 0               | Half (16 bit)   |
| 1               | Single (32 bit) |
| 2               | Double (64 bit) |
| 3               | Quad (128 bit)  |

|        |          |
|--------|----------|
| 4 to 7 | reserved |
|--------|----------|

## FCVT.Q.S – Convert 32-bit to 128-bit

### Description:

Convert the single precision value (32 bits) in Rs1 into a floating-point quad value (128 bits) and place the result into target register Rd.

**Instruction Format:** FLT1

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

## FCVT.D.S – Convert 32-bit to 64-bit

### Description:

Convert the single precision value (32 bits) in Rs1 into a floating-point double value (64 bits) and place the result into target register Rd.

**Instruction Format:** FLT1

**Clock Cycles:** 3

**Execution Units:** Floating Point

## FCVT.Q.D – Convert 64-bit to 128-bit

### Description:

Convert the double precision value (64 bits) in Rs1 into a floating-point quad value (128 bits) and place the result into target register Rd.

**Instruction Format:** FLT1

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

## FCVT.S.Q – Convert 128-bit to 32-bit

### Description:

Convert the quad precision value (128 bits) in Rs1 into a floating-point single value (32 bits) and place the result into target register Rd. Not all values are possible to convert to 32-bits, some precision is lost. Values outside the range of a 32-bit float may be set to either infinity or zero. The 108-bit mantissa is truncated to 23 bits. Nans are propagated.

**Instruction Format:** FLT1

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

## FCVT.D.Q – Convert 128-bit to 64-bit

### Description:

Convert the quad precision value (128 bits) in Rs1 into a floating-point double value (64 bits) and place the result into target register Rd. Not all values are possible to convert to 64-bits, some precision is lost. Values outside the range of a 64-bit float may be set to either infinity or zero. The 108-bit mantissa is truncated to 52 bits. Nans are propagated.

**Instruction Format:** FLT1

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

## FABS[.] – Floating Absolute Value

### Description:

This instruction is an alternate mnemonic for the FSGNCPY instruction where VRs2 is zero. It effectively takes the absolute value of the floating-point number in register VRs1 and places the result into target register VRd. The sign bit (bit 127) of the register is set to zero. No rounding of the number occurs.

**Instruction Format:** FLT2

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

## FADD[.] – Floating point addition

**Description:**

Add two floating point numbers in registers FRs1 and FRs2 or an immediate constant and place the result into target register FRd.

**Instruction Format:** FLT2, FLTI

**Supported Formats:** .h, .s, .d, .q, .hv, .sv, .dv, .qv

**Clock Cycles:** 6

**Execution Units:** Floating Point

# FASGE - Float Set if Absolute Value Greater Than or Equal

**Description:**

The register compare instruction compares the absolute values in two registers or a register and an immediate constant as floating-point values for greater than or equal and sets the target register as a result.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

**Operation:**

```
if abs(Rs1) >= abs(Rs2)
    Rd = true
else
    Rd = false
```



## FCLASS – Classify Value

### Description:

FCLASS classifies the value in register VRs1 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                 |

## FCMP - Float Compare

### Description:

The register compare instruction compares two registers or a register and an immediate constant as floating-point values and sets the flags in the target condition register as a result.

**Instruction Format:** FLT2, FLTI

**Supported Formats:** .h, .s, .d, .q

**Clock Cycles:** 2

**Execution Units:** Floating Point

### Operation:

```
if Rs1 < Rs2
    Cr.lt = true
else
    Cr.lt = false
if Rs1 = Rs2
    Cr.eq = true
else
    Cr.eq = false
if Rs1 > Rs2
    Cr.gt = true
else
    Cr.gt = false
if unordered
    Cr.un = true
else
    Cr.un = false
```

## FCMPM - Float Compare Magnitude

### Description:

The register compare instruction compares two registers or a register and an immediate constant as floating-point values and sets the flags in the target condition register as a result. Only the magnitude of the values are considered – the sign of the numbers is ignored.

