rfPhoenix

(c) 2022 Robert Finch

## Overview

rfPhoenix is a core aimed at GPGPU tasks. It is a barrel processor with fine-grained threading using four threads of execution.

### Instruction Cache

The instruction cache is four-way set associative with a 64B line size. The cache is 32kB in size. The instruction cache is accessed in a pipelined fashion with a latency of three clock cycles.

### Pipeline

Every clock cycle, a thread is scheduled for execution and an instruction fetch for the thread begun. The thread is marked busy at instruction fetch. Three clock cycles later, the instruction is placed in a fetch buffer. If there was a cache miss the thread is stalled until the cache is loaded. Other threads continue to run. Register data fetch begins with the instruction loaded into the fetch buffer. There is a three-clock cycle latency for access to the register file. While register data is being looked up the instruction is decoded and placed in an execution buffer. There is a separate execution buffer for each thread so that execution of long running instruction may overlap with execution of instructions by other threads.

### Interrupts

The interrupt controller may designate a specific thread to handle an interrupt.

## Register File

There are 64 general purpose registers. R0 is special in that it may be written once with the value zero. After a zero is written to r0 it can no longer be updated.

Registers designated as vector mask registers are also special in that one of them may be specified as a mask register in the instruction.

Link registers are special in that they may be specified by call, jump, or unconditional branch instructions as the target register to store the next instruction pointer value in.

Alternate stack pointers are reference via register code 31 for different operating modes. For example in machine mode a stack pointer reference will translate to register 46.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Reg | Mnen. |  |  | Reg |  |  |
| 0 | Zero |  |  | 32 | VM0 | Vector Mask |
| 1 | A0 | Return Value |  | 33 | VM1 |  |
| 2 | A1 |  |  | 34 | VM2 |  |
| 3 | T0 | Temporaries |  | 35 | VM3 |  |
| 4 | T1 |  |  | 36 | VM4 |  |
| 5 | T2 |  |  | 37 | VM5 |  |
| 6 | T3 |  |  | 38 | VM6 |  |
| 7 | T4 |  |  | 39 | VM7 |  |
| 8 | T5 |  |  | 40 | LC | Loop Counter |
| 9 | T6 |  |  | 41 | LR1 | Link Registers |
| 10 | T7 |  |  | 42 | LR2 |  |
| 11 | S0 | Register Vars |  | 43 | R43 |  |
| 12 | S1 |  |  | 44 | SSP | Alternate stack pointers |
| 13 | S2 |  |  | 45 | HSP |  |
| 14 | S3 |  |  | 46 | MSP |  |
| 15 | S4 |  |  | 47 | ISP |  |
| 16 | S5 |  |  | 48 | R48 |  |
| 17 | S6 |  |  | 49 | R49 |  |
| 18 | S7 |  |  | 50 | R50 |  |
| 19 | S8 |  |  | 51 | R51 |  |
| 20 | S9 |  |  | 52 | R52 |  |
| 21 | A2 | Arguments |  | 53 | R53 |  |
| 22 | A3 |  |  | 54 | R54 |  |
| 23 | A4 |  |  | 55 | R55 |  |
| 24 | A5 |  |  | 56 | R56 |  |
| 25 | A6 |  |  | 57 | R57 |  |
| 26 | A7 |  |  | 58 | R58 |  |
| 27 | GP3 |  |  | 59 | R59 |  |
| 28 | GP2 |  |  | 60 | R60 |  |
| 29 | GP1 |  |  | 61 | R61 |  |
| 30 | FP | Frame Pointer |  | 62 | R62 |  |
| 31 | SP | Stack Pointer |  | 63 | R63 |  |

## Vector Register File

The vector register file contains 63 general purpose vector registers. The vector registers are unified and may contain either integer or floating-point data. Vector register #63 is used to store the exception instruction pointer stack.

## Status and Control Registers, CSRs

S\_PTBR – CSR 0x1003 - shared

This register contains the base address of the page table, which must be a multiple of 65536. Also included in this register is table parameters depth and type.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31 16 | 15 11 | 10 8 | 7 6 | 5 | 4 | 3 1 | 0 |
| Page Table Address31..16 | ~5 | LVL3 | AL2 | ~ | S | ~ | Type |

Type: 0 = hash page table, 1 = hierarchical page table

S: 1=software managed TLB miss, 0 = hardware table walking

LVL3: 0 to 7 levels for hierarchical tables.

AL2: TLB entry replacement algorithm, 0=fixed,1=LRU,2=random,3=reserved

S\_HMASK – CSR 0x1005 - shared

This register contains a mask applied to address hash for hash tables. Ignored for hierarchical tables.

|  |
| --- |
| 31 0 |
| HMask32 |

S\_ASID – CSR 0x101F

This register contains the address space identifier (ASID) or memory map index (MMI). The ASID is used in this design to select (index into) a memory map in the paging tables. Only the low order ten bits of the register are implemented.

M\_HARTID - CSR 0x3001 - shared

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. The low order four bits of this register indicate the thread number. Although this register is shared a different value will be returned to each thread.

M\_TICK - CSR 0x3002 - shared

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

This register is shared between all threads.

M\_BADADDR - CSR 0x3007

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

M\_TVEC – CSR 0x3030 to 0x3033 -shared

These registers contain the address of the exception handling routine for a given operating level. The lower bits of the exception address are determined from the operating level. TVEC[0] to TVEC[2] are used by the REX instruction.

|  |  |
| --- | --- |
| Reg # |  |
| 0x3030 | TVEC[0] |
| 0x3031 | TVEC[1] |
| 0x3032 | TVEC[2] |
| 0x3033 | TVEC[3] |

M\_EIP – CSR 0x3038

This register contains the exception instruction pointer for the current exception level. This register is read-only.

M\_PLSTACK – CSR 0x303F

This register contains an eight-entry stack of privilege levels. Each stack entry occupies eight bits. When an exception occurs, this register is shifted to the left by eight bits and the low order bits are set to the highest privilege level, 0xFF. When an [RTI](#_RTI_–_Return) instruction is executed this register is shifted to the right by eight bits restoring the previous privilege level. Note that only the low half of the register is available as a CSR. The upper half is inaccessible. Only the low order bits of the register need to be manipulated. There is a separate register for each thread.

M\_PMSTACK – CSR 0x3040

This register contains an eight-entry operating mode and interrupt mask stack. When an exception or interrupt occurs, this register is shifted to the left by eight bits and the low order bits are set according to the exception mode, when an [RTI](#_RTI_–_Return) instruction is executed this register is shifted to the right by eight bits. On RTI the last stack entry is set to $CE masking all interrupts on stack underflow. The low order eight bits represent the current operating mode and interrupt mask. Note that only the low half of the register is available as a CSR. The upper half is inaccessible. Only the low order bits need to be manipulated. There is a separate register for each thread.

This register supports the [CSRRS](#_CSRx_–_Control) and [CSRRC](#_CSRx_–_Control) instructions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 8 | 7 6 | 5 4 | 3 1 | 0 |
| <seven more groups> | OM | ~2 | IPL | IM |

OM = operating mode, 0 to 3

IPL = interrupt priority level

IM = interrupt mask

M\_TIME – CSR 0x?FE0 - shared

The TIME register corresponds to the low order 32 bits of 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. This register is 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 bits represent the fraction of one second. The upper 32 bits present in the M\_TIMESEC register represent seconds passed. For example, if the tm\_clk\_i frequency is 100MHz the low order 32 bits should count from 0 to 99,999,999 then cycle back to 0 again. When the low order 32 bits cycle back to 0 again, the upper 32 bits in M\_TIMESEC are incremented. The upper 32 bits of the register represent the number of seconds passed since an arbitrary point in the past.

This register is available to all operating modes, however it may be written only from machine mode.

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.

M\_TIMESEC – CSR 0x?FE1- shared

This register contains the wall clock time in seconds. It is the upper 32-bits of a 64-bit value.

# Operating Modes

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

A subset of instructions is limited to machine mode.

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

# Exceptions

## External Interrupts

There is little difference between an externally generated exception and an internally generated one. An externally caused exception will set the exception cause code for the currently fetched instruction.

