# FT64

# Overview

FT64 is a two-way superscalar processing core capable of executing up to two instructions per clock cycle. The core features register renaming to avoid data hazards. The core has the following features:

* 32 general purpose scalar registers, 64 register sets
* 32 general purpose vector registers, length 63
* register renaming
* speculative loading
* 32 bit fixed instruction format
* 64 bit data width
* powerful branch prediction with target buffer (BTB)
* return address prediction (RSB)
* bus interface unit
* instruction and data caches
* Vector and SIMD operations
* dual ALU’s, one flow control unit, one memory unit, one floating point unit

## History

FT64 is a work-in-progress beginning in July 2017. FT64 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. FT64 is the author's fifth attempt at a 64 bit ISA. Other attempts including Raptor64, Thor, FISA64, and DSD9. The author has tried to be innovative with this design borrowing ideas from a number of other processing cores. Berkeley’s RiSC-V has had an influence on this core.

## Goals

One of the primary goals for the development of this core was the implementation of a register renaming mechanism. The author also wanted a stream-lined core as a starting place.

Implementing many features of the Thor core using a fixed 32 bit instruction set.

Easy implementation of a compiler.

Eventual implementation as a four-way superscalar processing core.

## Register Set

There are 64 sets of 32 general purpose registers in the architecture. R0 always has the value zero in all register sets.

|  |  |  |
| --- | --- | --- |
| Register | Description / Suggested Usage | Saver |
| r0 | always reads as zero |  |
| r1-r2 | return values / exception | caller |
| r3-r10 | temporaries | caller |
| r11-r17 | register variables | callee |
| r18-r22 | function arguments | caller |
| r23 | assembler usage |  |
| r24 | type number / function argument | caller |
| r25 | class pointer / function argument | caller |
| r26 | thread pointer | callee |
| r27 | global pointer |  |
| r28 | exception link register | caller |
| r29 | return address / link register | caller |
| r30 | base / frame pointer | callee |
| r31 | stack pointer (hardware) | callee |

The ISA supports up to 32 vector registers of length 63.

|  |  |
| --- | --- |
| Register |  |
| v0 to v31 | general purpose vector registers |
| vm0 to vm7 | vector mask registers |

The register file has six read ports and two write ports.

Notes:

The register set is implemented with block ram resources in the FPGA. In order to get byte write strobes for the registers it was possible to accommodate a large number of registers. Elucidating, the block rams in use provided 4096 eight-bit wide registers per block ram. Regardless of the number of registers actually used there was still a provision for 4096. All these available registers were put to good use as multiple register sets and vectored registers.

The register set currently selected is determined by the rs field in the machine status register (0x044).

Internally to the core a single register file is in use that uses a 12-bit register code:

|  |  |  |
| --- | --- | --- |
| 11 6 | 5 | 4 0 |
| Register Set | 0 | General Purpose Register number |
| Vector element | 1 | Vector register number |

To conserve hardware which would otherwise be quite large, the bypassing logic looks at only the six least significant bits of the register code for bypassing purposes. This allows it to differentiate between different general purpose registers and vector registers. This meets bypass logic requirements in most circumstances.

The core does not provide bypass logic between different elements of the same vector register. It only provides bypassing at the vector register number level. Normally this is not a problem because vector elements are processed independently.

Similarly, the core does not provide bypassing between register sets of the general purpose registers. Switching the register set should be followed by a synchronization operation to ensure contents of the previous instructions are updated before the new use.

There are only 63 usable elements to each vector register. Register codes for the 64th element are used to access the vector mask registers.

|  |  |  |  |
| --- | --- | --- | --- |
| 11 6 | 5 | 4 3 | 2 0 |
| 63 | 1 | 0 | mask register number |

This is hidden from the ISA and may be implemented differently in the future.

On reset register set #8 is selected.

## Program Counter

The program counter identifies which instruction to execute. The program counter increments by four with the least significant two bits always zero. The increment may be overridden using one of the flow control instructions. The program counter addresses 32 bit instruction parcels.

|  |  |
| --- | --- |
| 63 2 | 1 0 |
| Address[63..2] | 02 |

## Caches

The core has both instruction and data caches in order to improve performance.

The instruction cache is a two level cache (L1, L2) allowing better performance. The first level cache is fully associative, the second level cache is four-way set associative. L1 is 2kB in size and made from distributed ram in order to get single cycle performance. L1 is organized as 64 lines of 32 bytes. L2 is 16kB in size implemented with block ram. L2 is organized as 512 lines of 32 bytes. The instruction cache is dual ported to allow two instructions to be fetched at one time.

The data cache is organized as 512 lines of 32 bytes (16kB) and implemented with block ram. Access to the data cache is multicycle. The data cache has three read ports allowing three load operations to be in progress at the same time. Stores write through to memory. There is only a single write port on the data cache.

### Uncached Data Area

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

## 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, unless specified otherwise in the branch instruction.

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 it’s usefulness.

## 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 can 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 16 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.

## 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.

## Operating Levels

The core has eight 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.

|  |  |  |
| --- | --- | --- |
| Operating Level | Privilege Level | Moniker |
| 7 | 7 to 255 | user |
| 6 | 6 | supervisor |
| 5 | 5 | supervisor |
| 4 | 4 | supervisor |
| 3 | 3 | supervisor |
| 2 | 2 | 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 mode zero. Privilege level one is assigned to operating level one. Privilege levels 2 to 6 are assigned to their corresponding operating level. The remaining privilege levels are assigned to operating level seven.

## Control and Status Registers

### Control Register Zero (CSR #000)

This register contains a bit to enable protected mode.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 | 62 |  |  | 33 | 32 | 30 |  | 1514 | 13 8 | 7 1 | 0 |
| D | ~ |  |  | ~ | bpe | dce |  | 0 | ~ |  | Pe |

D: debug mode status. this bit is set during an interrupt routine if the processor was in debug mode when the interrupt occurred.

PE: Protected Mode enable: 1 = enabled, 0 = disabled.

DCE: data cache enable: 1=enabled, 0 = disabled

bpe: branch predictor enable: 1=enabled, 0=disabled

Disabling the data cache is useful for some codes with large data sets to prevent cache loading of values that are used infrequently. The instruction cache may not be disabled.

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 (unless determined otherwise by the instruction). No entries will be updated in the branch history table if the branch predictor is disabled.

This register supports bit set / clear CSR instructions.

### 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.

### TICK (0x002)

This register contains a tick count of the number of clock cycles that have passed since the last reset.

### 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 37 | 36 | 35 | 34 | 33 | 32 | 31 5 | 4 | 3 | 2 | 1 | 0 |
|  | DIV | MUL | ASL | SUB | ADD |  | DIV | MUL | ASL | SUB | ADD |

### 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.

### 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 EPC 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.

### 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.

### SBL (CSR 0x00E)

The SBL register contains the address representing the lower bound of the stack. If an address is formed using one of the stack indexing registers (stack pointer r31 or base pointer r30) is lower than the SBL a stack fault occurs. This represents a stack overflow condition.

### SBU (CSR 0x00F)

The SBU register contains the address representing the upper bound of the stack. If an address is formed using one of the stack indexing registers (stack pointer r31 or base pointer r30) is higher than the SBU a stack fault occurs. This represents a stack underflow condition.

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

### DBADx (CSR 0x018 to 0x01B) Debug Address Register

These registers contain addresses of instruction or data breakpoints.

|  |
| --- |
| 63 0 |
| Address 63..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 | half | | 11 | all but the three LSB’s should match | word | |  |  |
| 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 |
| 63 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.

|  |
| --- |
| 63 0 |
| Value 63..0 |

### TVEC (0x030 to 0x037)

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. For the other registers the two low order bits of the address must be zero in order to keep the program counter aligned on a half-word address. TVEC[1] to TVEC[7] are used by the REX instruction.

### 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 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 is set to seven masking all interrupts on stack underflow.

### OL\_STACK (0x041)

This register contains the operating level stack. 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 is set to zero which will select the machine operating level on stack underflow.

### PL\_STACK (0x042)

This register contains the privilege level stack. 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 zero which will select privilege level zero on stack underflow.

### 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 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.

### STATUS (0x044)

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

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 63 | 6261 | 60 56 | 55 | 5452 | 5150 | 4948 | 47 20 | 19 14 | 13 6 | 5 3 | 2 0 |
| SD1 | ~2 | VM5 | MPRV1 | ~3 | XS2 | FS2 | ~28 | RS6 | PL8 | OL3 | IM3 |

#### VM5

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.

|  |  |  |
| --- | --- | --- |
| 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 |  |

#### MPRV

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).

#### FS2

These two bits can be used to keep track of the floating point register state.

#### XS2

These two bits can be used to keep track of an additional core extension state.

### EPC (0x048 to 0x4F)

This sets of registers contains the interrupt or exception stack of the program counter register. The top of the stack is register 0x48. When an interrupt or exception occurs register 0x48 to 0x4E are copied to the next register and the program counter is placed into register 0x48. When an RTI instruction is executed the program counter is loaded from register 0x048 and registers 0x048 to 0x047 are loaded with the next register. Register 0x04F 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 in order to execute code which may change at run-time.

### TIME (0x7E0)

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 32 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 32 bits represent the fraction of one second. The upper 32 bits represent seconds passed. For example if the tm\_clk\_i frequency is 100MHz the low order 32 bits should count from 1 to 100,000,000 then cycle back to 1 again. When the low order 32 bits cycle back to 1 again, the upper 32 bits of the register is incremented. The upper 32 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 (0x7E1)

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

### INFO (0x7F0 to 0x7FF)

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

# 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.

## 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.

For a hardware interrupt the register set is set to the level of the hardware interrupt (0 to 7). For a software exception register set #8 is selected. Individual registers from alternate register sets may be selected with the move instruction.

## Exception Stack

The program counter 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 eight 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 four for instance and an interrupt occurs vector table address number four 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 interrupt |
| 1 | $FFFC0020 |  |
| 2 | $FFFC0040 |  |
| 3 | $FFFC0060 |  |
| 4 | $FFFC0080 |  |
| 5 | $FFFC00A0 |  |
| 6 | $FFFC00C0 |  |
| 7 | $FFFC00E0 | handler for operating level seven interrupt |

## Reset

The core begins executing instructions at address $FFFC0100. All registers are in an undefined state. Register set #8 is selected.

## Exception Cause Codes

The following table outlines the cause code for a given purpose. These codes are specific to FT64. 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 |  |
| 0 |  |  |  |  |
| 1 |  |  |  |  |
| 2 |  |  | FMTK Scheduler |  |
| 432 |  | e |  |  |
| 433 | KRST | e | Keyboard reset interrupt |  |
| 434 | MSI | e | Millisecond Interrupt |  |
| 435 | TICK | e | FMTK Tick Interrupt |  |
| … |  |  |  |  |
| 463 | KBD | e | Keyboard interrupt |  |
| 480 | SSM | x | single step |  |
| 481 | DBG | x | debug exception |  |
| 482 | TGT | x | call target exception |  |
| 483 | MEM | x | memory fault |  |
| 484 | IADR | x | bad instruction address |  |
| 485 | UNIMP | x | unimplemented instruction |  |
| 486 | FLT | x | floating point exception |  |
| 487 | CHK | x | bounds check exception |  |
| 488 | DBZ | x | divide by zero |  |
| 489 | OFL | x | overflow |  |
| 493 | FLT | x | floating point exception |  |
| 497 | EXF | x | Executable fault |  |
| 498 | DWF | x | Data write fault |  |
| 499 | DRF | x | data read fault |  |
| 500 | SGB | x | segment bounds violation |  |
| 501 | PRIV | x | privilege level violation |  |
|  |  |  |  |  |
|  |  |  |  |  |
| 504 | STK | x | stack fault |  |
| 505 | CPF | x | code page fault |  |
| 506 | DPF | x | data page fault |  |
| 508 | DBE | x | data bus error |  |
| 509 | IBE | x | instruction bus error |  |
| 510 | NMI | x | Non-maskable interrupt |  |
|  |  |  |  |  |

# Simplified Paged Memory Management Unit

## Overview

The memory management unit is a simplified paged memory management unit. Memory management by the MMU includes virtual to physical address mapping and read/write/execute permissions. The MMU divides memory into 64kB or 4MiB pages depending on the setting in PCR2.