**Instruction Format:** FLT2, FLTI

**Supported Formats:** .h, .s, .d, .q

**Clock Cycles:** 2

**Execution Units:** Floating Point

### Operation:

```
if mag Rs1 < Rs2
    Cr.lt = true
else
    Cr.lt = false
if Rs1 = Rs2
    Cr.eq = true
else
    Cr.eq = false
if mag Rs1 > Rs2
    Cr.gt = true
else
    Cr.gt = false
if unordered
    Cr.un = true
else
    Cr.un = false
```



## 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 register Rs1 and an immediate field in the instruction. Either the immediate or Rs1 should be zero.

### Instruction Format: FLT2

**Execution Units:** All Floating Point

### Operation:

### Exceptions:

| Bit | Exception Enabled   |
|-----|---|
| 0   | global invalid operation clears the following: <ul style="list-style-type: none"> <li>- division of infinities</li> <li>- zero divided by zero</li> <li>- subtraction of infinities</li> <li>- infinity times zero</li> <li>- NaN comparison</li> <li>- division by zero</li> </ul> |
| 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 register Rs1 and an immediate field in the instruction. Either the immediate or Rs1 should be zero. Exceptions won't 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.

**Instruction Format:** FXX**Clock Cycles:** 2**Execution Units:** Floating Point

## FDIV[.] – Floating point divide

### Description:

Divide two floating point numbers in registers VRs2 and VRs2 or an immediate constant and place the result into target register VRd.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 150 (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 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

## FMA – Floating Point Multiply and Add

### Description:

Multiply two floating point numbers in registers Rs1 and Rs2 or an immediate constant then add the floating-point number in Rs3, and place the result into target register Rd.

**Instruction Format:** FMA

**Clock Cycles:** 18

**Execution Units:** Floating Point

## FMAX – Maximum Value

### Description:

Determines the maximum of three values in registers Rs1, Rs2, Rs3 and places the result in the target register Rd. In order to obtain the maximum value of two registers specify one of the registers twice.

**Instruction Format:** FLT3

**Clock Cycles:** 0.33

**Execution Units:** Integer

### Operation:

```
IF Rs1 > Rs2 and Rs1 > Rs3
    Rd = Rs1
else if Rs2 > Rs3
    Rd = Rs2
else
    Rd = Rs3
```

**Exceptions:** none



## FMIN – Minimum Value

### Description:

Determines the minimum of three values in registers Rs1, Rs2, Rs3 and places the result in the target register Rd. In order to obtain the minimum value of two registers specify one of the registers twice.

**Instruction Format:** FLT3

**Clock Cycles:** 0.33

**Execution Units:** Integer

### Operation:

```
IF Rs1 < Rs2 and Rs1 < Rs3
    Rd = Rs1
else if Rs2 < Rs3
    Rd = Rs2
else
    Rd = Rs3
```

**Exceptions:** none

## FMS – Floating Point Multiply and Subtract

### Description:

Multiply two floating point numbers in registers Rs1 and Rs2 or an immediate constant then subtract the floating-point number in Rs3, and place the result into target register Rd.

**Instruction Format:** FMA

**Clock Cycles:** 18

**Execution Units:** Floating Point

## FMUL[.] – Floating point multiplication

**Description:**

Multiply two floating point numbers in registers VRs1 and VRs2 or an immediate constant and place the result into target register VRd.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 6

**Execution Units:** Floating Point

## FNABS – Floating Negative Absolute Value

### Description:

Take the negative absolute value of the floating-point number in register Rs1 and place the result into target register Rd. The sign bit (bit 79) of the register is set to one. No rounding of the number occurs. This instruction is an alternate mnemonic for the FSGNINV instruction where Rs2 is zero.

**Instruction Format:** FLT1

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

## FNEG – Floating Negative Value

### Description:

Negate the value of the floating-point number in register Rs1 and place the result into target register Rd. The sign bit (bit 79) of the register is inverted. No rounding of the number occurs. This instruction is an alternate mnemonic for the FSGNINV instruction where Rs1 and Rs2 specify the same register.