There are eight priority interrupt levels for external interrupts. When an external interrupt occurs the mask level is set to the level of the current interrupt. A subsequent interrupt must exceed the mask level to be recognized. There is a bit vector input to the core indicating the thread to use for interrupt processing.

## Effect on Machine Status

The operating mode is always switched to machine mode on exception. It is 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.

## Exception Stack

The operating mode, interrupt enable bits and privilege level 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. The exception stack is split between the plStack and pmStack CSR registers.

## Exception Vectoring

Exceptions are handled through a vector table. The vector table has four entries, one for each operating level the thread may be running at. The location of the vector table is determined by TVEC[3]. If the thread is operating at mode three for instance and an interrupt occurs vector table address number three is used for the interrupt handler. Note that the interrupt automatically switches the thread to operating mode three. 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[3] contains $F…FC0000) |  |
| 0 | $F…FC0000 | Handler for operating level zero |
| 1 | $F…FC0020 |  |
| 2 | $F…FC0040 |  |
| 3 | $F…FC0060 |  |

## Reset

Reset is treated as an exception. The reset routine should exit using an RTI instruction. The PL and PM stack registers must be set appropriately for the return.

The core begins executing instructions at address $FFFD0000. All registers are in an undefined state.

## Precision

Exceptions in rfPhoenix are precise. They are processed according to program order of the instructions. If an exception occurs during the execution of an instruction, then an exception field is set in the execute buffer. The exception is processed when the instruction commits which happens in program order.

## Exception Cause Codes

The following table outlines the cause code for a given purpose. These codes are specific to rfPhoenix. 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 |  |  | no exception |  |
| 1 | IBE | x | instruction bus error |  |
| 2 | EXF | x | Executable fault |  |
| 4 | TLB | x | tlb miss |  |
| 5 | DCM | X | Data cache miss |  |
|  |  |  | FMTK Scheduler |  |
| 128 |  | e |  |  |
| 129 | KRST | e | Keyboard reset interrupt |  |
| 130 | MSI | e | Millisecond Interrupt |  |
| 131 | TICK | e |  |  |
| 156 | KBD | e | Keyboard interrupt |  |
| 157 | GCS | e | Garbage collect stop |  |
| 158 | GC | e | Garbage collect |  |
| 159 | TSI | e | FMTK Time Slice Interrupt |  |
| 3 |  |  | Control-C pressed |  |
| 20 |  |  | Control-T pressed |  |
| 26 |  |  | Control-Z pressed |  |
|  |  |  |  |  |
| 32 | SSM | x | single step |  |
| 33 | DBG | x | debug exception |  |
| 34 | TGT | x | call target exception |  |
| 35 | MEM | x | memory fault |  |
| 36 | IADR | x | bad instruction address |  |
| 37 | UNIMP | x | unimplemented instruction |  |
| 38 | FLT | x | floating point exception |  |
| 39 | CHK | x | bounds check exception |  |
| 40 | DBZ | x | divide by zero |  |
| 41 | OFL | x | overflow |  |
|  |  |  |  |  |
| 47 |  |  |  |  |
| 48 | ALN | x | data alignment |  |
| 49 | KEY | x | memory key fault |  |
| 50 | DWF | x | Data write fault |  |
| 51 | DRF | x | data read fault |  |
| 52 | SGB | x | segment bounds violation |  |
| 53 | PRIV | x | privilege level violation |  |
| 54 | CMT | x | commit timeout |  |
| 55 | BT | x | branch target |  |
| 56 | STK | x | stack fault |  |
| 57 | CPF | x | code page fault |  |
| 58 | DPF | x | data page fault |  |
| 59 | LVL | x | level error |  |
| 60 | DBE | x | data bus error |  |
| 61 | PMA | x | physical memory attributes check fail |  |
| 62 | NMI | x | Non-maskable interrupt |  |
| 63 | BRK |  | BRK instruction encountered |  |
|  |  |  |  |  |
| 200 | PFX | x | Too many instruction prefixes |  |
| 225 | FPX\_IOP | x | Floating point invalid operation |  |
| 226 | FPX\_DBZ | x | Floating point divide by zero |  |
| 227 | FPX\_OVER | x | floating point overflow |  |
| 228 | FPX\_UNDER | x | floating point underflow |  |
| 229 | FPX\_INEXACT | x | floating point inexact |  |
| 231 | FPX\_SWT | x | floating point software triggered |  |
| 237 | RTI | x | RTI when no interrupt active |  |
| 238 | IRQ | x | Hardware interrupt |  |
| 239 |  |  | Software exception handling |  |
| 240 | SYS |  | Call operating system (FMTK) |  |
| 241 |  |  | FMTK Schedule interrupt |  |
| 242 | TMR | x | system timer interrupt |  |
| 243 | GCI | x | garbage collect interrupt |  |
| 253 | RST | x | reset |  |
| 254 | NMI | x | non-maskable interrupt |  |
| 255 | PFI |  | reserved for poll-for-interrupt instruction |  |

# Memory Management

## Regions

In any processing system there are typically several different types of storage assigned to different physical address ranges. These include memory mapped I/O, MMIO, DRAM, ROM, configuration space, and possibly others. rfPhoenix has a region table that supports up to eight separate regions.

The region table is a list of region entries. Each entry has a start address, an end address, an access type field, and a pointer to the PMT, page management table. To determine legal access types, the physical address is searched for in the region table, and the corresponding access type returned. The search takes place in parallel for all eight regions.

Once the region is identified the access rights for a particular page within the region can be found from the PMT corresponding to the region.

## PMA - Physical Memory Attributes Checker

### Overview

The physical memory attributes checker is a hardware module that ensures that memory is being accessed correctly according to its physical attributes.

Physical memory attributes are stored in an eight-entry region table. This table includes the address range the attributes apply to and the attributes themselves. Address ranges are resolved only to bit four of the address. Meaning the granularity of the check is 16 bytes.

Most of the entries in the table are hard-coded and configured when the system is built. However, they may be modified at the address range $FF9F0xxx.

Physical memory attributes checking is applied in all operating modes.

### Region Table Description

|  |  |  |  |
| --- | --- | --- | --- |
| Address | Bits |  |  |
| 00 | 64 | start | start address bits 4 to 67 of the physical address range |
| 10 | 64 | nd | end address bits 4 to 67 of the physical address range |
| 20 | 18 | pmt | associated PMT address |
| 30 | 64 | cta | card table address |
| 40 | 19 | at | memory attributes |
| 50 to 70 | … | … | not used |
| … | … | … | 7 more register sets |

### PMT Address

The PMT address specifies the location of the associated PMT. Only the low order 18 bits of this value are significant. The high order bits of the PMT table address are fixed at $F..FD.

### CTA – Card Table Address

The card table address is used during the execution of the store pointer, STPTR instruction to locate the card table.

### Attributes

|  |  |  |
| --- | --- | --- |
| Bitno |  |  |
| 0 | X | may contain executable code |
| 1 | W | may be written to |
| 2 | R | may be read |
| 3 | C | may be cached |
| 4-6 | G | granularity   |  |  | | --- | --- | | G |  | | 0 | byte accessible | | 1 | wyde accessible | | 2 | tetra accessible | | 3 | octa accessible | | 4 | hexi accessible | | 5 to 7 | reserved | |
| 7 | ~ | reserved |
| 8-15 | T | device type (rom, dram, eeprom, I/O, etc) |
| 16-18 | S | number of times to shift address to right and store for telescopic STPTR stores. |

## Page Tables

### Intro

Page tables are part of the memory management system used map virtual addresses to real physical addresses. There are several types of page tables. Hierarchical page tables are probably the most common. Almost all page tables map only the upper bits of a virtual address, called a page. The lower bits of the virtual address are passed through without being altered. The page size often 4kB which means the low order 12-bits of a virtual address will be mapped to the same 12-bits for the physical address.

### Hierarchical Page Tables

Hierarchical page tables organize page tables in a multi-level hierarchy. They can map the entire virtual address range. At the topmost level a register points to a page directory, that page directory points to a page directory at a lower level until finally a page directory points to a page containing page table entries. To map an entire 64-bit virtual address range approximately five levels of tables are required.