64kiB pages

Processor address bits 16 to 25 are used as a ten bit index into a mapping table to find the physical page. The MMU remaps the ten address bits into a sixteen bit value used as address bits 16 to 31 when accessing a physical address. The lower sixteen bits of the address pass through the MMU unchanged. The maximum amount of memory that may be mapped in the MMU is 64MiB per map out of a pool of 4GiB. Addresses with the most significant six bits set are not mapped.

4MiB pages

Some tasks require a lot of memory and a 64MB map isn’t sufficient. For instance, while in machine mode the core requires access to the entire address range. A memory page size of 4MiB may be selected by setting the bit corresponding to the memory map in PCR2.

Processor address bits 22 to 31 are used as a ten bit index into a mapping table to find the physical page. The MMU remaps the ten address bits into a ten bit value used as address bits 22 to 31 when accessing a physical address. The lower 22 bits of the address pass through the MMU unchanged. The maximum amount of memory that may be mapped in the MMU is 4GiB per map out of a pool of 4GiB. Addresses with the most significant six bits set are not mapped.

## Map Tables

The mapping tables for memory management are stored directly in the MMU rather than being stored in main memory as is commonly done. The MMU supports up to 64 independent mapping tables. Only a single mapping table may be active at one time. The active mapping table is set in the paging control register (CSR #3) bits 0 to 5 – called the operate key. Mapping tables may be shared between tasks.

## Map Caching / TLB

There isn’t a need for a TLB or ATC as the entire mapping table is contained in the MMU. A TLB isn’t required. Address mapping is still only two cycles.

## Operate Key

The operate key controls which mapping table is actively mapping the memory space. The operate key is located in CSR #3 bits 0 to 5. The operate key is similar to an ASID (address space identifier). The operate key is also used as part of the cores cache tags. When the operate key changes due to a task switch, the cache does not have to be invalidated.

## Access Key

The MMU mapping tables are present at I/O address $FFDC4000 to $FFDC4FFF. All the mapping tables share the same I/O space. Only one mapping table is visible in the address space at one time. Which table is visible is controlled by an access key. The access key is located in the paging control register (CSR #3) bits 8 to 13.

## Address Pass-through

Addresses pass through the MMU unaltered until the mapping enable bit is set. Until mapping is enabled, the physical address will match the virtual address. Additionally address bits 0 to 15 pass through the MMU unaltered.

## Mapping Table Layout

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | D20 | D19 | D18 | D17 | D16 | D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |  |
| 000 | S1 | S0 | W | R | X | PA31 | PA30 | PA29 | PA28 | PA27 | PA26 | PA25 | PA24 | PA23 | PA22 | PA21 | PA20 | PA19 | PA18 | PA17 | PA16 |  |
| 004 | S1 | S0 | W | R | X | PA31 | PA30 | PA29 | PA28 | PA27 | PA26 | PA25 | PA24 | PA23 | PA22 | PA21 | PA20 | PA19 | PA18 | PA17 | PA16 |  |
|  |  |  | … | | | | | | | | | | | | | | | | | | |  |
| FFC | S1 | S0 | W | R | X | PA31 | PA30 | PA29 | PA28 | PA27 | PA26 | PA25 | PA24 | PA23 | PA22 | PA21 | PA20 | PA19 | PA18 | PA17 | PA16 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

PAnn = physical address

X = executable page indicator.

W = writeable data page indicator.

R = readable data page indicator.

Note the low order six bits are not used for 4MiB pages.

S1,S0 = two bits for program use

## PCR- Paging Control Register Layout

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 | 30 14 | 13 8 | 7 6 | 5 0 |
| PE | ~18 | AKey6 | ~ | OKey6 |

PE = Paging Enable (1=enabled, 0 = disabled)

AKey = Access Key

OKey = Operate Key

## PCR2 – Page Size

This register controls the memory page size. Each bit in the register corresponds to a memory map. Memory may be paged in either 64kiB or 4MiB pages. All pages in a map have the same size.

## Latency

The address map operation when enabled has two cycles of latency. In the case of instructions address translation only takes place on a cache miss when the cache needs to be loaded from main memory.

# Instruction Set Description

## Formats

Instructions have a fixed 32 bit format. There are only a handful of different instruction formats.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Immed16 | | | | | | | Rt5 | Ra5 | Opcode6 | RI |
| Funct6 | | | ~3 | Sz2 | Rt5 | | Rb5 | Ra5 | Opcode6 | RR |
| Op2 | OL3 | Regno11 | | | | Rt5 | | Ra5 | 0Eh6 | CSR |

## Operation Sizes

Many instructions have an option to process data in sub-word data sizes including bytes, chars, and half-words. 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 64 bit operands as multiple independent lanes of data depending on the size selected. For a half-word size the operands are treated as two independent 32 bit operands. For a character size the operands are treated as four independent 16 bit operands. SIMD operations are selected by setting the parallel operation bit in the instruction (the most significant bit of the size field).

## Arithmetic Operations

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

## Logical Operations

Logical operations include bitwise and, or, and exclusive or.

## Memory Operations

Memory operations include loads and stores of bytes, words or half-words. There isn’t yet a full complement of memory operations in order to keep the size of the core smaller. Notably missing are instructions to load / store 16 bit quantities. The core can perform loads and stores using indexed addressing.

## Control Flow Instructions

Control flow instructions include call, return, jumps and branches, breakpoint and return instructions.

## Clock cycles

The clock cycles indicated are only approximate. The core hasn’t under gone significant timing measurements.

## ABS – Absolute Value

**Description:**

This instruction takes the absolute value of a register and places the result in a target register.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 016 | ~2 | Sz3 | 45 | Rt5 | Ra5 | 02h6 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

If Ra < 0

Rt = -Ra

else

Rt = Ra

Exceptions: none

Notes:

|  |  |
| --- | --- |
| Sz3 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |
| 4 | Byte Parallel |
| 5 | Char Parallel |
| 6 | Half Parallel |
| 7 | Word |

## 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.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 04h6 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 046 | ~1 | Ov | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |
| --- | --- |
| Ov |  |
| 0 | no overflow |
| 1 | overflow exception if overflow occurred and enabled in AEC |

Overflow works properly only on 64 bit values.

Instruction Format:

This format performs the ‘add’ operation with an immediate value to one of four quadrants of the target register. It may be used to build a 64 bit constant in a register. The immediate is sign extended to 64 bits then shifted by 0, 16, 32 or 48 bits to the left.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Immed16 | Rt5 | 13 | Q2 | 1Ah6 |

|  |  |
| --- | --- |
| Q2 | Bits |
| 0 | 0 to 15 |
| 1 | 16 to 31 |
| 2 | 32 to 47 |
| 3 | 48 to 63 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

Exceptions:

The immediate form of the instruction will not cause an exception. The registered form of the instruction may cause an overflow exception if enabled in the AEC register.

Notes:

For sub-word forms the part of the register updated corresponds to the size selected. For instance, if a byte operation is specified then only the low order eight bits of the target register is updated, the remaining bits hold their current value. For parallel operation forms the registers are treated as if they were a group of registers corresponding to the size selected. And the same operation is performed on each part of the register. For parallel forms the entire register is updated.

|  |  |
| --- | --- |
| Sz3 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |
| 4 | Byte Parallel |
| 5 | Char Parallel |
| 6 | Half Parallel |
| 7 | Word |

## AMO – Atomic Memory Operation

Description:

The atomic memory operations read from memory addressed by the Ra register and store the value in Rt. As a second step the value from memory is combined with the value in register Rb according to one of the available functions then stored back into the memory addressed by Ra.

Instruction Format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Funct6 | A | R | Sz3 | Rt5 | Rb5 | Ra5 | 2Fh6 |

Instruction Format (immediate operand):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Funct6 | A | R | Sz3 | Imm5 | Rt5 | Ra5 | 2Fh6 |

|  |  |  |  |
| --- | --- | --- | --- |
| Funct6 | Mnemonic | Operation Performed |  |
| 01 | swap | swap | memory[Ra] = Rb |
| 04 | add | addition | memory[Ra] = memory[Ra] + Rb |
| 08 | and | bitwise and | memory[Ra] = memory[Ra] & Rb |
| 09 | or | bitwise or | memory[Ra] = memory[Ra] | Rb |
| 0A | xor | bitwise exclusive or | memory[Ra] = memory[Ra] ^ Rb |
| 0C | shl | shift left | memory[Ra] = memory[Ra] << Rb |
| 0D | shr | shift right | memory[Ra] = memory[Ra] >> Rb |
|  |  |  |  |
|  |  |  |  |
| 1C | min | minimum | memory[Ra] = memory[Ra] < Rb ? memory[Ra] : Rb |
| 1D | max | maximum | memory[Ra] = memory[Ra] >Rb ? memory[Ra] : Rb |
| 1E | minu | minimum unsigned | memory[Ra] = memory[Ra] < Rb ? memory[Ra] : Rb |
| 1F | maxu | maximum unsigned | memory[Ra] = memory[Ra] > Rb ? memory[Ra] : Rb |
| 20 | swapi | swap | memory[Ra] = imm |
| 24 | addi | addition | memory[Ra] = memory[Ra] + imm |
| 28 | andi | bitwise and | memory[Ra] = memory[Ra] & imm |
| 29 | ori | bitwise or | memory[Ra] = memory[Ra] | imm |
| 2A | xori | bitwise exclusive or | memory[Ra] = memory[Ra] ^ imm |
| 2C | shli | shift left | memory[Ra] = memory[Ra] << imm |
| 2D | shri | shift right | memory[Ra] = memory[Ra] >> imm |
|  |  |  |  |
|  |  |  |  |
| 3C | mini | minimum | memory[Ra] = memory[Ra] < imm ? memory[Ra] : imm |
| 3D | maxi | maximum | memory[Ra] = memory[Ra] > imm ? memory[Ra] : imm |
| 3E | minui | minimum | memory[Ra] = memory[Ra] < imm ? memory[Ra] : imm |
| 3F | maxui | maximum | memory[Ra] = memory[Ra] > imm ? memory[Ra] : imm |

|  |  |
| --- | --- |
| Sz2 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |

Acquire and release bits determine the ordering of memory operations.

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

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

All combinations of A, R are allowed.

## AND – Bitwise And

Description:

Perform a bitwise ‘and’ operation between operands.

Instruction Format:

The immediate value is sign extended on the left before use.

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 08h6 |

Rt = Ra & Rb

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 086 | ~2 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Instruction Format:

This format performs the ‘and’ operation with an immediate value to one of four quadrants of the target register. It may be used to build a 64 bit constant in a register. The immediate is shifted to the left by 0, 16, 32, or 48 bits then one extended on both the left and right sides. Note this instruction will only mask out bits in the selected quadrant.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Immed16 | Rt5 | 23 | Q2 | 3Bh6 |

|  |  |
| --- | --- |
| Q2 | Bits |
| 0 | 0 to 15 |
| 1 | 16 to 31 |
| 2 | 32 to 47 |
| 3 | 48 to 63 |

Clock Cycles: 1

**Execution Units: All** ALUs

Exceptions: none

## ASL – Arithmetic Shift Left

Description:

Bits from the source register Ra are shifted left by the amount in register Rb 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.

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Func6 | 24 | E | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Func6 | Ah4 | Imm6 | Rt5 | Ra5 | 02h6 |

|  |  |  |
| --- | --- | --- |
| Func6 | Op Size | If E set |
| 0Fh | word | word |
| 1Fh | byte | byte parallel |
| 2Fh | char | char parallel |
| 3Fh | half | half parallel |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

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 Ra are shifted right by the amount in register Rb or an immediate value. The sign bit is shifted into the most significant bits.