**Instruction Format:** FLT1

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

## FNEXT – Floating Point Next Value

### Description:

FNEXT returns the next representable value in the direction indicated by the difference between the values in Rs1 and Rs2.

**Instruction Format:** FLT2

**Clock Cycles:** 1

**Execution Units:** Floating Point

## FNMA – Floating Point Negate Multiply and Add

### Description:

Multiply two floating point numbers in registers Rs1 and Rs2 or an immediate constant negate the result then add the floating-point number in Rs3, and place the result into target register Rd.

**Instruction Format:** FMA

**Clock Cycles:** 18

**Execution Units:** Floating Point

## FNMS – Floating Point Negate Multiply and Subtract

### Description:

Multiply two floating point numbers in registers Rs1 and Rs2 or an immediate constant negate the result then subtract the floating-point number in Rs3, and place the result into target register Rd.

**Instruction Format:** FMA

**Clock Cycles:** 18

**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 Rs1. The value is returned in Rd as an 80-bit floating-point value. The value returned is accurate to about eight bits.

**Instruction Format:** FLT1A

**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 Rs1 and place the result into target register Rd. This instruction may not be present in all models of the core.

**Instruction Format:** FLT1A

**Clock Cycles:** 3

**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 an 80-bit format. That means that input values must be 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:** 0.33

**Execution Units:** Floating Point

## FSEQ - Float Set if Equal

### Description:

The register compare instruction compares two registers or a register and an immediate constant as floating-point values for equality and sets the mask target register as a result. Note that negative and positive zero are considered equal.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 0.25

**Execution Units:** Floating Point

### Operation:

```
if Rs1 == Rs2
    Rd = true
else
    Rd = false
```

## FSGE - Float Set if Greater Than or Equal

### Description:

The register compare instruction compares two registers or a register and an immediate constant as floating-point values for greater than or equal and sets the mask target register as a result.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 0.25

**Execution Units:** Floating Point

### Operation:

```
if Rs1 >= Rs2
    Rd = true
else
    Rd = false
```

## FSGN[op] – Sign Operation

### Description:

FSGNCPY copies the sign of one value to another value. This instruction may be used to get the absolute value.

FSGNINV copies the inverted sign of one value to another. This instruction may be used to get the negative absolute value.

FSGNOR logically or's the sign of the two values.

FSGNAND logically and's the sign of the two values.

FSGNXOR logically xor's the sign of the two values.

FSGNXNOR logically xnor's the sign of the two values.

### Instruction Format: FLT2, FLTI

| Op <sub>3</sub> | Operation |
|-----------------|-----------|
| 0               | FSGNCPY   |
| 1               | FSGNINV   |
| 4               | FSGNAND   |
| 5               | FSGNOR    |
| 6               | FSGNXOR   |
| 7               | FSGNXNOR  |

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

**Operation:**

## FSGT - Float Set if Greater Than

### Description:

The register compare instruction compares two registers or a register and an immediate constant as floating-point values for greater than and sets the mask register as a result.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 0.25

**Execution Units:** Floating Point

### Operation:

```
if FRs1 > FRs2
    Md = true
else
    Md = false
```



## FSIG – Sigmoid Approximate

### Description:

This function uses a 1024 entry 32-bit precision lookup table with linear interpolation to approximate the logistic sigmoid function in the range -8.0 to +8.0. Outside of this range 0.0 or +1.0 is returned. The sigmoid output is between 0.0 and +1.0. The value of the sigmoid for register Rs1 is returned in register Rd as an 80-bit floating-point value.