A picture of multi-level page tables


### Inverted Page Tables

An inverted page table is a table used to store address translations for memory management. The idea behind an inverted page table is that there is a fixed number of pages of memory no matter how it is mapped. It should not be necessary to provide for a map of every possible address, only addresses that correspond to real pages of memory. Each page of memory can be allocated only once. It is either allocated or it is not. Compared to a non-inverted paged memory management system where tables are used to map potentially the entire address space an inverted page table uses less memory. There is typically only a single inverted page table supporting all applications in the system. This is a different approach than a non-inverted page table which may provide separate page tables for each process.

### The Simple Inverted Page Table

The simplest inverted page table contains only a record of the virtual address mapped to the page, and the index into the table is used as the physical page number. There are only as many entries in the inverted page table as there are physical pages of memory. A translation can be made by scanning the table for a matching virtual address, then reading off the value of the table index. The attraction of an inverted page table is its small size compared to the typical hierarchical page table. Unfortunately, the simplest inverted page table is not practical when there are thousands or millions of pages of memory. It simply takes too long to scan the table. The alternative solution to scanning the table is to hash the virtual address to get a table index directly.

Diagram of Inverted Page Table


### Hashed Page Tables

#### Hashed Table Access

Hashes are great for providing an index value immediately. The issue with hash functions is that they are just a hash. It is possible that two different virtual address will hash to the same value. What is then needed is a way to deal with these hash collisions. There are a couple of different methods of dealing with collisions. One is to use a chain of links. The chain has each link in the chain pointing the to next page table entry to use in the event of a collision. The hash page table is slightly more complicated then as it needs to store links for hash chains. The second method is to use open addressing. Open addressing calculates the next page table entry to use. The calculation may be linear, quadratic or some other function dreamed up. A linear probe simply chooses the next page table entry in succession from the previous one if no match occurred. Quadratic probing calculates the next page table entry to use based on squaring the count of misses.

### Shared Memory

Another issue to deal with is shared memory. Sometimes applications share memory with other apps for communication purposes, and to conserve memory space where there are common elements. With a hierarchical paged memory management system, it is easy to share memory, just modify the page table entry to point to the same physical memory as is used by another process. With an inverted page table having only a single entry for each physical page is not sufficient to support shared memory. There needs to be multiple page table entries available for some physical pages but not others because multiple virtual addresses might map to the same physical address. One solution would be to have multiple buckets to store virtual addresses in for each physical address. However, this would waste a lot of memory because much of the time only a single mapped address is needed. There must be a better solution. Rather than reading off the table index as the physical page number, the association of the virtual and physical address can be stored. Since we now need to record the physical address multiple times the simple mechanism of using the table index as the physical page number cannot be used. Instead, the physical page number needs to be stored in the table in addition to the virtual page number.

That means a table larger than the minimum is required. A minimally sized table would contain only one entry for each physical page of memory. So, to allow for shared memory the size of the table is doubled. This smells like a system configuration parameter.

### rfPhoenix Page Tables

### rfPhoenix Hash Page Table Setup

#### Hash Page Table Entries - HPTE

We have determined that a page table entry needs to store both the physical page number and the virtual page number for the translations. To keep things simple, the page table stores only the information needed to perform an address translation. Other bits of information are stored in a secondary table called the page management table, PMT. The author did a significant amount of juggling around the sizes of various fields, mainly the size of the physical and virtual page numbers. Finally, the author decided on a 64/128-bit HPTE format. Note that the first part of the HPTE has the same format as a PTE.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31 22 | | | | 21 | 20 | | 18 | 17 | | 16 | 15 0 |
| V | ~3 | VPN18..16 | PPN18..16 | M | | RWX 3 | | | A | C | Physical Page Number15..0 |
| ASID10 | | | | G | | BC4 | | | | ~ | Virtual Page Number15..0 |
| Physical Page Number50..19 | | | | | | | | | | | |
| Virtual Page Number47..16 | | | | | | | | | | | |

#### Small Hash Page Table Entry – SHPTE

For systems with less than 8GB of physical memory the small hash page table entry may be used. This is a configuration option.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | ~3 | VPN18..16 | PPN18..16 | M | RWX3 | A | C | Physical Page Number15..0 |
| ASID10 | | | | G | BC4 | | ~ | Virtual Page Number15..0 |

Fields Description

|  |  |
| --- | --- |
| V | translation Valid |
| G | global translation |
| MB | page access mask begin |
| ME | page access mask end |
| RWX | readable, writeable, executable |
| C | cachable page |
| ASID | address space identifier |
| BC | bounce count |

The page table does not include everything needed to manage pages of memory. There is additional information such as share counts and privilege levels to take care of, but this information is better managed in a separate table.

#### Page Table Groups – PTG

We want the search for translations to be fast. That means being able to search in parallel. So, PTEs are stored in groups that are searched in parallel for translations. This is sometimes referred to as a clustered table approach. Access to the group should be as fast as possible. There are also hardware limits to how many entries can be searched at once while retaining a high clock rate. So, the convenient size of 512 bits was chosen as the amount of memory to fetch.

A page table group then contains eight page-table entries. All entries in the group are searched in parallel for a match. Note that the entries are searched as the PTG is loaded, so that the PTG group load may be aborted early if a matching PTE is found before the load is finished.

|  |
| --- |
| 63 0 |
| PTE0 |
| PTE1 |
| PTE2 |
| PTE3 |
| PTE4 |
| PTE5 |
| PTE6 |
| PTE7 |

#### Size of Page Table

There are several conflicting elements to deal with, with regards to the size of the page table. Ideally, the page table is small enough to fit into the block RAM resources available in the FPGA. So, about 1/3 of the block RAMs available are dedicated to MMU use. At the same time a multiple of the number of physical pages of memory should be supported to support page sharing and swapping pages to secondary storage. To support swapping pages, double the number of physical entries were chosen. To support page sharing, double that number again. Therefore, a minimum size of a page table would contain at least four times the number of physical pages for entries. By setting the size of the page table instead of the size of pages, it can be worked backwards how many pages of memory can be supported.

For a system using 512k block RAM to store PTEs. 512k / 32 = 16384 entries. 16384 / 4 = 4096 physical pages. Since the RAM size is 512MB, each page would be 512MB/4096 = 128kB. Since half the pages may be in secondary storage, 1GB of address range is available.

Since there are 16,384 entries in the table and they are grouped into groups of eight, there are 2048 PTGs. To get to a page table group fast a hash function is needed then that returns a 11-bit number.

#### Hash Function

The hash function needs to vary with the size of the hash table. The hash table size is effectively controlled by the HMASK field in the page table base register, PTBR. The smallest hash table that may be configured it 128kB or 1024 PTGs. The largest supported table size is 32TB which is enough to support a 56-bit physical address. The hash function needs to reduce the size of a virtual address down to a size corresponding to the size of the hash table. The asid should be considered part of the virtual address. Including the asid an address is 42 bits. The first thing to do is to throw away the lowest sixteen bits as they pass through the MMU unaltered. We now have 26-bits to deal with. We can probably throw away some high order bits too, as a process is not likely to use the full 32-bit address range.

The hash function chosen uses the asid combined with virtual address bits 18 and up. This should space out the PTEs according to the asid. Address bits 16 and 17 select one of four address ranges. the PTG supports eight PTEs. The translations where address bits 16 and 17 are involved are likely consecutive pages that would show up in the same PTG. The hash is the asid aligned with the highest order visible address bits exclusively or’d with address bis 18 and up then masked by the HMASK field of the PTBR.

#### HMASK

The HMASK field of the PTBR is used to mask off hash result bits so that the hash may be applied appropriately for the table size. The lowest order 10 bits of the hash may not be masked off. For example, for a table with 4096 entries The HMASK value would be 3. This would allow 12 of the hash bits to be used as a hash table index.

#### The PTGHASH Instruction

To make developing software easier an instruction, PTGHASH, is available that returns the value of the hash of an address as seen by the processor. The hash value depends on the current ASID, HMASK field of the PTBR and the virtual address.