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Func6 | 34 | E | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Func6 | Bh4 | Imm6 | Rt5 | Ra5 | 02h6 |

|  |  |  |
| --- | --- | --- |
| Func6 | Op Size | If E set |
| 0Fh | word | word |
| 1Fh | byte | byte parallel |
| 2Fh | char | char parallel |
| 3Fh | half | half parallel |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

## BBC –Branch if Bit Clear

Description:

If the specified bit in a register is clear then an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 11 | | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 13 | Bitno6 | Ra5 | 13h5 | D1 |

Operation:

if (Ra[bitno]=0)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

Clock Cycles: 1 with accurate prediction, otherwise 8 or more

**Execution Units:** FCU Only

Exceptions: none

## BBS –Branch if Bit Set

Description:

If the specified bit in a register is set then an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 11 | | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 03 | Bitno6 | Ra5 | 13h5 | D1 |

Operation:

if (Ra[bitno]=1)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

Clock Cycles: 1 with accurate prediction, otherwise 8 or more

**Execution Units:** FCU Only

Exceptions: none

## Bcc – Conditional Branch

Description:

If the branch condition is true, an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. The immediate value may not be extended with a prefix instruction.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | Cond4 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | Cond4 | Rc5 | Rb5 | Ra5 | 03h6 |

|  |  |  |
| --- | --- | --- |
| Cond3 | Mne. |  |
| 0 | BEQ | Ra = Rb signed |
| 1 | BNE | Ra <> Rb |
| 2 | BLT | Ra < Rb |
| 3 | BGE | Ra >= Rb |
| 4 | BLTU | Ra < Rb (unsigned) |
| 5 | BGEU | Ra >= Rb (unsigned) |
| 6 |  | reserved |
| 7 | BOR | Ra || Rb (either Ra or Rb is true) |
| 8 | FBEQ | Ra = Rb (floating point) |
| 9 | FBNE | Ra != Rb (floating point) |
| 10 | FBLT | Ra < Rb (floating point) |
| 11 | FBGE | Ra >= Rb (floating point) |
| 12 |  | reserved |
| 13 |  | reserved |
| 14 |  | reserved |
| 15 | FBUN | register Ra contains unordered floating point constant |

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

Clock Cycles:

Typically 1 with correct branch outcome and target prediction.

## BCDADD - Register-Register

**Description:**

Adds two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is an eight bit BCD number. The result is zero extended to 64 bits.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 006 | 05 | Rt5 | Rb5 | Ra5 | 02h6 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra + Rb

**Exceptions:** none

## BCDMUL - Register-Register

**Description:**

Multiplies two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is a 16 bit BCD value. The result is zero extended to 64 bits.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 006 | 25 | Rt5 | Rb5 | Ra5 | 02h6 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 Only

**Operation:**

Rt = Ra \* Rb

**Exceptions:** none

## BCDSUB - Register-Register

**Description:**

Subtracts two registers using BCD arithmetic and places the result in a target register. Only the low order byte of the register is used. The result is an eight bit BCD number. The result is zero extended to 64 bits.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 006 | 15 | Rt5 | Rb5 | Ra5 | 02h6 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = Ra - Rb

**Exceptions:** none

## BEQ –Branch if Equal

Description:

If two registers are equal an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 04 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | 04 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra <> 0)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

Clock Cycles: Typically 1 with correct branch outcome and target prediction.

## BEQI –Branch if Equal Immediate

Description:

If a register is equal to a nine bit sign extended value then an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. This instruction is useful for implementing case statements based on small values.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | Immed9 | Ra5 | 19h5 | D1 |

Operation:

if (Ra = Immediate)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

Clock Cycles: Typically 1 with correct branch outcome and target prediction.

## BFCHG – Bitfield Change

Description:

A bitfield is inverted in the target register.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 24 | Me6 | Mb6 | Rt5 | Ra5 | 22h6 |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

## BFCLR – Bitfield Clear

Description:

A btifield is cleared in the target register. This is an alternate mnemonic for the bitfield insert instruction.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 34 | Me6 | Mb6 | Rt5 | 05 | 22h6 |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

## BFEXT – Bitfield Extract

Description:

A bitfield is extracted from the source register Ra 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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 54 | Me6 | Mb6 | Rt5 | Ra5 | 22h6 |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

## BFEXTU – Bitfield Extract

Description:

A btifield is extracted from the source register Ra 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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 64 | Me6 | Mb6 | Rt5 | Ra5 | 22h6 |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

## BFINS – Bitfield Insert

Description:

A btifield is inserted into the source register Ra by shifting to the left.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 34 | Me6 | Mb6 | Rt5 | Ra5 | 22h6 |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

## BFINSI – Bitfield Insert Immediate

Description:

A bitfield is inserted into the target register Rt by shifting a constant to the left. The bitfield may not be larger than five bits. To accommodate a larger field multiple instructions can be used.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 44 | Me6 | Mb6 | Rt5 | Imm5 | 22h6 |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

## BGE –Branch if Greater or Equal

Description:

If register Ra is greater than or equal to register Rb then an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. This instruction may also be used to branch on less than or equal by swapping the registers around.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 34 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | 34 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra < 0)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## BGEU –Branch if Greater or Equal Unsigned

Description:

If register Ra is greater than or equal to register Rb then an eleven bit sign extended value is shifted left twice and added to the program counter. The values are treated as unsigned numbers. The branch is relative to the address of the instruction directly following the branch. This instruction may also be used to branch on less than or equal by swapping the registers around.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 54 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | 54 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra < 0)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## BLT –Branch if Less Than

Description:

If register Ra is less than register Rb then an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. This instruction may also be used to branch on greater than by swapping the registers around.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 24 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | 24 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra < 0)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## BLTU –Branch if Less Than Unsigned

Description:

If register Ra is less than register Rb then an eleven bit sign extended value is shifted left twice and added to the program counter. The values are treated as unsigned numbers. The branch is relative to the address of the instruction directly following the branch. This instruction may also be used to branch on greater than by swapping the registers around.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 44 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | 44 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra < 0)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## BMM – Bit Matrix Multiply

BMM Rt, Ra, Rb

Description:

The BMM instruction treats the bits of register Ra and Rb as an 8x8 bit 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.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 036 | ~3 | ~2 | Rt5 | Rb5 | Ra5 | 02h6 |

Operation:

for I = 0 to 7

for j = 0 to 7

Rt.bit[i][j] = (Ra[i][0]&Rb[0][j]) | (Ra[i][1]&Rb[1][j]) | … | (Ra[i][7]&Rb[7][j])

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

Exceptions: none

Notes:

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

## BNE –Branch if Not Equal

Description:

If two registers are unequal an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 14 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | 14 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra <> 0)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## 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 $FFFC0100. 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](#_REX_–_Redirect) instruction.

For hardware interrupts a register set is selected automatically according to the hardware interrupt level (0 to 7). For a software interrupt register set #8 is selected. Registers from alternate register sets are available with the [MOV](#_MOV_–_Move) instruction.

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

|  |  |  |
| --- | --- | --- |
| Item Stacked | CSR reg |  |
| program counter | pc\_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 |
| register set | rs\_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.

**Instruction Format:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 19 | 18 16 | 15 | 14 6 | 5 0 |
| Immed13 | L3 | S | Cause Code9 | 00h6 |

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

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

L3 = 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 Immed13 field may be used to pass constant data to the break handler.

## CACHE – Cache Command

## CACHEX –

CACHE Cmd, d(Rn)

CACHE Cmd, d(Ra + Rb \* scale)

**Description:**

This instruction commands the cache controller to perform an operation. Commands are summarized in the command table below.

Instruction Formats:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Displacement16 | | | | Cmd5 | Ra5 | 1Eh6 | CACHE Cmd,d16(Rn) |
| 1Eh6 | ~3 | Sc2 | Cmd5 | Rb5 | Ra5 | 02h6 | CACHE Cmd,d(Ra+Rb\*sc) |

**Commands:**

|  |  |  |
| --- | --- | --- |
| Cmd5 | Mne. | Operation |
| 00h |  | reserved |
| 01h |  | reserved |
| 02h | inviline | invalidate instruction cache line |
| 03h | invic | invalidate entire instruction cache (address is ignored) |
| 10h | disabledc | disable data cache |
| 11h | enabledc | enable data cache |
| 12h |  | invalidate data cache line |
| 13h | invdc | invalidate entire data cache (address is ignored) |
|  |  |  |

Operation:

Register Indirect with Displacement Form

Line = round32(sign extend(memory[displacement + Ra]))

Register-Register Form

Line = round32(sign extend(memory[Ra + Rb \* scale]))

Notes:

The displacement constant may be extended up to 64 bits.

|  |  |
| --- | --- |
| Sc2 Code | Multiply By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## CALL – Call Subroutine

Description:

Call a subroutine. This instruction is a longer address form than the JAL instruction and has the link register as an implied target for the return address.

Instruction Format:

The address of the following instruction is stored in the link register. The format shifts the address field of the instruction by two bits to the left then modifies only PC bits 0 to 27. The high order PC bits are not affected. This allows accessing a subroutine within a 256MB region of memory. Note that with the use of a mmu this address range is often sufficient.

|  |  |
| --- | --- |
| Address[27..2] | 19h6 |

The change of address takes place during the fetch stage of the core. This makes the instruction faster than other alternatives.

**Execution Units:** FCU

Clock Cycles:

## CAS – Compare and Swap

**Description:**

If the contents of the addressed memory cell is equal to the contents of CAS register then a sixty-four bit value is stored to memory from the source register Rst and Rst is set equal to one. Otherwise Rst is set to zero and the contents of the memory cell is loaded into the CAS register. The memory address is the sum of the sign extended displacement and register Ra. The compare and swap operation is an atomic operation; the bus is locked during the load and potential store operation. This operation assumes that the addressed memory location is part of the volatile region of memory and bypasses the data cache. Note that the memory system must support bus locks in order for this instruction to work as expected.

This instruction is typically used to implement semaphores. The LWR and SWC may also be used to perform a similar function where the memory system does not support bus locks, but support address reservations instead.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| Disp16 | Rst5 | Ra5 | 25h6 |

**Operation:**

if memory[Ra+displacement] = casreg

memory[Ra + displacement] = Rst

Rst = 1

else

casreg = memory [Ra + displacement]

Rst = 0

**Assembler:**

CAS Rt, displacement[Ra]

## CHK – Check Register Against Bounds

Description:

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

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| 31 16 | 15 11 | 10 6 | 5 0 |
| Immediate16 | Rb5 | Ra5 | 34h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31 26 | 25 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 34h6 | ~5 | Rc5 | Rb5 | Ra5 | 02h5 |

Clock Cycles: 1

Exceptions: bounds check

Notes:

## CLI – Clear Interrupt Mask

Description:

The interrupt level mask is set to zero enabling all interrupts. This is an alternate mnemonic for the SEI instruction where the mask level to set is set to zero by the assembler.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 306 | ~7 | 03 | ~5 | 05 | 02h6 |

Clock Cycles: 0.5

## CMOVEQ – Conditional Move Equal

Description:

The conditional move if equal instruction moves the contents of register Rb to the target register Rt if Ra is zero. Otherwise the contents of register Rc are moved to the target register.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 28h6 | Rt5 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

## CMOVNE – Conditional Move Not Equal

Description:

The conditional move if not equal instruction moves the contents of register Rb to the target register Rt if Ra is non-zero. Otherwise the contents of register Rc are moved to the target register.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 29h6 | Rt5 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

## CMP – Signed Comparison

Description:

The compare instruction places a 1, 0 or -1 in the target register based on the relationship between the two source operands. If they are equal a zero is placed in the target register, if register Ra is less than the second operand then a -1 is placed in the target register, otherwise a 1 is placed in the target register. The values are treated as signed operands. The immediate constant is sign extended to the width of the machine.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 06h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 06h6 | Cnd3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Parallel Operand (SIMD) compare