**Instruction Format:** FLT1A

**Clock Cycles:** 5

**Execution Units:** Floating Point

## FSLE - Float Set if Less Than or Equal

### Description:

The register compare instruction compares two registers or a register and an immediate constant as floating-point values for less than or equal and sets the integer target register as a result.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

### Operation:

```
if Rs1 <= Rs2
    Rd = true
else
    Rd = false
```

## FSLT - Float Set if Less Than

### Description:

The register compare instruction compares two registers or a register and an immediate constant as floating-point values for less than and sets the integer target register as a result.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

### Operation:

```
if Rs1 < Rs2
    Rd = true
else
    Rd = false
```

## FSNE - Float Set if Not Equal

### Description:

The register compare instruction compares two registers or a register and an immediate constant as floating-point values for inequality and sets the integer target register as a result. Note that negative and positive zero are considered equal.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

### Operation:

```
if Rs1 <> Rs2
    Rd= true
else
    Rd= false
```

## FSQRT[.] – Floating point square root

**Description:**

Take the square root of the floating-point number in register VRs1 and place the result into target register Rd. The sign bit (bit 127) of the register is set to zero. This instruction can generate NaNs.

**Instruction Format:** FLT1

**Clock Cycles:** 150 (est).

**Execution Units:** Floating Point

## FSUB[.] – Floating point subtraction

### Description:

Subtract two floating-point numbers in registers FRs1 and FRs2 or an immediate constant and place the result into target register FRd.

**Instruction Format:** FLT2, FLTI

**Clock Cycles:** 10

**Execution Units:** Floating Point

## FSUN - Float Set if Unordered

### Description:

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

**Instruction Format:** FLT2

**Clock Cycles:** 0.25

**Execution Units:** Floating Point

### Operation:

```
if Ra ?? Rb
    Rt = true
else
    Rt = false
```

## 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 Rs1 into an integer and place the result into target register Rd. If the result overflows the value placed in Rd is a maximum integer value.

**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, trunc(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:** F1

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

## FTX – Trigger Floating Point Exceptions

**Description:**

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

**Instruction Format:** FXX**Execution Units:** Floating Point**Operation:****Exceptions:**

| Bit | Exception Enabled        |
|-----|--------------------------|
| 0   | global invalid operation |
| 1   | overflow                 |
| 2   | underflow                |
| 3   | divide by zero           |
| 4   | inexact operation        |
| 5   | reserved                 |



## ISNAN – Is Not a Number

**Description:**

Test the value in Rs1 to see if it's a nan (not a number) and return true (1) or false (0) in register Rd.

**Instruction Format:** FLT1

**Clock Cycles:** 0.33

**Execution Units:** Floating Point

**Example:**

```
isnan    $r1,$r7
```

## ITOF – Convert Integer to Float

**Description:**

Convert the integer value in Rs1 into a floating-point value and place the result into target register Rd. Rs1 is from either the floating-point register set or the integer register set, Rd is in the floating-point register set. Some precision of the integer converted may be lost if the integer is larger than 64 bits. Extended double precision floating point values only have a precision of 65 bits.

**Instruction Format:** FLT1

M1: Ra register set. 0 = use float register set, 1 = move from integer register set to float

**Clock Cycles:** 3

**Execution Units:** Floating Point

## Vector Instructions

The ISA supports up to 32 array registers of length 1 to 32. Vector operations are enabled by setting the most significant bit of the format field in the instruction.

### Vector Length (VL register)

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

|    |   |                          |   |
|----|---|--------------------------|---|
| 15 | 8 | 7                        | 0 |
| 0  |   | Elements <sub>7..0</sub> |   |

### Vector Masking

All vector operations are performed conditionally depending on the setting in the vector mask register unless otherwise noted. Vector masking has two options. 1) perform the operation on an element if the mask is set, otherwise retain the current value of the target element. This is called a mask merge. 2) perform the operation on an element if the mask is set, otherwise set the target element to zero. This is called mask zero.

### Vector Mask (mx registers)

The ISA supports up to eight, 32 element vector mask registers. All vector instructions are executed conditionally based on the value in a vector mask register. The mask register may be set using one of the vector set instructions SEQ, SNE, SLT, SLE, SLTU, SLEU. Mask registers may also be manipulated using one of the mask register operations MAND, MOR, MXOR, MXNOR, MFILL.