#### Collision Handling

Quadratic probing of the page table is used when a collision occurs. The next PTG to search is calculated as the hash plus the square of the miss count. On the first miss the PTG at the hash plus one is searched. Next the PTG at the hash plus four is searched. After that the PTG at the hash plus nine is searched, and so on.

#### Finding a Match

Once the PTG to be searched is located using the hash function, which PTE to use needs to be sorted out. The match operation must include both the virtual address bits and the asid, address space identifier, as part of the test for a match. It is possible that the same virtual address is used by two or more different address spaces, which is why it needs to be in the match.

#### Locality of Reference

The page table group may be cached in the system read cache for performance. It is likely that the same PTG group will be used multiple times due to the locality of reference exhibited by running software.

#### Access Rights

To avoid duplication of data the access rights are stored in another table called the PMT for access rights table. The first time a translation is loaded the access rights are looked-up from the PMT. A bit is set in the TLB entry indicating that the access rights are valid. On subsequent translations the access rights are not looked up, but instead they are read from values cached in the TLB.

#### Location of Page Table

rfPhoenix’s hash page table is in the physical address space at $FFAxxxxx. It is a specially dedicated block RAM memory which has two sides. One side is updateable and readable via the vector load tetra-byte and store tetra-byte LDT, STT instructions. The other side is updateable and readable in terms of page groups by the hash page table control logic.

#### Software Support

rfPhoenix has instructions to support a software managed clustered hash page table. Page table groups may be loaded into a vector register using the LDT instruction. The entire group may be searched in a single clock cycle for a translation of the virtual address using the [SHPTENDX](#_SHPTENDX_–_SHPTE) instruction.

To locate the page table group, a hash generator function is available [PTGHASH](#_PTGHASH_–_Compute) which hashes a virtual address and masks it to the correct size for the page table.

### rfPhoenix Hierarchical Page Table Setup

#### Page Size

rfPhoenix uses 16kB memory pages. This size was chosen to allow the L1D$ to be virtually indexed.

#### Page Table Entries - PTE

For hierarchical tables the structure is like that of hashed page tables except that there is no need to store the virtual address. We know the virtual address because it is what is being translated and there is no chance of collisions unlike the hash table. The structure is 64 bits in size. This allows 2048 PTEs to fit into a 16kB page.

#### Page Table Entry Format - PTE

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | LVL3 | ~3 | PPN18..16 | M | RWX3 | A | C | Physical Page Number15..0 |
| ASID10 | | | | G | ~21 | | | |

|  |  |  |
| --- | --- | --- |
| Field | Size | Purpose |
| PPN | 19 | Physical page number |
| C | 1 | 1=cachable |
| A | 1 | 1=accessed |
| X | 1 | 1=executable |
| W | 1 | 1=writeable |
| R | 1 | 1=readable |
| M | 1 | 1=modified |
| V | 1 | 1 if entry is valid, otherwise 0 |
| LVL | 3 | the page table level of the entry pointed to |
| G | 1 | 1=global translation |
| ASID | 10 | Address space identifier |

Note the LVL field for both PTEs and PDEs is in the same position.

#### Page Directory Entries - PDE

A hierarchical table usually consists of multiple levels of pages for the page table. The leaf entries are the PTEs non-leaf entries are page directory entries or PDEs. PDEs are 32-bits in size, therefore 4096 PDEs fit in one 16kB page of memory.

|  |  |  |
| --- | --- | --- |
| V | LVL3 | Physical Page Number27..0 |

|  |  |  |
| --- | --- | --- |
| Field | Size | Purpose |
| PPN | 28 | Page number of next lower page table |
| V | 1 | 1 if entry is valid, otherwise 0 |
| LVL | 3 | the page table level of the entry pointed to |

#### MMU Cache

To improve the performance of PDE lookups. The MMU has a small fully associative cache for PDE lookups.

## TLB – Translation Lookaside Buffer

### Overview

The page map is limited in the translations it can perform because of its size. The solution to allowing more memory to be mapped is to use main memory to store the translations tables.

However, if every memory access required two or three additional accesses to map the address to a final target access, memory access would be quite slow, slowed down by a factor or two or three, possibly more. To improve performance, the memory mapping translations are stored in another unit called the TLB standing for Translation Lookaside Buffer. This is sometimes also called an address translation cache ATC. The TLB offers a means of address virtualization and memory protection. A TLB works by caching address mappings between a real physical address and a virtual address used by software. The TLB deals with memory organized as pages. Typically, software manages a paging table whose entries are loaded into the TLB as translations are required.

The TLB is a cache specialized for address translations. rfPhoenix’s TLB is quite large being five way associative with 1024 entries per way. This choice of size was based on the minimum number of block RAMs that could be used to implement the TLB. On a TLB miss the page table is searched for a translation and if found the translation is stored in one of the ways of the TLB. The way selected is determined either randomly or in a least-recently-used fashion as one of the first four ways. The fifth way may not be updated automatically by a page table search, it must be updated by software.

### Size / Organization

The TLB has 1024 entries per set. The size was chosen as it is the size of one block ram for 32-bit data in the FPGA. This is quite a large TLB. Many systems use smaller TLBs. Typically, systems vary between 64 and 1024 entries. There is not really a need for such a large one, however it is available.

The TLB is organized as a five-way set associative cache. The fifth way may only be updated by software. The fifth way allows translations to be stored that will not be overwritten.

### TLB Entries - TLBE

Closely related to page table entries are translation look-aside buffer, TLB, entries. TLB entries have more fields to provide access counting and keyed access. The additional field are populated from the page management table, PMT.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | LVL3 | VPN3 | PPN3 | M | RWX3 | A | C | Physical Page Number15..0 | |
| ASID10 | | | | G | BC4 | | ~ | VPN15..10 | ~10 |

The following two fields needed only for systems with more than 256MB of memory

|  |
| --- |
| Physical Page Number50..19 |
| Virtual Page Number50..19 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V | N | M | ~10 | | E | AL2 | PCI16 | |
| ACL16 | | | | | | | | Share Count16 |
| Access Count32 | | | | | | | | |
| PL8 | | | | Key24 | | | | |

The TLB entry also contains pointers to the PTE and PMT entries used to update the TLB. The TLB needs this information to be able to update those structures in memory.

### What is Translated

The TLB processes addresses including both instruction and data addresses for all modes of operation. It is known as a *unified* TLB.

### Page Size

Because the TLB caches address translations it can get away with a much smaller page size than the page map can for a larger memory system. 4kB is a common size for many systems. There are some indications in contemporary documentation that a larger page size would be better. In this case the TLB uses 16kB. For a 512MB system (the size of the memory in the test system) there are 32,768 16kB pages.

### Management

The TLB unit may be updated by either software or hardware. This is selected in the page table base register. If software miss handling is selected when a translation miss occurs, an exception is generated to allow software to update the TLB. It is left up to software to decide how to update the TLB. There may be a set of hierarchical page tables in memory, or there could be a hash table used to store translations.

### Accessing the TLB

The TLB is access with the [TLBRD](#_TLBRD_–_Read) and [TLBRW](#_TLBRW_–_Read) instructions. A TLB entry contains too much information to be updated with a single 32-bit register write. Since the information must also be updated atomically to ensure correct operation, the TLB update is done using a vector register.

The low order bits of the Ra value determine which way to update in the TLB if the algorithm is a fixed or LRU way algorithm. Otherwise, a way to update will be selected randomly. When the LRU algorithm is active the most recently used entry is placed in way #0. It may be desirable to bump out entry #3 and replace it with the new entry for LRU operation.

Vector Register Rb contents

|  |  |
| --- | --- |
| Element | 31 0 |
| 0 | PTE31..0 |
| 1 | PTE63..32 |
| 2 | PTE95..64 |
| 3 | PTE127..96 |
| 4 | PMTE31..0 |
| 5 | PMTE63..32 |
| 6 | PMTE95..64 |
| 7 | PMTE127..96 |
| 8 | PTE Address31..0 |
| 9 | PTE Address63..32 |
| 10 | PMTE Address31..0 |
| 11 | PMTE Address63..32 |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |

#### Example TLB Update Routine

|  |  |
| --- | --- |
| \_TLBMap:  ldt v0,0[sp]  ldt a1,64[sp]  tlbrw a0,a1,v0  add sp,sp,68  rts |  |
|  |  |

### TLB Entry Replacement Policies

The TLB supports three algorithms for replacement of entries with new entries on a TLB miss. These are fixed replacement (0), least recently used replacement (1) and random replacement (2). The replacement method is stored in the AL2 bits of the page table base register.