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 19h6 | Cnd3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

|  |  |
| --- | --- |
| Sz2 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |

|  |  |
| --- | --- |
| Cnd3 |  |
| 0 | CMP |
| 2 | SEQ |
| 3 | SNE |
| 4 | SLT |
| 5 | SGE |
| 6 | SLE |
| 7 | SGT |

## CMPU – Unsigned Comparison

Description:

The compare instruction places a 1, 0 or -1 in the target register based on the relationship between the two source operands. If they are equal a zero is placed in the target register, if register Ra is less than the second operand then a -1 is placed in the target register, otherwise a 1 is placed in the target register. The values are treated as unsigned operands. Note the immediate constant is sign extended but otherwise treated as an unsigned value.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 07h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 07h6 | Cnd3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Parallel Operand (SIMD) compare

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1Ah6 | Cnd3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

|  |  |
| --- | --- |
| Sz2 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |

|  |  |
| --- | --- |
| Cnd3 |  |
| 0 | CMPU |
| 4 | SLTU |
| 5 | SGEU |
| 6 | SLEU |
| 7 | SGTU |

## CNTLO – Count Leading Ones

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 016 | ~3 | Sz2 | 15 | Rt5 | Ra5 | 02h6 |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

|  |  |
| --- | --- |
| Sz2 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |

## CNTLZ – Count Leading Zeros

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 016 | ~3 | Sz2 | 05 | Rt5 | Ra5 | 02h6 |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

|  |  |
| --- | --- |
| Sz2 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |

## CNTPOP – Count Population

Description:

Count the number of ones and place the count in the target register.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 016 | ~3 | Sz2 | 25 | Rt5 | Ra5 | 02h6 |

Clock Cycles: 1

**Execution Units:** ALU #0 Only

Exceptions: none

|  |  |
| --- | --- |
| Sz2 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |

## 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 Rt then the CSR is updated from register Ra. The CSR read / update operation is an atomic operation.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Op2 | OL3 | Regno11 | Rt5 | Ra5 | 0Eh6 |

|  |  |  |
| --- | --- | --- |
| Op2 |  | Operation |
| 0 | CSRRD | Only read the CSR, no update takes place, Ra 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 |

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

The OL3 field is reserved to specify the operating level. Note that registers cannot be accessed by a lower operating level.

|  |  |  |  |
| --- | --- | --- | --- |
| Regno12 |  | 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 |
| 040 | 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 |
| 7F0 | INFO | R | Manufacturer name |
| 7F1 | “ | R | “ |
| 7F2 | “ | R | cpu class |
| 7F3 | “ | R | “ |
| 7F4 | “ | R | cpu name |
| 7F5 | “ | R | “ |
| 7F6 | “ | R | model number |
| 7F7 | “ | R | serial number |
| 7F8 | “ | R | cache sizes instruction (bits 32 to 63), data (bits 0 to 31) |

Clock Cycles: 0.5

## 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. Note that for the registered form of the instruction both the quotient and remainder may be calculated at the same time.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 3Eh6 |

Return quotient

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3E6 | 02 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Return remainder

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3E6 | 12 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 68 (n + 4) where n is the width

ALU Support: ALU #0 Only

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

## DIVSU – Signed-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 first operand is treated as a signed value. The second operand is an unsigned value. The result is a signed result.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 3Dh6 |

Return quotient

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Dh6 | 02 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Return remainder

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Dh6 | 12 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 68 (n + 4) where n is the width

ALU Support: ALU #0 Only

Exception: 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.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 3Ch6 |

Return quotient

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Ch6 | 02 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Return remainder

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Ch6 | 12 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 68 (n + 4) where n is the width

ALU Support: ALU #0 Only

Exceptions: none

## EXEC – Execute Code Buffer

Description:

Execute code from code buffer. The N6 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 has to change at runtime.

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ~10 | N6 | ~5 | ~5 | 0Fh6 |

Clock Cycles: Minimum 0.5 – depends on the instruction in the code buffer

## JAL – Jump-And-Link

Description:

Instruction Format:

This instruction loads the program counter with the sum of a register and a constant value specified in the instruction. In addition the address of the instruction following the JAL is stored in the specified target register. This instruction may be used to implement subroutine calls and returns. The two least significant bits of the program counter are forced to zero.

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 18h6 |

**Execution Units:** FCU

Clock Cycles:

## JMP – Jump to Address

Description:

A jump is made to the address specified in the instruction. The format first shifts the address field of the instruction by two bits to the left then modifies only PC bits 0 to 27. The high order PC bits are not affected. This allows accessing code within a 256MB region of memory. Note that with the use of a mmu this address range is often sufficient. If a larger address range is required the JAL instruction must be used.

Instruction Format:

|  |  |
| --- | --- |
| Address[27..2] | 28h6 |

Clock Cycles: 1

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

## LB – 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 Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 13h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 13h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LBO – Load Byte Only

Description:

This instruction loads a byte (8 bit) value from memory. Only the lower eight bits of the target register are updated, the upper bits of the register are not affected.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 2Ah6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2Ah6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LBU – 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 Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 23h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 23h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LC – Load Char (16 bits)

Description:

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

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 20h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 20h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LCO – Load Char Only (16 bits)

Description:

This instruction loads a char (16 bit) value from memory. Only the low order sixteen bits of the target register are updated, the remaining bits are not affected.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 2Bh6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2Bh6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LCU – Load Unsigned Char (16 bits)

Description:

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

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 21h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 21h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LDI – Load Immediate

Description:

This instruction loads an immediate value into a register. It is an alternate mnemonic for the OR instruction.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | 05 | 09h6 |

Clock Cycles: 0.5

## LEAX – Load Effective Address

Description:

This instruction loads an address value into a register.

Instruction Format:

This instruction format is simply an alternate mnemonic and representation for the ADD instruction. The ADD instruction is sufficient to calculate the effective address for register indirect with displacement addressing.

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 04h6 |

This instruction format is of the indexed load / store format, but places the calculated address in the target register rather than fetching or storing data.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 18h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LH – Load Half-Word (32 bits)

Description:

This instruction loads a half-word (32 bit) value from memory. The memory address must be half-word aligned. The value is sign extended to 64 bits when placed in the target register.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 10h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 10h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LHO – Load Half-Word Only (32 bits)

Description:

This instruction loads a half-word (32 bit) value from memory. The memory address must be half-word aligned. Only the lower 32 bits of the register are updated, the remaining bits are unchanged.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 35h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 35h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LHU – Load Half-Word (32 bits)

Description:

This instruction loads a half-word (32 bit) value from memory. The memory address must be half-word aligned. The value is zero extended to 64 bits when placed in the target register.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 11h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 11h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LW – Load Word (64 bits)

Description:

This instruction loads a word (64 bit) value from memory. The memory address must be word aligned.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 12h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 12h6 | ~3 | Sc2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## LWR – Load Word and Reserve Address

Description:

This instruction loads a word (64 bit) value from memory and places a reservation on the address. The memory address must be word aligned. This instruction activates the sr\_o signal output by the core. It relies on external hardware to implement the address reservation. This instruction performs an un-cached load operation.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 1Dh6 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1Dh6 | A | R | Sc3 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

Acquire and release bits determine the ordering of memory operations.

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

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

All combinations of A, R are allowed.

## MAJ – Majority Logic

**Description:**

Determines the majority logic bits of three values in registers Ra, Rb, and Rc and places the result in the target register Rt.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2Eh6 | Rt5 | Rc5 | Rb5 | Ra5 | 02h6 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

Rt = (Ra & Rb) | (Ra & Rc) | (Rb & Rc)

## MAX – Maximum Value

**Description:**

Determines the maximum of two values in registers Ra, Rb and places the result in the target register Rt.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2Dh6 | ~2 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

IF Ra > Rb

Rt = Ra

else

Rt = Rb

## 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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 01h6 | ~5 | 10h5 | ~5 | ~5 | 02h6 |

Clock Cycles: varies depending on queue contents

## 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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 01h6 | ~5 | 11h5 | ~5 | ~5 | 02h6 |

Clock Cycles: varies depending on queue contents

## MIN – Minimum Value

**Description:**

Determines the minimum of two values in registers Ra, Rb and places the result in the target register Rt.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2Ch6 | ~2 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

**Clock Cycles:** 1

**Execution Units:** ALU #0 only

**Operation:**

IF Ra < Rb

Rt = Ra

else

Rt = Rb

## 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:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 2Eh6 |

Return remainder

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3E6 | 12 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 68 (n + 4) where n is the width

ALU Support: ALU #0 Only

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

## MODSU – Signed-Unsigned 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 first operand is treated as a signed value. The second operand is an unsigned value. The result is a signed result.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 2Dh6 |

Return remainder

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Dh6 | 12 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 68 (n + 4) where n is the width

ALU Support: ALU #0 Only

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

## MODU – Unsigned 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 unsigned values and the result is an unsigned result.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 2Ch6 |

Return remainder

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Ch6 | 12 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 68 (n + 4) where n is the width

ALU Support: ALU #0 Only

Exceptions: none

## MOV – Move register to register

Description:

This instruction is an alternate mnemonic for the or instruction.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| 016 | Rt5 | Ra5 | 09h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 22h6 | D3 | ~1 | Rgs6 | Rt5 | Ra5 | 02h6 |

|  |  |
| --- | --- |
| D2 | Operation |
| 0 | move from current Ra to Rt in register set Rgs |
| 1 | move from Ra in register set Rgs to Rt in current register set |
| 2 | move from current Ra to Rt in excepted register set (Rgs is ignored). |
| 3 | move from Ra in excepted register to Rt in current register set (Rgs is ignored). |
| 4 |  |
| 5 |  |
| 6 |  |
| 7 | move from current Ra to current Rt (rgs ignored). |

Clock Cycles: 0.5

## MUL – 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. For the registered form of the instruction both the high order and low order halves of the result are available. For the immediate form of the instruction, only the low order half (bits 0 to 63) of the product is available.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 3Ah6 |

Multiply, return low order product

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Ah6 | 03 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Multiply, return high order product

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Ah6 | 13 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 19

## MULSU – Signed-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. The first operand is treated as a signed value. The second operand is treated as an unsigned value. The result is a signed result.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 39h6 |

Multiply, return low order product

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39h6 | 03 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Multiply, return high order product

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39h6 | 13 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 19

## 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.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 38h6 |

Multiply, return low order product

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 38h6 | 03 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Multiply, return high order product

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 38h6 | 13 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 19

Exceptions: none

## MUX – Multiplex

Description:

The MUX instruction performs a bit-by-bit copy of a bit of Rb to the target register if the corresponding bit in Ra is set, or a copy of a bit from Rc if the corresponding bit in Ra is clear.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1Bh6 | Rt5 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

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:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0C6 | ~3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

Exceptions: none

## NEG - Negate

Description:

This is an alternate mnemonic for the SUB instruction where the first register operand is R0.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 056 | ~2 | Sz3 | Rt5 | Rb5 | 05 | 02h6 |

Clock Cycles: 0.5

## 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:

|  |  |
| --- | --- |
| Immediate26 | 1Ch6 |

## NOR – Bitwise Nor

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0D6 | ~3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

Exceptions: none

## OR – Bitwise Or

Description:

Perform a bitwise or operation between operands.

Instruction Format:

The immediate value is sign extended to the left before use.