On reset the array mask registers are set to all ones.

### Instruction Fields

M<sub>3</sub> specifies the mask register to use

z specifies to zero the result if the operation is masked, otherwise merge the result into the target vector register

## Precision and Rounding

When  $Rm_3$  is one of 0,1,2,3,4 or 7  $prec_3$  represents a floating-point precision.

| $Prec_3$ | Precision       |
|----------|-----------------|
| 0        | Half (16 bit)   |
| 1        | Single (32 bit) |
| 2        | Double (64 bit) |
| 3        | Quad (128 bit)  |
| 4 to 7   | reserved        |

The  $Rm_3$  field is used to select integer operations.

| $Prec_3$ | $Rm_3 = 5$ |
|----------|------------|
| 0        | byte       |
| 1        | wyde       |
| 2        | tetra      |
| 3        | octa       |
| 4        | hexi       |
| 5        | reserved   |
| 6        | reserved   |
| 7        | reserved   |

Parallel (SIMD) operations

| $Prec_3$ | $Rm_3 = 6$ |
|----------|------------|
| 0        | byte       |
| 1        | wyde       |
| 2        | tetra      |
| 3        | octa       |
| 4        | hexi       |
| 5        | reserved   |
| 6        | reserved   |
| 7        | reserved   |

# MADD –Mask Add

## Synopsis

Vector mask register addition.

## Description

Two vector mask registers (Vma and Vmb) are added together and placed in the target vector register Vmt.

## Instruction Format: V2

## Operation

$$Vmt = Vma + Vmb$$

**Execution Units:** ALUs

## MAND – Bitwise Mask And

### Synopsis

Vector mask register bitwise and.  $Vmt = Vma \& Vmb$

### Description

Two vector mask registers (Vma and Vmb) are bitwise and'ed together and placed in the target vector register Vmt.

### Instruction Format: V2

### Operation

$Vmt = Vma \& Vmb$

### Execution Units: ALUs

## MEOR – Bitwise Mask Exclusive Or

### Synopsis

Vector mask register bitwise exclusive or.

### Description

Two vector mask registers (Vma and Vmb) are bitwise eor'ed together and placed in the target vector register Vmt.

### Instruction Format: V2

### Operation

$Vmt = Vma \wedge Vmb$

### Execution Units: ALUs

## MFILL –Mask Fill

### Synopsis

Fill vector mask register with bits.

### Description

The first Rs1 bits of the vector mask register (Rd) are set to one. The remaining bits of the mask register are set to zero.

### Instruction Format: V1

### Operation

$$Vmt = 0$$

for  $x = 0$  to VLMAX

if  $(x < Rs1)$   $Vmt[x] = 1$

### Execution Units: ALUs

## MFIRST – Find First Set Bit

### Synopsis

### Description

The position of the first bit set in the mask register is copied to the target register. If no bits are set the value is 128. The search begins at the least significant bit and proceeds to the most significant bit.

### Operation

$$Rd = \text{first set bit number of } (Vm)$$

### Exceptions: none

### Execution Units: ALUs

## MLAST – Find Last Set Bit

### Synopsis

### Description

The position of the last bit set in the mask register is copied to the target register. If no bits are set the value is 128. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

### Operation

$Rd = \text{first set bit number of } (Vm)$

**Exceptions:** none

**Execution Units:** ALUs

## MOR – Bitwise Mask Or

### Synopsis

Vector mask register bitwise and.  $Vmt = Vma \mid Vmb$

### Description

Two vector mask registers (Vma and Vmb) are bitwise or'ed together and placed in the target vector register Vmt.

**Instruction Format:** V2

### Operation

$Vmt = Vma \mid Vmb$

**Execution Units:** ALUs

## MPOP – Mask Population Count

### Synopsis

### Description

A count of the number of bits set in the mask register is copied to the target register.

### Operation

$Rd = \text{population count}(Vm)$

**Exceptions:** none

**Execution Units:** ALUs

## MSHL – Shift Left Mask

### Synopsis

Shift a vector mask register to the left.