For fixed replacement, the way to update must be specified by a software instruction. Least recently used replacement, LRU, rotates the most recent address translation to the first way and updates by over-writing the value in the third way. Random replacement chooses a way to replace at random.

### Flushing the TLB

The TLB maintains the address space (ASID) associated with a virtual address. This allows the TLB translations to be used without having to flush old translations from the TLB during a task switch.

### Reset

On a reset the TLB is preloaded with translations that allow access to the system ROM.

#### Global Bit

In addition to the ASID the TLB entries contain a bit that indicates that the translation is a global translation and should be present in every address space.

## Page Management

There are typically at least three primary lists that a page may be on. These are the active list, the inactive list, and the free list. Typically, a list link field would be present in the PMTE.

## Instruction Formats

There are relatively few instruction formats. Instructions are 40-bits in size with two exceptions. The NOP instruction is a single byte to allow code to be aligned on any byte boundary. The constant postfix instruction is 24-bits in size as that is all that is needed.

Register Format, Indexed Load / Store Format

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | Func6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | Opcode6 |

Float Register Format

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Rm2 | Tc | Rc6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | Opcode6 |

Immediate Format, Load / Store Format

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | Opcode6 |

Branch

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| Displacement17 | Cnd3 | 0 | Ra6 | 0 | Rb6 | Opcode6 |

Call, Jump

|  |  |  |
| --- | --- | --- |
| 39 8 | 7 6 | 5 0 |
| Target32 | Rt2 | Opcode6 |

Postfix

|  |  |  |
| --- | --- | --- |
| 23 8 | 7 6 | 5 0 |
| Immediate16 | ~2 | Opcode6 |

### Major Opcode

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0  BRK | 1  PFX | 2  R2 | 3 | 4  ADDI | 5  SUBFI | 6  MULI | 7  CSR |
| 1x | 8  ANDI | 9  ORI | 10  XORI | 11  NOP | 12 | 13  CMPI | 14  CMP EQI | 15  CMP NEI |
| 2x | 16  CMP LTI | 17  CMP GEI | 18  CMP LEI | 19  CMP GTI | 20  CMP LTUI | 21  CMP GEUI | 22  CMP LEUI | 23  CMP GTUI |
| 3x | 24  JSR abs | 25  BSR rel | 26  JSR reg | 27 | 28  Bcc | 29  FBcc | 30  FCMP EQI | 31  FCMP NEI |
| 4x | 32 | 33 | 34 | 35 | 36  FCMP LTI | 37  FCMP GEI | 38  FCMP LEI | 39  FCMP GTI |
| 5x | 40 | 41 | 42 | 43 | 44  FMA | 45  FMS | 46  FNMA | 47  FNMS |
| 6x | 48  LDB | 49  LDBU | 50  LDW | 51  LDWU | 52  LDT | 53 | 54 | 55 |
| 7x | 56  STB | 57  STW | 58  STT | 59 | 60 | 61 | 62 | 63  CACHE |

### Major Func

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0 | 1  R1 | 2  VSHUF | 3  VEX | 4  ADD | 5  SUB | 6  MUL | 7  TLB |
| 1x | 8  AND | 9  OR | 10  XOR | 11 | 12  VEINS | 13  CMP | 14  CMP EQ | 15  CMP NE |
| 2x | 16  CMP LT | 17  CMP GE | 18  CMP LE | 19  CMP GT | 20  CMP LTU | 21  CMP GEU | 22  CMP LEU | 23  CMP GTU |
| 3x | 24  SLLI | 25  SRLI | 26  SRAI | 27  SLL | 28  SRL | 29  SRA | 30  FCMP EQ | 31  FCMP NE |
| 4x | 32  VSLLVI | 33  VSRLVI | 34  VSLLV | 35  VSRLV | 36  FCMP LT | 37  FCMP GE | 38  FCMP LE | 39  FCMP GT |
| 5x | 40  SHPTENDX | 41 | 42 | 43 | 44  FADD | 45  FSUB | 46  FMUL | 47 |
| 6x | 48  LDBX | 49  LDBUX | 50  LDWX | 51  LDWUX | 52  LDTX | 53 | 54 | 55 |
| 7x | 56  STBX | 57  STWX | 58  STTX | 59 | 60 | 61 | 62 | 63  CACHEX |

### R1 Func

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x | 0  CNTLZ | 1 | 2  CNTPOP | 3 | 4 | 5 | 6 | 7  PTGHASH |
| 1x | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 2x | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 3x | 24 | 25  RTI | 26  REX | 27 | 28 | 29 | 30 | 31  TLBRW |
| 4x | 32  FFINITE | 33 | 34 | 35  FNEG | 36  FRSQRTE | 37  FRES | 38  FSIGMOID | 39 |
| 5x | 40  ITOF | 41  FTOI | 42  FABS | 43  FNABS | 44  FCLASS | 45  FMAN | 46  FSIGN | 47  FTRUNC |
| 6x | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 7x | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

### Extended Constants

The upper 16 bits of a 32-bit constant may be specified using a postfix instruction which overrides sign extension of the constant.

### Vector Masking

If masking is enabled via the ‘m’ bit in the instruction then the target lanes corresponding to one bits in the mask register are updated. Other lanes of the target register are not affected.

### ADD - Register-Register

**Description:**

Add two registers and place the sum in the target register. All register values are treated as integers.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 04h6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra + Rb

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### ADDI - Register-Immediate

**Description:**

Add a register and an immediate value and place the sum in the target register. All values are treated as signed integers.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 046 |

**Operation:**

Rt = Ra + Imm

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### AND - Register-Register

**Description:**

Bitwise ‘and’ two registers and place the result in the target register. All register values are treated as integers.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 08h6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra & Rb

**Clock Cycles:** 1

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

**Exceptions:** none

### ANDI - Register-Immediate

**Description:**

Bitwise ‘And’ a register and an immediate value and place the sum in the target register. All values are treated as signed integers.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 086 |

**Operation:**

Rt = Ra & Imm

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### BRK – Software Break

**Description**

The exception pointer vector is shifted left once, and the instruction pointer inserted into element zero.

**Instruction Format:** RI

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | 12 | ~ | 326 | 1 | Imm6 | Mask3 | 1 | 636 | 1 | 636 | 026 |

**Operation**

pmStack << 8

plStack << 8

EIP[0] <= IP + 5

**Clock Cycles:** 1

**Exceptions**: none

### CNTLZ – Count Leading Zeros

**Description**:

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

**Instruction Format: R1**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | 0 | 06 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles**: 1

**Execution Units: First** Integer ALU

**Exceptions:** none

### CNTPOP – Count Population

CNTPOP r1,r2

CNTPOP v1,v2

**Description:**

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

**Vector Operation**

for x = 0 to VL - 1

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

else Vt[x] = Vt[x]

**Instruction Format:** R1

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | 0 | 26 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Execution Units: First Integer** ALU

**Exceptions:** none

### CMP – Compare Register

**Description**

Compare two registers and return the relationship between them in a bit vector. The lower 16 bits of the result represent the relationship between integers the upper 16 bits of the result represent the relationship between floats.