Rt = Ra | immed

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 09h6 |

Rt = Ra | Rb

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 096 | ~2 | Sz3 | Rt5 | Rb5 | Ra5 | 02h6 |

Instruction Format:

This format performs the ‘or’ operation with an immediate value to one of four quadrants of the target register. It may be used to build a 64 bit constant in a register. The immediate constant is zero extended then shifted to the left by 0, 16, 32, or 48 bits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Immed16 | Rt5 | 03 | Q2 | 1Ah6 |

|  |  |
| --- | --- |
| Q2 | Bits |
| 0 | 0 to 15 |
| 1 | 16 to 31 |
| 2 | 32 to 47 |
| 3 | 48 to 63 |

Clock Cycles: 0.5

**Execution Units: All** ALUs

Exceptions: none

## RET – Return from Subroutine

Description:

This instruction performs a subroutine return by loading the program counter with the contents of the link register. Additionally, the stack pointer is adjusted by a constant supplied in the instruction. The immediate constant should be a multiple of eight to keep the stack word aligned.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | 1Dh5 | 1Fh5 | 29h6 |

PC = LR

SP = SP + Immediate

Exceptions:

## 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 Ra. 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:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 24 | 23 16 | 1514 | 13 11 | 10 6 | 5 0 |
| ~5 | IM3 | PL8 | ~2 | Tgt3 | Ra5 | 0Dh6 |

|  |  |
| --- | --- |
| Tgt3 |  |
| 0 | not used |
| 1 | redirect to hypervisor level |
| 2 | redirect to supervisor level |
| 3 | redirect to supervisor level |
| 4 | redirect to supervisor level |
| 5 | redirect to supervisor level |
| 6 | redirect to supervisor level |
| 7 | not used |

Clock Cycles: 3

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.

## 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:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Func6 | 44 | E | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Func6 | Ch4 | Imm6 | Rt5 | Ra5 | 02h6 |

|  |  |  |
| --- | --- | --- |
| Func6 | Op Size | If E set |
| 0Fh | word | word |
| 1Fh | byte | byte parallel |
| 2Fh | char | char parallel |
| 3Fh | half | half parallel |

Clock Cycles: 1

ALU Support: ALU #0 Only

Exceptions: none

## ROR – Rotate Right

Description:

Bits from the source register Ra are shifted right by the amount in register Rb 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:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Func6 | 54 | E | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Func6 | Dh4 | Imm6 | Rt5 | Ra5 | 02h6 |

|  |  |  |
| --- | --- | --- |
| Func6 | Op Size | If E set |
| 0Fh | word | word |
| 1Fh | byte | byte parallel |
| 2Fh | char | char parallel |
| 3Fh | half | half parallel |

Clock Cycles: 1

ALU Support: ALU #0 Only

Exceptions: none

## RTI – Return from Interrupt

Description:

Return from an interrupt subroutine. The interrupted program counter is loaded into the program counter 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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 32h6 | ~4 | Sema6 | ~5 | Ra5 | 02h6 |

Semaphore[Sema6|[Ra]] = 0

**Clock Cycles:** 8 minimum

**Execution Units:** Flow Control Unit

## RTE – Return from Exception

Description:

This is an alternate mnemonic for the RTI instruction.

Instruction Format:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 32h6 | ~4 | Sema6 | ~5 | Ra5 | 02h6 |

Semaphore[Sema6|[Ra]] = 0

Clock Cycles:

## SB – Store Byte

Description:

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

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rb5 | Ra5 | 15h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 15h6 | ~3 | Sc2 | Rc5 | Rb5 | Ra5 | 02h6 |

Operation:

Memory8[Ra + immediate] = Rb

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## SC – Store Char (16 bits)

Description:

This instruction stores a char (16 bit) value to memory. The memory address must be char (16 bit) aligned.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rb5 | Ra5 | 24h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 24h6 | ~3 | Sc2 | Rc5 | Rb5 | Ra5 | 02h6 |

Operation:

Memory16[Ra + immediate] = Rb

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## Scc – Set

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands.

Instruction Format:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cond4 | Immed12 | Rt5 | Ra5 | 1Bh6 |

Signed

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 066 | Cnd3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Unsigned

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 076 | Cnd3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

|  |  |
| --- | --- |
| Cond3 / Cond4 |  |
| 0 | CMP / CMPU |
| 2 | SEQ |
| 3 | SNE |
| 4 /12 | SLT / SLTU |
| 5 /13 | SGE / SGEU |
| 6 /14 | SLE / SLEU |
| 7 /15 | SGT / SGTU |

|  |  |
| --- | --- |
| Sz2 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |

## SEI – Set Interrupt Mask

SEI #3

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 Ra and an immediate (M3) supplied in the instruction. The assembler assumes a mask value of seven, masking all interrupts, if no mask value is specified. Usually either M3 or Ra should be zero.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 306 | ~4 | ~3 | M3 | ~5 | Ra5 | 02h6 |

Operation:

im = M3 | Ra

## SGN – Get Sign

Description:

The SGN instruction places a 1, 0 or -1 in the target register depending on the sign of the source operand. This instruction is an alternate mnemonic for the compare instruction where the value is compared to zero.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 066 | ~3 | Sz2 | Rt5 | 05 | Ra5 | 02h6 |

Clock Cycles: 0.5

|  |  |
| --- | --- |
| Sz2 |  |
| 0 | Byte |
| 1 | Char |
| 2 | Half |
| 3 | Word |

## SH – Store Half-Word (32 bits)

Description:

This instruction stores a half-word (32 bit) value to memory. The memory address must be half-word aligned.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rb5 | Ra5 | 14h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 14h6 | ~3 | Sc2 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## SHL – Shift Left

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Func6 | 04 | E | Rt5 | Rb5 | Ra5 | 02h6 |

E indicates to update all lanes of target register.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Func6 | 84 | Imm6 | Rt5 | Ra5 | 02h6 |

|  |  |  |
| --- | --- | --- |
| Func6 | Op Size | If E set |
| 0Fh | word | word |
| 1Fh | byte | byte parallel |
| 2Fh | char | char parallel |
| 3Fh | half | half parallel |

Clock Cycles: 1

ALU Support: ALU #0 Only

Exceptions: none

## SHR – Shift Right

Description:

Bits from the source register Ra are shifted right by the amount in register Rb or an immediate value. Zeros are shifted into the most significant bits.

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Func6 | 14 | E | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Func6 | 94 | Imm6 | Rt5 | Ra5 | 02h6 |

|  |  |  |
| --- | --- | --- |
| Func6 | Op Size | If E set |
| 0Fh | word | word |
| 1Fh | byte | byte parallel |
| 2Fh | char | char parallel |
| 3Fh | half | half parallel |

Clock Cycles: 1

ALU Support: ALU #0 Only

Exceptions: none

## SUB - Subtract

Description:

Subtract two values. Both operands must be in a register.

Instruction Format:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 056 | ~2 | Ov | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

|  |  |
| --- | --- |
| Ov |  |
| 0 | no overflow |
| 1 | overflow exception if overflow occurred and enabled in AEC |

Overflow works properly only on 64 bit values.

Clock Cycles: 0.5

Exceptions:

The registered form of the instruction may cause an overflow exception if enabled in the AEC register.

## SW – Store Word (64 bits)

Description:

This instruction stores a word (64 bit) value to memory. The memory address must be word aligned.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rb5 | Ra5 | 16h6 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 16h6 | ~3 | Sc2 | Rc5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

## SWC – Store Word and Clear Reservation

Description:

This instruction conditionally stores a word (64 bit) value to memory and clears any memory reservation that was previously set at the address. If the memory address was reserved at the time of the store the store will succeed, otherwise the data is not stored. The previous status of the reservation is copied to the least significant bit of the semaphore register. This instruction depends on external hardware to implement the reservation. The instruction activates the cr\_o signal output by the core. The memory address must be word aligned. This instruction should be both preceded and succeeded by SYNC instructions to ensure that the reservation status bit is updated correctly in the semaphore CSR.

Instruction Format:

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rb5 | Ra5 | 17h6 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 17h6 | A | R | Sc3 | Rc5 | Rb5 | Ra5 | 02h6 |

Side Effect: the reservation status bit (bit 0) in the semaphore register is set accordingly.

Clock Cycles: 4 minimum depending on memory access time

|  |  |
| --- | --- |
| Sc2 | Scale Rb By |
| 0 | 1 |
| 1 | 2 |
| 2 | 4 |
| 3 | 8 |

Acquire and release bits determine the ordering of memory operations.

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

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

All combinations of A, R are allowed.

## 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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 01h6 | 05 | 12h5 | ~5 | ~5 | 02h6 |

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

**Execution Units:** All ALU’s

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.

## WAIT – Wait For Signal

**Description:**

This instruction causes the core to pause execution during the execute phase of the instruction until an external signal is true. Note that instructions already in the queue before the wait will continue to execute to completion. Also additional instructions may be fetched after the wait instruction however they will not be able to update the state of the machine until the wait is done.

The signal to wait for is specified as the union of register Ra and an immediate value. Either Ra or the immediate value should be zero.

A timeout for the wait may be specified in register Rb. If a timeout is not desired use R0 for Rb and the instruction will wait indefinitely.

**Instruction Formats**:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 316 | ~5 | Imm5 | Rb5 | Ra5 | 02h6 |

**Operation**:

if (no signal)

delay instruction

else

mark instruction done

Notes:

This instruction waits for a signal to occur before proceeding.

## XNOR – Bitwise Exclusive Nor

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0E6 | ~3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Clock Cycles: 0.5

Exceptions: none

## XOR – Bitwise Exclusive Or

Description:

Perform a bitwise exclusive or operation between operands.

Instruction Format:

The immediate constant is sign extended to the left before use.

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Rt5 | Ra5 | 0Ah6 |

Rt = Ra ^ Rb ^ Rc

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0A6 | ~3 | Sz2 | Rt5 | Rb5 | Ra5 | 02h6 |

Instruction Format:

This format performs the ‘xor’ operation with an immediate value to one of four quadrants of the target register. The immediate constant is zero extended then shifted to the left by 0, 16, 32, or 48 bits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Immed16 | Rt5 | 33 | Q2 | 1Ah6 |

|  |  |
| --- | --- |
| Q2 | Bits |
| 0 | 0 to 15 |
| 1 | 16 to 31 |
| 2 | 32 to 47 |
| 3 | 48 to 63 |

Clock Cycles: 0.5

# Floating Point Instruction Set

## Overview

The floating point unit provides basic floating point operations including addition, subtraction, multiplication, division and float to integer and integer to float conversions. The core contains only a single floating point unit. Only double precision floating point operations are supported.

The precision field (prec2) should be set to 1.

The rounding mode is normally specified by the rounding mode bits in the floating point control and status register. However it may be overridden by specification of a rounding mode in the instruction.

**Representation**

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

**Double Precision Format:**

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

SM – sign of mantissa

SE – sign of exponent

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

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

The exponent ranges from -1024 to +1023 for double precision numbers

## Instruction Format

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| Func6 | Prec2 | Rm3 | Rt5 | Rb5 | Ra5 | 0Bh6 |

Not all instructions required the Rb5 field. If not required Rb should be set to zero.

## FABS – Floating Absolute Value

**Description:**

Take the absolute value of the floating point number in registers Ra and place the result into target register Rt. The sign bit (bit 63) of the register is set to zero. No rounding of the number occurs.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 15h6 | Prec2 | Rm3 | Rt5 | 05 | Ra5 | 0Bh6 |

**Clock Cycles: 2**

**Execution Units:** Floating Point

## FADD – Floating point addition

**Description:**

Add two floating point numbers in registers Ra and Rb and place the result into target register Rt.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 046 | Prec2 | Rm3 | Rt5 | Rb5 | Ra5 | 0Bh6 |

**Clock Cycles: 10**

**Execution Units:** Floating Point

## FBEQ –Branch if Equal

Description:

If two registers are equal an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. The displacement value may not be extended with a prefix instruction. Note that positive and negative zero are treated as equal.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 84 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | 84 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra = Rb)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## FBGE –Branch if Greater than or Equal

Description:

If register Ra is greater than or equal to register Rb an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. The displacement value may not be extended with a prefix instruction. A branch on less than or equal may be achieved by swapping registers.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | Bh4 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | Bh4 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra >= Rb)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## FBLT –Branch if Less Than

Description:

If register Ra is less than register Rb an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. The displacement value may not be extended with a prefix instruction. A branch on greater than may be achieved by swapping the registers.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | Ah4 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | Ah4 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra < Rb)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## FBNE –Branch if Not Equal

Description:

If two registers are unequal an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. The displacement value may not be extended with a prefix instruction. Note that positive and negative zero are treated as equal.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | 94 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | 94 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra <> Rb)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## FBUN –Branch if Unordered

Description:

If the comparison is unordered an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. The displacement value may not be extended with a prefix instruction.