### Description

A vector mask register (Vma) is shifted to the left by a constant amount and placed in the target vector register Vmt. Low order bits are filled with zeros.

### Instruction Format: V2

### Operation

$$Vmt = Vma \ll amt$$

### Execution Units: ALUs

## MSHR – Shift Right Mask

### Synopsis

Shift a vector mask register to the right.

### Description

A vector mask register (Vma) is shifted to the right by a constant amount and placed in the target vector register Vmt. High order bits are filled with zeros.

### Instruction Format: V2

### Operation

$$Vmt = Vma \gg amt$$

### Execution Units: ALUs

## MTM – Move to Mask

### Synopsis

Move a general-purpose register to a mask register.

### Description

Move a general-purpose register to a mask register.

### Instruction Format: M2

### Operation

$$Vmt = Rs1$$

### Execution Units: ALUs



# MTVL – Move to Vector Length

## Synopsis

Move a general-purpose register to the vector length register.

## Description

Move a general-purpose register to the vector length register.

## Instruction Format: M2

## Operation

$$VL = Rs1$$

**Execution Units:** ALUs

## V2BITS

### Synopsis

Convert Boolean vector to bits.

### Description

The least significant bit of each vector element is copied to the corresponding bit in the target register.

### Operation

For  $x = 0$  to  $VL-1$

$$Rt[x] = Va[x].LSB$$

**Exceptions:** none

**Execution Units:** ALUs

## VACC - Accumulate

### Synopsis

Register accumulation.  $Rd = Rs1 + Rs2$

### Description

A vector register ( $Rs1$ ) and scalar register ( $Rs2$ ) are added together and placed in the target scalar register  $Rd$ .  $Rs2$  and  $Rd$  may be the same register which results in an accumulation of the values in the vector register.

**Instruction Format:** V2

### Operation

for  $x = 0$  to  $VL - 1$

$$\text{if } (Vm[x]) \text{ } Rd = Rs1[x] + Rs2$$

### Example

```
ldi    r1,#0           ; clear results
vfmul.s v1,v2,v3        ; multiply inputs (v2) times weights (v3)
vfacc.s r1,v1,r1         ; accumulate results
fadd.s  r1,r1,r2         ; add bias (r2 = bias amount)
fsigmoid.s    r1,r1      ; compute sigmoid
```

# VBITS2V

## Synopsis

Convert bits to Boolean vector.

## Description

Bits from a general register are copied to the corresponding vector target register.

## Operation

For  $x = 0$  to  $VL-1$

if ( $Vm[x]$ )  $Vt[x] = Ra[x]$

else if ( $z$ )  $Vt[x] = 0$

else  $Vt[x] = Vt[x]$

**Exceptions:** none

**Execution Units:** ALUs

# VCMPRSS – Compress Vector

## Synopsis

Vector compression.

## Description

Selected elements from vector register Rs1 are copied to elements of vector register Rd guided by a vector mask register.

## Instruction Format: V1

## Operation

$y = 0$

$Rd = 0$

for  $x = 0$  to  $VL - 1$

    if ( $Vm[x]$ )

$Rd.element[y] = Rs1.element[x]$

$y = y + 1$



## Glossary

### Burst Access

A burst access is a number of bus accesses that occur rapidly in a row in a known sequence. If hardware supports burst access the cycle time for access to the device is drastically reduced. For instance dynamic RAM memory access is really fast for sequential burst access, and somewhat slower for random access.

### BTB

An acronym for Branch Target Buffer. The branch target buffer is used to improve the performance of a processing core. The BTB is a table that stores the branch target from previously executed branch instructions. A typical table may contain 1024 entries. The table is typically indexed by part of the branch address. Since the target address of a branch type instruction may not be known at fetch time, the address is speculated to be the address in the branch target buffer. This allows the machine to fetch instructions in a continuous fashion without pipeline bubbles. In many cases the calculated branch address from a previously executed instruction remains the same the next time the same instruction is executed. If the address from the BTB turns out to be incorrect, then the machine will have to flush the instruction queue or pipeline and begin fetching instructions from the correct address.