**Instruction Format:** RI

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 136 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

Rt = Ra ? Imm

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] ? Imm

else Vt[x] = Vt[x]

**Execution Units:** Integer ALU, Floating Point Unit

**Exceptions**: none

|  |  |
| --- | --- |
| Rt bit | Meaning |
|  | **Integer Compare Results** |
| 0 | = equal |
| 1 | < less than |
| 2 | <= less than or equal |
| 3 | reserved |
| 4 | reserved |
| 5 | < unsigned less than |
| 6 | <= unsigned less than or equal |
| 8 | < > not equal |
| 9 | >= greater than or equal |
| 10 | > greater than |
| 11 | reserved |
| 12 | reserved |
| 13 | unsigned greater than or equal |
| 14 | unsigned greater than |
|  | **Float Compare Results** |
| 16 | = equal |
| 17 | < less than |
| 18 | <= less than or equal |
| 19 | < magnitude less than |
| 20 | unordered |
| 21 | <> not equal |
| 22 | >= greater than or equal |
| 23 | > greater than |
| 24 | >= magnitude greater than or equal |
| 25 | ordered |
|  |  |
|  |  |
| 27 to 31 | zero (reserved) |
|  |  |

### CMP\_EQ - Compare for Equality

**Description:**

The register compare instruction compares two registers as signed integer values for equality and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 146 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

if Ra == Rb

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] == Vb[x] ? 1 : 0

else Vmt[x] = Vmt[x]

### CMP\_EQI - Compare for Equality Immediate

**Description:**

The register compare instruction compares a register and an immediate value as signed integer values for equality and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: RI**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 146 |

**Operation:**

if Ra == Imm

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] == imm ? 1 : 0

else Vmt[x] = Vmt[x]

### CMP\_GE - Compare for Greater or Equal

**Description:**

The register compare instruction compares two registers as signed integer values for greater than or equal and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 176 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

if Ra >= Rb

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] >= Vb[x] ? 1 : 0

else Vmt[x] = Vmt[x]

### CMP\_GEI - Compare for Greater or Equal Immediate

**Description:**

The register compare instruction compares a register and an immediate value as signed integer values for greater than or equal and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: RI**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 176 |

**Operation:**

if Ra >= Imm

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] >= imm ? 1 : 0

else Vmt[x] = Vmt[x]

### CMP\_LE - Compare for Less or Equal

**Description:**

The register compare instruction compares two registers as signed integer values for less than or equal and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 186 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

if Ra <= Rb

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] <= Vb[x] ? 1 : 0

else Vmt[x] = Vmt[x]

### CMP\_LEI - Compare for Less or Equal Immediate

**Description:**

The register compare instruction compares a register and an immediate value as signed integer values for less than or equal and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: RI**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 186 |

**Operation:**

if Ra >= Imm

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] >= imm ? 1 : 0

else Vmt[x] = Vmt[x]

### CMP\_LT - Compare for Less Than

**Description:**

The register compare instruction compares two registers as signed integer values for less than and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 166 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

if Ra < Rb

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] < Vb[x] ? 1 : 0

else Vmt[x] = Vmt[x]

### CMP\_LTI - Compare for Less Than Immediate

**Description:**

The register compare instruction compares a register and an immediate value as signed integer values for less than and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: RI**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 166 |

**Operation:**

if Ra < Imm

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] < imm ? 1 : 0

else Vmt[x] = Vmt[x]

### CMP\_NE - Compare for Not Equal

**Description:**

The register compare instruction compares two registers as signed integer values for inequality and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 156 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

if Ra != Rb

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] != Vb[x] ? 1 : 0

else Vmt[x] = Vmt[x]

### CMP\_NEI - Compare for Not Equal Immediate

**Description:**

The register compare instruction compares a register and an immediate value as signed integer values for inequality and sets the result in the target register. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations if the target is a scaler register each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

**Instruction Format: RI**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 156 |

**Operation:**

if Ra != Imm

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Integer ALU

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] != imm ? 1 : 0

else Vmt[x] = Vmt[x]

### CMPI – Compare Immediate

**Description**

Compare a register and an immediate value and return the relationship between them in a bit vector. The immediate is sign extended to the machine width. The lower 16 bits of the result represent the relationship between integers the upper 16 bits of the result represent the relationship between floats.

**Instruction Format:** RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 136 |

**Operation:**

Rt = Ra ? Imm

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Vt[x] = Va[x] ? Imm

else Vt[x] = Vt[x]

**Execution Units:** Integer ALU, Floating Point Unit

**Exceptions**: none

|  |  |
| --- | --- |
| Rt bit | Meaning |
|  | **Integer Compare Results** |
| 0 | = equal |
| 1 | < less than |
| 2 | <= less than or equal |
| 3 | reserved |
| 4 | reserved |
| 5 | < unsigned less than |
| 6 | <= unsigned less than or equal |
| 8 | < > not equal |
| 9 | >= greater than or equal |
| 10 | > greater than |
| 11 | reserved |
| 12 | reserved |
| 13 | unsigned greater than or equal |
| 14 | unsigned greater than |
|  | **Float Compare Results** |
| 16 | = equal |
| 17 | < less than |
| 18 | <= less than or equal |
| 19 | < magnitude less than |
| 20 | unordered |
| 21 | <> not equal |
| 22 | >= greater than or equal |
| 23 | > greater than |
| 24 | >= magnitude greater than or equal |
| 25 | ordered |
|  |  |
|  |  |
| 27 to 31 | zero (reserved) |
|  |  |

### COM – Ones Complement

**Description:**

Bitwise complement all the bits in the register. 1’s become 0’s and 0’s become 1’s. This is an alternate mnemonic for the [XOR](#_XORI_-_Register-Immediate) function.

**Instruction Format:** RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | FFFFh16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 0A6 |

**Operation**

**Rt = ~Ra**

**Vector Operation**

for x = 0 to VL-1

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

else Vt[x] = Vt[x]

**Exceptions**: none

### LDI – Load Immediate

**Description:**

Load an immediate value into a register. This instruction allows an immediate to be loaded into general purpose registers, alternate stack pointers, vector mask registers, link registers, or the loop counter. It is an alternate mnemonic for the ‘[OR](#_ORI_-_Register-Immediate) instruction where register Ra is assumed to be zero.

**Instruction Format:** RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | 0 | 06 | Tt | Rt6 | 096 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = immediate

**Exceptions:**

**Notes:**

### NEG - Negate

**Description:**

This instruction takes the negative of a value contained in a register Rb. This is an alternate mnemonic for the [SUB](#_SUB_–_Subtract) instruction where register Ra is r0.

**Instruction Format**: R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 56 | Tb | Rb6 | Mask3 | Ta | 06 | Tt | Rt6 | 26 |

**Scalar Operation**

Rt = - Rb

**Vector Operation**

for x = 0 to VL - 1

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

else Vt[x] = Vt[x]

**Notes**

**Exceptions:**

### NOP – No Operation

**Description:**

NOP does not perform any operation. Unlike most instructions, NOP occupies only a single byte. This is to allow NOPs to be used to align code to byte addresses.

**Instruction Format:** NOP

|  |  |
| --- | --- |
| 7 6 | 5 0 |
| ~2 | 0Bh6 |

**Operation:**

< none >

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### OR - Register-Register

**Description:**

Bitwise ‘or’ two registers and place the result in the target register. All register values are treated as integers.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 09h6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra | Rb

**Clock Cycles:** 1

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

**Exceptions:** none

### ORI - Register-Immediate

**Description:**

Bitwise ‘Or’ a register and an immediate value and place the sum in the target register. All values are treated as signed integers.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 096 |

**Operation:**

Rt = Ra | Imm

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### PTGHASH – Compute PTG Hash

**Description:**

This instruction computes a hash value from a value in register Ra.