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 22 | 21 | 19 16 | 15 11 | 10 6 | 5 1 | 0 |
| Displacement10 | P2 | Fh4 | Rb5 | Ra5 | 18h5 | D1 |

A branch to a value computed in a register may be performed using the instruction format shown below. Rc contains the target address which is an absolute address.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 27 | 26 | 24 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| ~5 | P2 | Fh4 | Rc5 | Rb5 | Ra5 | 03h6 |

Operation:

if (Ra ? Rb)

pc = pc + displacement

The P2 field is reserved for branch prediction hints.

|  |  |
| --- | --- |
| P2 | Prediction Type |
| 0 | no static prediction (use branch history) |
| 1 | reserved |
| 2 | always predict as not-taken |
| 3 | always predict as taken |

If a branch prediction is supplied, then the branch instruction doesn’t occupy room in the history tables.

## FCMP - Float Compare

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 066 | Prec2 | Rm3 | Rt5 | Rb5 | Ra5 | 0Bh6 |

**Clock Cycles:** 2

**Execution Units:** FPU

**Operation:**

if Ra < Rb

Rt[1]= true

else

Rt[1] = false

if mag Ra < mag Rb

Rt[2] = true

else

Rt[2] = false

if Ra = Rb

Rt[0] = true

else

Rt[0] = false

if Ra <= Rb

Rt[3] = true

else

Rt[3] = false

if unordered

Rt[4] = true

else

Rt[4] = false

## FCVTSD – Convert Single to Double

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 19h6 | Prec2 | Rm3 | Rt5 | 05 | Ra5 | 0Bh6 |

**Clock Cycles: 3**

**Execution Units:** Floating Point

## FDIV – Floating point divide

**Description:**

Divide two floating point numbers in registers Ra and Rb and place the result into target register Rt.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 086 | Prec2 | Rm3 | Rt5 | Rb5 | Ra5 | 0Bh6 |

**Clock Cycles: 115**

**Execution Units:** Floating Point

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

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 2322 | 21 16 | 15 11 | 10 6 | 5 0 |
| 21h6 | Prec2 | ~2 | Imm6 | 05 | Ra5 | 0Bh6 |

**Execution Units:** All Floating Point

**Operation:**

**Exceptions:**

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

## FDX – Floating Disable Exceptions

**Description:**

This instruction disables floating point exceptions. The Exceptions disabled are identified as the bits set in the union of register Ra and an immediate field in the instruction. Either the immediate or Ra 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:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 2322 | 21 16 | 15 11 | 10 6 | 5 0 |
| 23h6 | Prec2 | ~2 | Imm6 | 05 | Ra5 | 0Bh6 |

**Clock Cycles: 2**

**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 Ra and an immediate field in the instruction. Either the immediate or Ra 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:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 2322 | 21 16 | 15 11 | 10 6 | 5 0 |
| 22h6 | Prec2 | ~2 | Imm6 | 05 | Ra5 | 0Bh6 |

**Clock Cycles: 2**

**Execution Units:** Floating Point

## FMUL – Floating point multiplication

**Description:**

Multiply two floating point numbers in registers Ra and Rb and place the result into target register Rt.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 086 | Prec2 | Rm3 | Rt5 | Rb5 | Ra5 | 0Bh6 |

**Clock Cycles: 12**

**Execution Units:** Floating Point

## FNABS – Floating Negative Absolute Value

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 18h6 | Prec2 | Rm3 | Rt5 | 05 | Ra5 | 0Bh6 |

**Clock Cycles: 2**

**Execution Units:** Floating Point

## FNEG – Floating Negative Value

**Description:**

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

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 14h6 | Prec2 | Rm3 | Rt5 | 05 | Ra5 | 0Bh6 |

**Clock Cycles: 2**

**Execution Units:** Floating Point

## 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:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 16h6 | Prec2 | Rm3 | Rt5 | 05 | Ra5 | 0Bh6 |

**Clock Cycles: 2**

**Execution Units:** Floating Point

## FSUB – Floating point subtraction

**Description:**

Subtract two floating point numbers in registers Ra and Rb and place the result into target register Rt.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 056 | Prec2 | Rm3 | Rt5 | Rb5 | Ra5 | 0Bh6 |

**Clock Cycles: 10**

**Execution Units:** Floating Point

## FSYNC -Synchronize

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 36h6 | ~2 | ~3 | ~5 | ~5 | ~5 | 0Bh6 |

Clock Cycles: varies depending on queue contents

## FTOI – Floating Convert to Integer

**Description:**

Convert the floating point value in Ra into an integer and place the result into target register Rt. If the result overflows the value placed in Rt is a maximum integer value.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 12h6 | Prec2 | Rm3 | Rt5 | 05 | Ra5 | 0Bh6 |

**Clock Cycles: 3**

**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 Ra and an immediate field in the instruction. Either the immediate or Ra should be zero.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 2322 | 21 16 | 15 11 | 10 6 | 5 0 |
| 20h6 | Prec2 | ~2 | Imm6 | 05 | Ra5 | 0Bh6 |

**Execution Units:** All Floating Point

**Operation:**

**Exceptions:**

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

## ITOF – Convert Integer to Float

**Description:**

Convert the integer value in Ra into a floating point value and place the result into target register Rt. Some precision of the integer converted may be lost if the integer is larger than 52 bits. Double precision floating point values only have a precision of 53 bits.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 15h6 | Prec2 | Rm3 | Rt5 | 05 | Ra5 | 0Bh6 |

**Clock Cycles: 3**

**Execution Units:** Floating Point

# Vector Programming Model

The ISA supports up to 31 vector registers of length 64.

|  |  |
| --- | --- |
| Reg no |  |
| 0 | <vector mask registers> |
| 1 to 31 | general purpose vector registers |

## Vector Length (VL register)

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

|  |  |
| --- | --- |
| 7 | 6 0 |
| 0 | Length6..0 |

## Vector Masking

All vector operations are performed conditionally depending on the setting in the vector mask register unless otherwise noted.

## Vector Mask (Vm registers)

The ISA supports up to eight, sixty-four element vector mask registers. In the proof-of-concept version there is are four sixteen 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 VSEQ, VSNE, VSLT, VSGE, VSLE, VSGT. Mask registers may also be manipulated using one of the mask register operations VMAND, VMOR, VMXOR, VMXNOR, VMFILL.

After a change to a mask register a SYNC instruction should be used to ensure that the updated mask register is visible to following instructions.

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

The vector mask registers are aliased with vector register #0. The mask registers may be manipulated as a group by referencing v0.

## Detailed Vector Instruction Set

### LV – Load Vector

Synopsis

Load vector

**Description:**

Load a vector register from memory. Vector mask register #0 is used to mask the operation.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Vt5 | Ra5 | 36h6 |

**Operation**

for x = 0 to VL-1

if vm[x]

Vt[x] = memory64[Ra + Immed + 8 \* x]

else

NOP

**Exceptions:** DBE, DBG, LMT

### LVWS – Load Vector With Stride

Synopsis

Load vector

**Description:**

Load a vector register from memory using indexed addressing.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 26h6 | Vm3 | 32 | Vt5 | Rb5 | Ra5 | 02h6 |

**Operation**

for x = 0 to VL-1

Vt[x] = memory64[Ra+Rb\*x\*8]

**Exceptions:** DBE, DBG, LMT

### LVX – Load Vector

Synopsis

Load vector

**Description:**

Load a vector register from memory using vector indexed addressing.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 36h6 | ~3 | 32 | Vt5 | Vb5 | Ra5 | 02h6 |

**Operation**

for x = 0 to VL-1

Vt[x] = memory64[Ra+Vb[x]]

**Exceptions:** DBE, DBG, LMT

### SV – Store Vector

Synopsis

Load vector

**Description:**

Store a vector register to memory. Vector mask register #0 is used to mask the operation.

**Instruction Format:**

|  |  |  |  |
| --- | --- | --- | --- |
| Immed16 | Vb5 | Ra5 | 37h6 |

**Operation**

for x = 0 to VL-1

if (vm[x])

memory64[Ra + Immed + 8 \* x] = Vb[x]

else

NOP

**Exceptions:** DBE, DBG, LMT

### SVWS – Store Vector With Stride

Synopsis

Store vector

**Description:**

Store a vector register to memory using indexed addressing.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 27h6 | Vm3 | 32 | Vc5 | Rb5 | Ra5 | 02h6 |

**Operation**

for x = 0 to VL-1

memory64[Ra+Rb\*(x\*8)] = Vc[x]

**Exceptions:** DBE, DBG, LMT

### SVX – Store Vector

Synopsis

Load vector

**Description:**

Store a vector register to memory using vector indexed addressing.

**Instruction Format:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 37h6 | ~3 | 32 | Vc5 | Vb5 | Ra5 | 02h6 |

**Operation**

for x = 0 to VL-1

memory64[Ra+Vb[x]] = Vc[x]

**Exceptions:** DBE, DBG, LMT

### V2BITS

Synopsis

Convert Boolean vector to bits.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 21h6 | Vm3 | 02 | 05 | Rt5 | Va5 | 01h6 |

**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

### VABS – Absolute value

Synopsis

Vector register absolute value. Vt = Va < 0 ? –Va : Va

**Description**

The absolute value of a vector register is placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 036 | Vm3 | T2 | Vt5 | 05 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] < 0 ? –Va[x] : Va[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VADD - Add

Synopsis

Vector register add. Vt = Va + Vb

**Description**

Two vector registers (Va and Vb) are added together and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 046 | Vm3 | T2 | Vt5 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] + Vb[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VADDS – Add Scalar

Synopsis

Vector register add. Vt = Va + Rb

**Description**

A vector and a scalar (Va and Rb) are added together and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 14h6 | Vm3 | T2 | Vt5 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Vb[x] + Rb

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VAND – Bitwise And

Synopsis

Vector register bitwise and. Vt = Va & Vb

***Description***

Two vector registers (Va and Vb) are bitwise and’ed together and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 086 | Vm3 | 02 | Vt5 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] & Vb[x]

**Execution Units:** ALUs

### VANDS – Bitwise And with Scalar

Synopsis

Vector register bitwise and. Vt = Va & Rb

***Description***

A vector register (Va) is bitwise and’ed with a scalar register and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 18h6 | Vm3 | 02 | Vt5 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] & Rb[x]

### VASR – Arithmetic Shift Right

Synopsis

Vector signed shift right.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0Eh6 | S | M2 | S | A | Vt5 | Amt5 | Va5 | 01h6 |

**Description**

Elements of the vector are shifted right. The most significant bits are loaded with the sign bit.

**Operation**

For x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] >> amt

**Exceptions:** none

|  |  |  |
| --- | --- | --- |
| S2 | Amount Field |  |
| 0 | general purpose register |  |
| 1 | vector register |  |
| 2 | immediate |  |
| 3 | reserved |  |

### VBITS2V

Synopsis

Convert bits to Boolean vector.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 20h6 | Vm3 | 02 | Vt5 | 05 | Ra5 | 01h6 |

**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]

**Exceptions:** none

**Execution Units:** ALUs

### VCIDX – Compress Index

Synopsis

Vector compression.

**Description**

A value in a register Ra is multiplied by the element number and copied to elements of vector register Vt guided by a vector mask register.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 016 | Vm3 | 02 | Vt5 | 05 | Ra5 | 01h6 |

**Operation**

y = 0

for x = 0 to VL - 1

if (Vm[x])

Vt[y] = Ra \* x

y = y + 1

### VCMPRSS – Compress Vector

Synopsis

Vector compression.