### FPGA

An acronym for Field Programmable Gate Array. FPGA's consist of a large number of small RAM tables, flip-flops and other logic. These are all connected together with a programmable connection network. FPGA's are 'in the field' programmable, and usually re-programmable. An FPGA's re-programmability is typically RAM based. They are often used with configuration PROM's so they may be loaded to perform specific functions.

### HDL

An acronym that stands for 'Hardware Description Language'. A hardware description language is used to describe hardware constructs at a high level.

### Instruction Bundle

A group of instructions. It is sometimes required to group instructions together into bundle. For instance, all instructions in a bundle may be executed simultaneously on a processor as a unit. Instructions may also need to be grouped if they are oddball in size for example 41 bits, so that they can be fit evenly into memory. Typically, a bundle has some bits that are global to the bundle, such as template bits, in addition to the encoded instructions.

### Instruction Pointers

A processor register dedicated to addressing instructions in memory. It is also often called a program counter. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but

instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

## ISA

An acronym for Instruction Set Architecture. The group of instructions that an architecture supports. ISA's are sometimes categorized at extreme edges as RISC or CISC. FT64 falls somewhere in between with features of both RISC and CISC architectures.

## Linear Address

A linear address is the resulting address from a virtual address after segmentation has been applied.

## Physical Address

A physical address is the final address seen by the memory system after both segmentation and paging have been applied to a virtual address. One can think of a physical address as one that is “physically” wired to the memory.

## Program Counter

A processor register dedicated to addressing instructions in memory. It is also often and perhaps more aptly called an instruction pointer. The program counter got its name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

## ROB

An acronym for ReOrder Buffer. The re-order buffer allows instructions to execute out of order yet update the machine's state in order by tracking instruction state and variables. In FT64 the re-order buffer is a circular queue with a head and tail pointers. Instructions at the head are committed if done to the machine's state then the head advanced. New instructions are queued at the buffer's tail as long as there is room in the queue. Instructions in the queue may be processed out of the order that they entered the queue in depending on the availability of resources (register values and functional units).

## RSB

An acronym that stands for return stack buffer. A buffer of addresses used to predict the return address which increases processor performance. The RSB is usually small, typically 16 entries. When a return instruction is detected at time of fetch the RSB is accessed to determine the address of the next instruction to fetch. Predicting the return address allows the processing core to continuously fetch instructions in a speculative fashion without bubbles in the pipeline. The return address in the RSB may turn out to be detected as incorrect during execution of the return instruction, in which case the pipeline

or instruction queue will need to be flushed and instructions fetched from the proper address.

## **SIMD**

An acronym that stands for ‘Single Instruction Multiple Data’. SIMD instructions are usually implemented with extra wide registers. The registers contain multiple data items, such as a 128-bit register containing four 32-bit numbers. The same instruction is applied to all the data items in the register at the same time. For some applications SIMD instructions can enhance performance considerably.

## **Stack Pointer**

A processor register dedicated to addressing stack memory. Sometimes this register is assigned by convention from the general register pool. This register may also sometimes index into a small dedicated stack memory that is not part of the main memory system. Sometimes machines have multiple stack pointers for different purposes, but they all work on the idea of a stack. For instance, in Forth machines there are typically two stacks, one for data and one for return addresses.

## **TLB**

TLB stands for translation look-aside buffer. This buffer is used to store address translations for fast memory access in a system with an mmu capable of performing address translations.

## **Vector Length (VL register)**

The vector length register controls the maximum number of elements of a vector that are processed. The vector length register may not be set to a value greater than the number of elements supported by hardware. Vector registers often contain more elements than are required by program code. It would be wasteful to process all elements when only a few are needed. To improve the processing performance only the elements up to the vector length are examined.