**Instruction Format**: R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | 0 | 76 | Mask3 | Ta | 06 | Tt | Rt6 | 26 |

**Scalar Operation**

Rt = PTGHASH(Ra)

**Vector Operation**

for x = 0 to VL - 1

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

else Vt[x] = Vt[x]

**Notes**

**Exceptions:**

### SLL – Shift Left Logical

**Description:**

Shift register value Ra to the left by the number of bits specified in Rb and place the result in the target register. Low order bits are set to the fill value specified by the ‘F’ bit in the instruction.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | F | 27h6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra << Rb

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### SLLI – Shift Left Logical Immediate

**Description:**

Shift register value Ra to the left by the number of bits specified by an immediate constant and place the result in the target register. Low order bits are set to the fill value specified by the ‘F’ bit in the instruction.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | F | 27h6 | Tb | Imm6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra << Imm

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### SHPTENDX – SHPTE Index

**Description:**

This instruction searches vector register Rb, which is treated as an array of 8 [SHPTE](#_Small_Hash_Page)s, for a SHPTE whose virtual address matches the one specified by Ra and places the index of the SHPTE into the target register Rt. If the SHPTE is not found -1 is placed in the target register. The index result may vary from -1 to +7. The index of the first found SHPTE is returned (closest to zero). The SHPTE may then be extracted with the [VEX](#_VEX_/_VMOVS) instruction.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 406 | 1 | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles:** 1

**Execution Units: First** Integer ALU

**Operation:**

Rt = Index of (Ra in Rb)

**Exceptions:** none

### SRA – Shift Right Arithmetic

**Description:**

Shift register value Ra to the right by the number of bits specified in Rb and place the result in the target register. High order bits are set to the sign bit of Ra.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 28h6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra >> Rb

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### SRL – Shift Right Logical

**Description:**

Shift register value Ra to the right by the number of bits specified in Rb and place the result in the target register. High order bits are set to the fill value specified by the ‘F’ bit of the instruction.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | F | 28h6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra >> Rb

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### SUB – Subtract

**Description:**

Subtract Rb from Ra and place the difference in the target register Rt.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 05h6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra - Rb

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

### SUBFI – Subtract from Immediate

**Description:**

Subtract a register from a sign extended immediate value and place the difference in the target register.

**Instruction Format:** RI

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 056 |

**Clock Cycles:** 1

**Execution Units:** All ALU’s

**Operation:**

Rt = Immediate - Ra

**Exceptions:** none

**Notes:**

### VEX / VMOVS – Vector Element Extract

**Synopsis**

Vector element extract.

**Description**

A vector register element from Va is transferred into a general-purpose register Rt. The element to extract is identified by Rb. Rb and Rt are scalar registers. This instruction may also be used to broadcast a vector element to all elements of a target vector register.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 036 | 0 | Rb6 | Mask3 | 1 | Ra6 | Tt | Rt6 | 026 |

**Operation**

Rt = Va[Rb]

**Clock Cycles:** 1

**Exceptions**: none

### VSHUF – Vector Shuffle

**Synopsis**

Shuffle vector elements.

**Description**

Vector register elements from Va are transferred into a vector register Rt. The elements to transfer are guided by vector register Rb.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 026 | 1 | Rb6 | Mask3 | 1 | Ra6 | 1 | Rt6 | 026 |

**Operation**

Rt = Va[Rb]

**Clock Cycles:** 1

**Exceptions**: none

### VSLLVI – Vector Shift Left Immediate

**Synopsis**

Shift vector elements to the left.

**Description**

Vector register elements from Ra are transferred into a vector register Rt. The elements to transfer are shifted in position to a higher position by the amount specified as an immediate constant. Lower elements are set to zero.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | 02 | ~ | 326 | 1 | Imm6 | Mask3 | 1 | Ra6 | 1 | Rt6 | 026 |

**Operation**

For n = 0 to VL – 1

If mask[n]

If (n < imm)

Rt[n] = 0

else

Rt[n+imm] = Ra[n]

**Clock Cycles:** 1

**Exceptions**: none

### VSRLVI – Vector Shift Right Immediate

**Synopsis**

Shift vector elements to the right.

**Description**

Vector register elements from Ra are transferred into a vector register Rt. The elements to transfer are shifted in position to a lower position by the amount specified as an immediate constant. Upper elements are set to zero.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | 02 | ~ | 326 | 1 | Imm6 | Mask3 | 1 | Ra6 | 1 | Rt6 | 026 |

**Operation**

For n = 0 to VL – 1

If mask[n]

If (n > VL - imm)

Rt[n] = 0

else

Rt[n] = Ra[n + imm]

**Clock Cycles:** 1

**Exceptions**: none

### XOR - Register-Register

**Description:**

Bitwise exclusive ‘or’ two registers and place the result in the target register. All register values are treated as integers.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 0Ah6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra ^ Rb

**Clock Cycles:** 1

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

**Exceptions:** none

### XORI - Register-Immediate

**Description:**

Bitwise exclusive ‘Or’ a register and an immediate value and place the sum in the target register. All values are treated as signed integers.

**Instruction Format:** R2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 096 |

**Operation:**

Rt = Ra ^ Imm

**Clock Cycles:** 1

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

**Exceptions:** none

**Notes:**

## Floating-Point Operations

### Representation

Floating-point values are represented in 32-bit IEEE-754 format.

|  |  |  |
| --- | --- | --- |
| 31 | 30 23 | 22 0 |
| S | EEEEEEEE | MMMMMMM MMMMMMMM MMMMMMMM |

S: sign 0 = positive, 1 = negative

E..E: exponent 0=subnormal unless M is also zero, FFh with M zero is infinity

M…M: significand, has 1 leading hidden one bit unless the number is subnormal

### Rounding Modes

Two rounding modes are supported, truncate to zero and round ties to nearest even.

### FABS – Absolute Value

**Description:**

This instruction takes the absolute value of a register and places the result in a target register. The values are treated as floating-point values. The sign bit of the number is cleared, no rounding of the number takes place.

**Integer Instruction Format: R1**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | Tb | 426 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

If Ra < 0

Rt = -Ra

else

Rt = Ra

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Rt[x] = Ra[x] < 0 ? -Ra[x] : Ra[x]

else Rt[x] = Rt[x]

**Execution Units:** Float

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### FCLASS – Classify Value

**Description**:

FCLASS classifies the value in register Ra and returns the information as a bit vector in register Rt.

**Integer Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | ~ | 446 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

|  |  |
| --- | --- |
| Bit in Rt | Meaning |
| 0 | 1 = negative infinity |
| 1 | 1 = negative number |
| 2 | 1 = negative subnormal number |
| 3 | 1 = negative zero |
| 4 | 1 = positive zero |
| 5 | 1 = positive subnormal number |
| 6 | 1 = positive number |
| 7 | 1 = positive infinity |
| 8 | 1 = signalling nan |
| 9 | 1 = quiet nan |
| 10 to 30 | not used |
| 31 | 1 = negative, 0 = positive number |

### FCMP\_EQ - Float Compare for Equality

**Description:**

The register compare instruction compares two registers as floating-point values for equality and sets the result in the target register. Note that negative and positive zero are considered equal. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

If either source operand is a Nan then the result will be false.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 306 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

if Ra == Rb

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Floating Point

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] == Vb[x] ? 1 : 0

else Vmt[x] = Vmt[x]

### FCMP\_EQI - Float Compare for Equality Immediate

**Description:**

The compare instruction compares a register to an immediate value as floating-point values for equality and sets the result in the target register. Note that negative and positive zero are considered equal. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

The immediate form of the instruction will normally be used with a constant postfix instruction.

If either source operand is a Nan then the result will be false.

**Instruction Format: RI**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 306 |

**Operation:**

if Ra == Imm

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Floating Point

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] == Vb[x] ? 1 : 0

else Vmt[x] = Vmt[x]

### FCMP\_LT - Float Compare for Less Than

**Description:**

The register compare instruction compares two registers as floating-point values for les than and sets the result in the target register. Note that negative and positive zero are considered equal. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

If either source operand is a Nan then the result will be false.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 366 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

if Ra < Rb

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Floating Point

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] < Vb[x] ? 1 : 0

else Vmt[x] = Vmt[x]

### FCMP\_NE - Float Compare for Not Equal

**Description:**

The register compare instruction compares two registers as floating-point values for inequality and sets the result in the target register. Note that negative and positive zero are considered equal. For scalar instructions the result of 1 or 0 is placed in a scalar register. For vector operations each bit of the register is set according to the result of the compare for the corresponding vector lane. If the target register is a mask register it may be used subsequently to mask vector operations.

The immediate form of the instruction will normally be used with a constant postfix instruction.

If either source operand is a Nan then the result will be true.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 316 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

if Ra == Rb

Rt= 1

else

Rt= 0

**Instruction Format: RI**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 316 |

**Operation:**

if Ra == Imm

Rt= 1

else

Rt= 0

**Clock Cycles:** 1

**Execution Units:** Floating Point

**Vector Operation:**

for x = 0 to VL - 1

if (Vm[x]) Vmt[x] = Va[x] == Vb[x] ? 1 : 0

else Vmt[x] = Vmt[x]

### FFINITE – Number is Finite

**Description:**

Test the value in Ra to see if it’s a finite number and return Z=1 or Z = 0 in register Rt.