**Description**

Selected elements from vector register Va are copied to elements of vector register Vt guided by a vector mask register.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 006 | Vm3 | 02 | Vt5 | 05 | Va5 | 01h6 |

**Operation**

y = 0

for x = 0 to VL - 1

if (Vm[x])

Vt[y] = Va[x]

y = y + 1

### VCNTPOP – Population Count

Synopsis

Vector register population count. Vt = popcnt(Va)

**Description**

The number of bits set in a vector register is placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 28h6 | Vm3 | 02 | Vt5 | 05 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = popcnt(Va[x])

### VDIV - Divide

Synopsis

Vector register divide. Vt = Va / Vb

**Description**

Vector register Va is divided by Vb and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Eh6 | Vm3 | T2 | Vt5 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] / Vb[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VDIVS – Divide by Scalar

Synopsis

Vector register divide by scalar. Vt = Va / Rb

**Description**

Vector register Va is divided by Rb and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2Eh6 | Vm3 | T2 | Vt5 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] / Rb[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VEINS / VMOVSV – Vector Element Insert

Synopsis

Vector element insert.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 22h6 | ~ | M2 | 02 | Vt5 | Rb5 | Ra5 | 01h6 |

**Description**

A general purpose register Rb is transferred into one element of a vector register Vt. The element to insert is identified by Ra.

**Operation**

Vt[Ra] = Rb

Exceptions: none

### VEX / VMOVS – Vector Element Extract

Synopsis

Vector element insert.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 23h6 | ~ | M2 | 02 | Rt5 | Vb5 | Ra5 | 01h6 |

**Description**

A vector register element from Vb is transferred into a general purpose register Rt. The element to extract is identified by Ra.

**Operation**

Rt = Vb[Ra]

Exceptions: none

### VFLT2INT – Float to Integer

Synopsis

Vector float to integer.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 24h6 | ~ | M2 | T2 | Vt5 | 05 | Va5 | 01h6 |

**Description**

Elements of the vector are converted from floating point to integer.

**Operation**

For x = 0 to [Ra]-1

Vt[x] = (int)Va[x]

**Exceptions:** none

### VINT2FLT – Integer to Float

Synopsis

Vector float to integer.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 25h6 | ~ | M2 | T2 | Vt5 | 05 | Va5 | 01h6 |

**Description**

Elements of the vector are converted from integer to floating point.

**Operation**

For x = 0 to VL-1

Vt[x] =(float) Va[x]

**Exceptions:** none

### VMAND – 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**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 30h6 | 03 | 04 | Vmt3 | 02 | Vmb3 | 02 | Vma3 | 01h6 |

**Operation**

Vmt = Vma & Vmb

**Execution Units:** ALUs

### VMFILL –Mask Fill

Synopsis

Fill vector mask register with bits.

***Description***

The first Ra bits of the vector mask register are set to one. The remaining bits of the mask register are set to zero.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 30h6 | 53 | 02 | Vmt5 | 05 | Ra5 | 01h6 |

**Operation**

for x = 0 to VLMAX

if (x < Ra) Vmt[x] = 1

else Vmt[x] = 0

**Execution Units:** ALUs

### VMFIRST – Find First Set Bit

Synopsis

Convert Boolean vector to bits.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 30h6 | 63 | 02 | 05 | Rt5 | ~2 | Vm3 | 01h6 |

**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 64. The search begins at the least significant bit and proceeds to the most significant bit.

**Operation**

Rt = first set bit number of (Vm)

**Exceptions:** none

**Execution Units:** ALUs

### VMLAST – Find Last Set Bit

Synopsis

Convert Boolean vector to bits.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 30h6 | 73 | 02 | 05 | Rt5 | ~2 | Vm3 | 01h6 |

**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 64. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

**Operation**

Rt = first set bit number of (Vm)

**Exceptions:** none

**Execution Units:** ALUs

### VMOR – Bitwise Mask Or

Synopsis

Vector mask register bitwise and. Vmt = Vma | Vmb

***Description***

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

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 30h6 | 13 | 04 | Vmt3 | 02 | Vmb3 | 02 | Vma3 | 01h6 |

**Operation**

Vmt = Vma | Vmb

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | reserved |  |
| 2 | reserved |  |
| 3 | reserved |  |

**Execution Units:** ALUs

### VMOV – Move Vector Control Register

**Description:**

.

**Instruction Format:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 33h6 | 05 |  | Vt5 | Ra5 | 02h6 |

|  |  |  |
| --- | --- | --- |
| Va5/Vt5 |  |  |
| 0 to 7 | Vector Mask |  |
| 15 | Vector Length |  |
|  |  |  |
|  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 33h6 | 15 |  | Rt5 | Va5 | 02h6 |

**Clock Cycles:** 1

**Execution Units:** ALUs

### VMPOP – Mask Population Count

Synopsis

Convert Boolean vector to bits.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 30h6 | 43 | 02 | 05 | Rt5 | ~2 | Vm3 | 01h6 |

**Description**

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

**Operation**

Rt = population count(Vm)

**Exceptions:** none

**Execution Units:** ALUs

### VMUL - Multiply

Synopsis

Vector register multiply. Vt = Va \* Vb

**Description**

Two vector registers (Va and Vb) are multiplied together and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3Ah6 | Vm3 | T2 | Vt5 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] \* Vb[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VMULS – Multiply by Scalar

Synopsis

Vector register multiply by scalar. Vt = Va \* Rb

**Description**

A vector registers (Va) and a scalar register (Rb) are multiplied together and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2Ah6 | Vm3 | T2 | Vt5 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] \* Rb

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VMXNOR – Bitwise Mask Exclusive Nor

Synopsis

Vector mask register bitwise and. Vmt = ~(Vma ^ Vmb)

***Description***

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

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 30h6 | 33 | 04 | Vmt3 | 02 | Vmb3 | 02 | Vma3 | 01h6 |

**Operation**

Vmt = Vma ^ Vmb

**Execution Units:** ALUs

### VMXOR – Bitwise Mask Exclusive Or

Synopsis

Vector mask register bitwise and. Vmt = Vma ^ Vmb

***Description***

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

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 30h6 | 23 | 04 | Vmt3 | 02 | Vmb3 | 02 | Vma3 | 01h6 |

**Operation**

Vmt = Vma ^ Vmb

**Execution Units:** ALUs

### VNEG – Negate

Synopsis

Vector register subtract. Vt = R0 - Va

**Description**

A vector is made negative by subtracting it from zero and placing it in the target vector register Vt. This instruction is an alternate mnemonic for the VSUBRS instruction.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 16h6 | Vm3 | T2 | Vt5 | 05 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = R0 - Va[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VOR – Bitwise Or

Synopsis

Vector register bitwise or. Vt = Va | Vb

***Description***

Two vector registers (Va and Vb) are or’ed together and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 096 | Vm3 | T2 | Vt5 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] | Vb[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VORS – Bitwise Or with Scalar

Synopsis

Vector register bitwise and. Vt = Va | Rb

***Description***

A vector register (Va) is bitwise ord’ed with a scalar register and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 19h6 | ~ | M2 | T2 | Vt5 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] | Rb[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | reserved |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSxx / VSxxS

Synopsis

Vector register set. Vm = Va ? Vb

**Description**

A vector register is compared to either a second vector register or a scalar register and the comparison result is placed in the target vector mask register Vmt.

**Instruction Format**

Vector-Vector Compare (VSxx)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 066/3F6 | M3 | T2 | Cn2 | Vmt3 | Vb5 | Va5 | 01h6 |

Vector-Vector Unsigned Compare (VSxxU)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 27h6/2F6 | M3 | T2 | Cn2 | Vmt3 | Vb5 | Va5 | 01h6 |

Vector-Scalar Compare (VSxxS)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 076/0F6 | M3 | T2 | Cn2 | Vmt3 | Rb5 | Va5 | 01h6 |

Vector-Scalar Unsigned Compare (VSxxSU)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 17h6/1F6 | M3 | T2 | Cn2 | Vmt3 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

Vt[x] = Va[x] ? Vb[x]

**Operation:**

**For each vector element**

if signed Va op signed Vb

Vm = true

else

Vm = false

Set Condition

|  |  |  |
| --- | --- | --- |
| Cn3 |  |  |
| 0 | Equal |  |
| 1 | Not Equal |  |
| 2 | Less Than |  |
| 3 | Greater Than or Equal |  |
| 4 | Less Than or Equal |  |
| 5 | Greater Than |  |
| 6 | reserved |  |
| 7 | unordered |  |

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSCAN

Synopsis

.

**Description**

Elements of Vt are set to the cumulative sum of a value in register Ra. The summation is guided by a vector mask register.

**Instruction Format**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 026 | ~ | M2 | 02 | Vt5 | 05 | Ra5 | 01h6 |

**Operation**

sum = 0

for x = 0 to VL - 1

Vt[x] = sum

if (Vm[x])

sum = sum + Ra

### VSEQ – Set if Equal

Synopsis

Vector register set. Vm = Va == Vb

**Description**

Two vector registers (Va and Vb) are compared for equality and the comparison result is placed in the target vector mask register Vmt.

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 066 | 0 | M2 | T2 | 02 | Vmt3 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

Vm[x] = Va[x] == Vb[x]

**Operation:**

**For each vector element**

if signed Va equals signed Vb

Vm = true

else

Vm = false

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSEQS – Set if Equal Scalar

Synopsis

Vector register set. Vm = Va == Rb

**Description**

All elements of a vector are compared for equality to a scalar value. If equal a one is written to the output vector mask register, otherwise a zero is written to the output mask register.

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 076 | 0 | M2 | T2 | 02 | Vmt3 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

Vm[x] = Va[x] == Rb[x]

**Operation:**

**For each vector element**

if signed Va equals signed Rb

Vm = true

else

Vm = false

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSGE – Set if Greater or Equal

Synopsis

Vector register set. Vm = Va >= Vb

**Description**

Two vector registers (Va and Vb) are compared for greater or equal and the comparison result is placed in the target vector mask register Vmt.

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 066 | 0 | M2 | T2 | 32 | Vmt3 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

Vm[x] = Va[x] >= Vb[x]

**Operation:**

**For each vector element**

if signed Va greater than or equal signed Vb

Vm = true

else

Vm = false

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSGES – Set if Greater or Equal Scalar

Synopsis

Vector register set. Vm = Va >= Rb

**Description**

All elements of a vector are compared for greater or equal to a scalar value. If the condition is true a one is written to the output vector mask register, otherwise a zero is written to the output mask register.

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 076 | 0 | M2 | T2 | 32 | Vmt3 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

Vm[x] = Va[x] >= Rb

**Operation:**

**For each vector element**

if signed Va greater than or equal signed Rb

Vm = true

else

Vm = false

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSHL – Shift Left

Synopsis

Vector shift left.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0Ch6 | S | M2 | S | A | Vt5 | Amt5 | Va5 | 01h6 |

**Description**

Elements of the vector are shifted left. The least significant bits are loaded with the value zero.

**Operation**

For x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] << amt

**Exceptions:** none

|  |  |  |
| --- | --- | --- |
| S2 | Amount Field |  |
| 0 | general purpose register |  |
| 1 | vector register |  |
| 2 | immediate |  |
| 3 | reserved |  |

### VSHLV – Shift Vector Left

Synopsis

Vector shift left.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 10h6 | ~ | M2 | 02 | Vt5 | Amt5 | Va5 | 01h6 |

**Description**

Elements of the vector are transferred upwards to the next element position. The first is loaded with the value zero.

**Operation**

For x = VL-1 to Amt

Vt[x] = Va[x-amt]

For x = Amt-1 to 0

Vt[x] = 0

**Exceptions:** none

### VSHR – Shift Right

Synopsis

Vector shift left.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0Dh6 | S | M2 | S | A | Vt5 | Amt5 | Va5 | 01h6 |

**Description**

Elements of the vector are shifted right. The most significant bits are loaded with the value zero.

**Operation**

For x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] >> amt

**Exceptions:** none

|  |  |  |
| --- | --- | --- |
| S2 | Amount Field |  |
| 0 | general purpose register |  |
| 1 | vector register |  |
| 2 | immediate |  |
| 3 | reserved |  |

### VSHRV – Shift Vector Right

Synopsis

Vector shift right.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 11h6 | ~ | M2 | 02 | Vt5 | Amt5 | Va5 | 01h6 |

**Description**

Elements of the vector are transferred downwards to the next element position. The last is loaded with the value zero.