## **Vector Mask (VM)**

A vector mask is used to restrict which elements of a vector are processed during a vector operation. A one bit in a mask register enables the processing for that element, a zero bit disables it. The mask register is commonly set using a vector set operation.



## Miscellaneous

### Reference Material

Below is a short list of some of the reading material the author has studied. The author has downloaded a fair number of documents on computer architecture from the web. Too many to list.

Modern Processor Design Fundamentals of Superscalar Processors by John Paul Shen, Mikko H. Lipasti. Waveland Press, Inc.

Computer Architecture A Quantitative Approach, Second Edition, by John L Hennessy & David Patterson, published by Morgan Kaufman Publishers, Inc. San Francisco, California is a good book on computer architecture. There is a newer edition of the book available.

Memory Systems Cache, DRAM, Disk by Bruce Jacob, Spencer W. Ng., David T. Wang, Samuel Rodriguez, Morgan Kaufman Publishers

PowerPC Microprocessor Developer's Guide, SAMS publishing. 201 West 103<sup>rd</sup> Street, Indianapolis, Indiana, 46290

80386/80486 Programming Guide by Ross P. Nelson, Microsoft Press

Programming the 286, C. Vieillefond, SYBEX, 2021 Challenger Drive #100, Alameda, CA 94501

Tech. Report UMD-SCA-2000-02 ENEE 446: Digital Computer Design — An Out-of-Order RiSC-16

Programming the 65C816, David Eyes and Ron Lichty, Western Design Centre Inc.

Microprocessor Manuals from Motorola, and Intel,

The SPARC Architecture Manual Version 8, SPARC International Inc, 535 Middlefield Road. Suite 210 Menlo Park California, CA 94025

The SPARC Architecture Manual Version 9, SPARC International Inc, San Jose California, PTR Prentice Hall, Englewood Cliffs, New Jersey, 07632

The MMIX processor: <http://mmix.cs.hm.edu/doc/instructions-en.html>

RISCV 2.0 Spec, Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović CS Division, EECS Department, University of California, Berkeley {waterman|yunsup|pattsrn|krste}@eecs.berkeley.edu

### Trademarks

IBM® is a registered trademark of International Business Machines Corporation. Intel® is a registered trademark of Intel Corporation. HP® is a registered trademark of Hewlett-Packard Development Company. "SPARC® is a registered trademark of SPARC International, Inc.

## WISHBONE Compatibility Datasheet

The NVIO3 core may be directly interfaced to a WISHBONE compatible bus.

|  |   |                 |
|--|---|-----------------|
| WISHBONE Datasheet   |   |                 |
| WISHBONE SoC Architecture Specification, Revision B.3                    |   |                 |
|  |   |                 |
| Description:   | Specifications:   |                 |
| General Description:   | Central processing unit (CPU core)  |                 |
| Supported Cycles:  | MASTER, READ / WRITE<br>MASTER, READ-MODIFY-WRITE<br>MASTER, BLOCK READ / WRITE, BURST READ (FIXED ADDRESS) |                 |
| Data port, size:   | 128 bit   |                 |
| Data port, granularity:  | 8 bit   |                 |
| Data port, maximum operand size:   | 128 bit   |                 |
| Data transfer ordering:  | Little Endian   |                 |
| Data transfer sequencing   | any (undefined)   |                 |
| Clock frequency constraints:   |   |                 |
| Supported signal list and cross reference to equivalent WISHBONE signals | Signal Name:  | WISHBONE Equiv. |
|  | ack_i   | ACK_I           |
|  | adr_o(31:0)   | ADR_O()         |
|  | clk_i   | CLK_I           |
|  | dat_i(127:0)  | DAT_I()         |
|  | dat_o(127:0)  | DAT_O()         |
|  | cyc_o   | CYC_O           |
|  | stb_o   | STB_O           |
|  | wr_o  | WE_O            |
|  | sel_o(15:0)   | SEL_O           |
|  | cti_o(2:0)  | CTI_O           |
|  | bte_o(1:0)  | BTE_O           |
| Special Requirements:  |   |                 |