**Operation**

For x = 0 to VL-Amt

Vt[x] = Va[x+amt]

For x = VL-Amt +1 to VL-1

Vt[x] = 0

**Exceptions:** none

### VSIGN – Sign

Synopsis

Vector register sign value. Vt = Va < 0 ? –1 : Va = 0 ? 0 : 1

**Description**

The sign of a vector register is placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 26h6 | ~ | M2 | T2 | Vt5 | 05 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] < 0 ? –1 : Va[x]=0 ? 0 : 1

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSLT – Set if Less Than

Synopsis

Vector register set. Vm = Va < Vb

**Description**

Two vector registers (Va and Vb) are compared for less than and the comparison result is placed in the target vector mask register Vmt.

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 066 | 0 | M2 | T2 | 22 | Vmt3 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

Vm[x] = Va[x] < Vb[x]

**Operation:**

**For each vector element**

if signed Va less than signed Vb

Vm = true

else

Vm = false

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSNE – Set if Not Equal

Synopsis

Vector register set. Vm = Va != Vb

**Description**

Two vector registers (Va and Vb) are compared for inequality and the comparison result is placed in the target vector mask register Vmt.

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 066 | 0 | M2 | T2 | 12 | Vmt3 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

Vm[x] = Va[x] <> Vb[x]

**Operation:**

**For each vector element**

if signed Va not equal signed Vb

Vm = true

else

Vm = false

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSUB - Subtract

Synopsis

Vector register add. Vt = Va - Vb

**Description**

Two vector registers (Va and Vb) are subtracted and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 056 | ~ | M2 | T2 | Vt5 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] - Vb[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSUBRS – Subtract from Scalar

Synopsis

Vector register subtract. Vt = Rb - Va

**Description**

A vector and a scalar (Va and Rb) are subtracted and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 16h6 | ~ | M2 | T2 | Vt5 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Rb - Va[x]

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSUBS – Subtract Scalar

Synopsis

Vector register subtract. Vt = Va - Rb

**Description**

A vector and a scalar (Va and Rb) are subtracted and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 15h6 | ~ | M2 | T2 | Vt5 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] - Rb

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSUN – Set if Unordered

Synopsis

Vector register set. Vm = Va ? Vb

**Description**

Two vector registers (Va and Vb) are compared and the comparison result is placed in the target vector mask register Vmt.

**Instruction Format**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 066 | 1 | M2 | T2 | 32 | Vmt3 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

Vm[x] = Va[x] ? Vb[x]

**Operation:**

**For each vector element**

if is unordered Va or Vb

Vm = true

else

Vm = false

Operand Type

|  |  |  |
| --- | --- | --- |
| T2 | Operand Type |  |
| 0 | Integer |  |
| 1 | Float double |  |
| 2 | reserved |  |
| 3 | reserved |  |

### VSYNC -Synchronize

Description:

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

Instruction Format:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 26 | 25 24 | 23 21 | 20 16 | 15 11 | 10 6 | 5 0 |
| 36h6 | ~2 | ~3 | ~5 | ~5 | ~5 | 01h6 |

Clock Cycles: varies depending on queue contents

### VXCHG - Exchange

Synopsis

Vector register exchange. Va = Vb;Vb= Va

**Description**

Exchange two vector registers (Va and Vb)

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0B6 | Vm3 | 02 | Va5 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL - 1

if (Vm[x])

Vb[x] = Va[x]

Va[x] = Vb[x]

### VXOR – Bitwise Exclusive Or

Synopsis

Vector register bitwise or. Vt = Va ^ Vb

***Description***

Two vector registers (Va and Vb) are exclusive or’ed together and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0Ah6 | Vm3 | 02 | Vt5 | Vb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] ^ Vb[x]

### VXORS – Bitwise Exclusive Or with Scalar

Synopsis

Vector register bitwise and. Vt = Va ^ Rb

***Description***

A vector register (Va) is bitwise exclusive ord’ed with a scalar register and placed in the target vector register Vt.

**Instruction Format**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1Ah6 | Vm3 | 02 | Vt5 | Rb5 | Va5 | 01h6 |

**Operation**

for x = 0 to VL-1

if (Vm[x]) Vt[x] = Va[x] ^ Rb[x]

# Opcode Tables

## Major Opcode (inst. bits 0 to 5)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | BRK | {VECTOR} | {RR} | BccR | ADDI | SUBI | CMPI | CMPUI | ANDI | ORI | XORI | {FLOAT} |  | REX | CSR | EXEC |
| 1x | LH | LHU | LW | LB | SH | SB | SW | SWC | JAL | CALL | QOPI | Scc | NOP | LWR | CACHE |  |
| 2x | LC | LCU | {BITFIELD} | LBU | SC | CAS | BBc | | JMP | RET | LBO | LCO | MODUI | MODSUI | MODI | {AMO} |
| 3x | Bcc | | BEQ# | | CHK | LHO | LV | SV | MULUI | MULSUI | MULI |  | DIVUI | DIVSUI | DIVI |  |

## Major Funct (inst. bits 26 to 31)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | {BCD} | {R1} | {Bitfield} | BMM | ADD | SUB | CMP | CMPU | AND | OR | XOR |  | NAND | NOR | XNOR | {shift} |
| 1x | LHX | LHUX | LWX | LBX | SHX | SBX | SWX | SWCX | LEAX | CMP | CMPU | MUX |  | LWRX | CACHEX | {shiftb} |
| 2x | LCX | LCUX | MOV | LBUX | SCX | CASX | LVWS | SVWS | CMOVEQ | CMOVNE | LBOX | LCOX | MIN | MAX | MAJ | {shiftc} |
| 3x | SEI / CLI | WAIT | RTI | VMOV | CHK | LHOX | LVX | SVX | MULU | MULSU | MUL |  | DIVMODU | DIVMODSU | DIVMOD | {shifth} |

## Float Funct (inst. bits 26 to 31)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x |  |  |  |  | FADD | FSUB | FCMP |  | FMUL | FDIV |  |  |  |  |  |  |
| 1x | FMOV |  | FTOI | ITOF | FNEG | FABS | FSIGN | FMAN | FNABS | FCVTSD |  | FCVTSQ | FSTAT |  |  |  |
| 2x | FTX | FCX | FEX | FDX | FRM |  |  |  |  | FCVTDS |  |  |  |  |  |  |
| 3x |  |  |  |  |  |  | FSYNC |  |  |  |  |  |  |  |  |  |

## R1 (inst. bits 21 to 25)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | CNTLZ | CNTLO | CNTPOP |  | ABS | NOT | REDOR |  |  |  |  |  |  |  |  |  |
| 1x | MEMDB | MEMSB | SYNC |  |  |  |  |  |  |  |  |  |  |  |  |  |

## Shift (inst. bits 22 to 25)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | SHL | SHR | ASL | ASR | ROL | ROR |  |  | SHLI | SHRI | ASLI | ASRI | ROLI | RORI |  |  |

## Vector Funct (inst. bits 26 to 31)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | x0 | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | x9 | xA | xB | xC | xD | xE | xF |
| 0x | VCMPRSS | VCIDX | VSCAN | VABS | VADD | VSUB | VSxx | VSxxS | VAND | VOR | VXOR | VXCHG | VSHL | VSHR | VASR |  |
| 1x | VSHLV | VSHRV |  |  | VADDS | VSUBS | VSUBRS | VSxxSU | VANDS | VORS | VXORS |  |  |  |  |  |
| 2x | VBITS2V | V2BITS | VEINS / VMOVSV | VEX / VMOVS | VFLT2INT | VINT2FLT | VSIGN | VSxxU | VCNTPOP |  | VMULS |  |  |  | VDIVS |  |
| 3x | VMAND | VMOR | VMXOR | VMXNOR | VMPOP | VMFILL | VMFIRST | VMLAST |  |  | VMUL |  |  |  | VDIV |  |

# Appendix

## Reducing the size of the core.

The vector instructions add considerably to the size of the core consuming approximately 40,000 LUTs. IF they are not required the core should be built without the vector instructions.

* Only for the FT64 core. Register renaming adds considerably to the size of the core. It uses approximately 30,000 LUTs to implement register renaming. The core (FT64a) may be built without register renaming by setting the RENAME parameter to zero.

Architectural Register vs Physical Registers

Architectural registers are the registers visible to the programmer as part of the programming model. Physical registers are the registers physically present in the machine’s hardware. There are substantially more physical registers than there are architectural ones. For FT64 there are 32 registers visible to be programmed which are supported by 64 physical registers.

Register Renaming

The core maintains an eight entry deep history file for register rename mappings and register in use flags. The depth of the history file corresponds to the number of entries in the re-order buffer. At most a new map will be needed for each re-order buffer entry. Typically the history file is cycled through at half or less the rate of the instruction queue as approximately 50% of instructions don’t have target registers.

The core can allocate up to two registers as target registers for every pair of instructions queued. If there are no target registers available the core stalls until previous instructions have made more target registers available.

Instruction Cache Miss

During a cache miss the core streams NOP operations to the instruction fetch unit while the core is waiting for the instruction cache to load. The program counters are not incremented however, and they remain at the value when the cache miss occurred.

Branches

Branches store the target address in iqentry\_a0 the immediate constant field of the queue. The target address has to be stored somewhere in the instruction queue so that it may be used to update the branch target buffer later. It can’t be stored in the result field, and it can’t be stored in one of the other argument fields. Arg0 is the only place it can be stored safely.

Branches are evaluated after the following instruction enqueues so that false branch mispredictions don’t occur. Mispredict logic looks at the address of the instruction following the branch to ensure that the branch address was predicted correctly.

## Configuration Defines

Q2VECTORS

* allows queuing two vector elements per cycle, rather than just one
* increases code size and complexity
* not known to be working

## Parameters

SUP\_TXE

* default 0
* enables support for the call target exception

SUP\_VECTOR

* default 1
* enables support for vector instructions

## Instructions Supported Only on ALU #0

The following less frequently used instructions are only supported on ALU #0 in order to reduce the size of the core.

* + division and remainder instructions (DIV,DIVSU,DIVU,MOD,MODSU, MODU)
  + bit-field instructions (BFCLR, BFSET, BFCHG, BFINS, BFINSI, BFEXT, BFEXTU)
    - these are rarely used instructions
  + shift instructions (ASR, SHL, SHR)
    - The shift instructions use barrel shifters to shift by any amount in a single clock cycle and so are relatively resource expensive compared to how often they are used.
  + indexed memory loads / stores (LBX, LHX, LHUX, LWX, SBX, SHX, SWX)
    - since indexed memory instructions are infrequently used they are supported only on alu #0.
  + CSR instruction
    - CSR instructions are rarely used. They often also have synchronization issues as there is no bypassing for the CSR registers. Since they typically require synchronization operations there is no benefit to having multiple CSR instructions executing at the same time.

# WISHBONE Compatibility Datasheet

The FT64 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  MASTER, READ-MODIFY-WRITE  MASTER, BLOCK READ / WRITE, BURST READ (FIXED ADDRESS) | |
| Data port, size:  Data port, granularity:  Data port, maximum operand size:  Data transfer ordering:  Data transfer sequencing | 64 bit  8 bit  64 bit  Little Endian  any (undefined) | |
| Clock frequency constraints: |  | |
| Supported signal list and cross reference to equivalent WISHBONE signals | Signal Name:  ack\_i  adr\_o(31:0)  clk\_i  dat\_i(63:0)  dat\_o(63:0)  cyc\_o  stb\_o  wr\_o  sel\_o(7:0)  cti\_o(2:0)  bte\_o(1:0) | WISHBONE Equiv.  ACK\_I  ADR\_O()  CLK\_I  DAT\_I()  DAT\_O()  CYC\_O  STB\_O  WE\_O  SEL\_O  CTI\_O  BTE\_O |
| Special Requirements: |  | |