**Integer Instruction Format: F1**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | Tb | 326 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles: 1**

**Execution Units:** Floating Point

**Example**:

finite $a1,$f7

### FMA – Floating Point Multiply Add

**Description:**

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

**Instruction Format: F3**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Rm2 | Tc | Rc6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 446 |

**Operation:**

Rt = Ra \* Rb + Rc

**Clock Cycles: 7**

**Execution Units:** All Floating Point

### FMS – Floating Point Multiply Subtract

**Description:**

Multiply two floating point numbers in registers Ra and Rb subtract a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding.

**Instruction Format: F3**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Rm2 | Tc | Rc6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 456 |

**Operation:**

Rt = Ra \* Rb - Rc

**Clock Cycles: 7**

**Execution Units:** All 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:** R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Rm2 | ~ | 46h6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 02h6 |

**Operation:**

Rt = Ra \* Rb

**Clock Cycles: 7**

**Execution Units:** All Floating Point

### FNABS – Negative Absolute Value

**Description:**

This instruction takes the negative of the absolute value of a register and places the result in a target register. The values are treated as floating-point values. The sign bit of the number is set, no rounding of the number takes place.

**Integer Instruction Format: R1**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | Tb | 436 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Operation:**

If Ra > 0

Rt = -Ra

else

Rt = Ra

**Vector Operation**

for x = 0 to VL - 1

if (Vm[x]) Rt[x] = Ra[x] > 0 ? -Ra[x] : Ra[x]

else Rt[x] = Rt[x]

**Execution Units:** Float

**Clock Cycles: 1**

**Exceptions:** none

**Notes:**

### FNMA – Floating Point Negate Multiply Add

**Description:**

Multiply two floating point numbers in registers Ra and Rb add a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding. The Ra operand is negated before the operation takes place.

**Instruction Format: F3**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Rm2 | Tc | Rc6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 446 |

**Operation:**

Rt = - Ra \* Rb + Rc

**Clock Cycles: 7**

**Execution Units:** All Floating Point

### FNMS – Floating Point Negate Multiply Subtract

**Description:**

Multiply two floating point numbers in registers Ra and Rb subtract a third number from register Rc and place the result into target register Rt. The multiplication and addition are fused with no intermediate rounding. The Ra operand is negated before the operation takes place.

**Instruction Format: F3**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Rm2 | Tc | Rc6 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 456 |

**Operation:**

Rt = - Ra \* Rb - Rc

**Clock Cycles: 7**

**Execution Units:** All Floating Point

### FRES – Reciprocal Estimate

**Description:**

This function uses a 1024 entry 16-bit precision lookup table to create a piece-wise approximation of the reciprocal and linear interpolation to approximate the reciprocal of the value in Ra. The value is returned in Rt as a 32-bit floating-point value. The value returned is accurate to about eight bits.

**Instruction Format: R1**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | Tb | 376 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles: 7**

**Execution Units:** Floating Point

### FRSQRTE – Float Reciprocal Square Root Estimate

**Description:**

Estimate the reciprocal of the square root of the number in register Ra and place the result into target register Rt.

**Instruction Format: R1**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | Tb | 366 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles: 7**

**Execution Units:** Floating Point

**Notes**:

The estimate is only accurate to about 3%.

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

### FSIGMOID – Sigmoid Approximate

**Description:**

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

**Instruction Format: R1**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | Tb | 386 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles: 7**

**Execution Units:** Floating Point

### FSIGN – Sign of Number

**Description:**

This instruction provides the sign of a floating-point number contained in a general-purpose register as a floating-point result. The result is +1.0 if the number is positive, 0.0 if the number is zero, and -1.0 if the number is negative.

**Instruction Format: R1**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | ~ | 466 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles:** 1

**Execution Units:** All Floating Point

**Operation:**

Rt = sign of (Ra)

### FTOI – Float to Integer

**Description:**

This instruction converts a floating-point value to an integer value. Many floating-point values will not fit into an integer. If overflow occurs the integer will be set to the maximum integer value. If a fraction less than one is converted, the result is rounded and will be either one or zero.

**Instruction Format: R1**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | ~ | 416 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles:** 7

**Execution Units:** All Floating Point

### FTRUNC – Truncate Value

**Description**:

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

**Instruction Format**: R1

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 16 | ~ | 476 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles**: 7

**Execution Units:** Floating Point

### ITOF – Integer to Float

**Description:**

This instruction converts an integer value to a floating-point representation.

**Instruction Format: R2**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 406 | ~ | ~6 | Mask3 | Ta | Ra6 | Tt | Rt6 | 26 |

**Clock Cycles:** 7

**Execution Units:** All Floating Point

## Memory Operations

### CACHE – Cache Command

**Description:**

This instruction commands the cache controller to perform an operation. Commands are summarized in the command table below. Commands may be issued to both the instruction and data cache at the same time. The address of the cache line to be invalidated is passed in Ra if needed.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 9 | 8 6 | 5 0 |
| m | Displacement16 | Mask3 | Ta | Ra6 | 0 | DC3 | IC3 | 636 |

**Commands:**

|  |  |  |
| --- | --- | --- |
| IC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 1 | invline | invalidate line associated with given address |
| 2 | invall | invalidate the entire cache (address is ignored) |
| 3 to 7 |  | reserved |

|  |  |  |
| --- | --- | --- |
| DC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 1 | enable | enable cache (instruction cache is always enabled) |
| 2 | disable | not valid for the instruction cache |
| 3 | invline | invalidate line associated with given address |
| 4 | invall | invalidate the entire cache (address is ignored) |
| 5 to 7 |  | reserved |

**Clock Cycles:** 20

**Execution Units:** All ALU’s / Memory

**Operation:**

**Exceptions:** DBG

### CACHEX – Cache Command

**Description:**

This instruction commands the cache controller to perform an operation. Commands are summarized in the command table below. Commands may be issued to both the instruction and data cache at the same time.

**Instruction Format:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 9 | 8 6 | 5 0 |
| m | ~2 | ~ | 636 | Tb | Rb6 | Mask3 | Ta | Ra6 | Tt | DC3 | IC3 | 26 |

**Commands:**

|  |  |  |
| --- | --- | --- |
| IC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 1 | invline | invalidate line associated with given address |
| 2 | invall | invalidate the entire cache (address is ignored) |
| 3 to 7 |  | reserved |

|  |  |  |
| --- | --- | --- |
| DC3 | Mne. | Operation |
| 0 | NOP | no operation |
| 1 | enable | enable cache (instruction cache is always enabled) |
| 2 | disable | not valid for the instruction cache |
| 3 | invline | invalidate line associated with given address |
| 4 | invall | invalidate the entire cache (address is ignored) |
| 5 to 7 |  | reserved |

**Clock Cycles:** 20

**Execution Units:** All ALU’s / Memory

**Operation:**

**Exceptions:** DBG

### LDB – Load Byte

**Description:**

An eight-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra. For vector instructions Ra is a scalar register.

This instruction may load data from the cache and cause a cache load operation if the data isn’t in the cache provided the current memory page is cacheable.

**Instruction Format: RI**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Displacement16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 486 |

**Clock Cycles:** 20 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (memory8[Ra+displacement])

**Exceptions:** DBE, TLB, RDV

### LDT – Load Tetra

**Description:**

A thirty-two-bit value is loaded from memory and sign extended, then placed in the target register. The memory address is the sum of the sign extended offset and register Ra.

This instruction may load data from the cache and cause a cache load operation if the data isn’t in the cache provided the current memory page is cacheable.

Loading a vector register loads all 512-bits. A vector mask register may be used to determine which elements of the vector are updated.

**Instruction Format:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Displacement16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 526 |

**Clock Cycles:** 20 (one memory access)

**Execution Units:** All ALU’s / Memory

**Operation:**

Rt = sign extend (memory32[Ra+displacement])

**Exceptions:** DBE, TLB, RDV

## Branches

### Conditions

Conditional branches branch to the target address only if the condition is true. The condition is determined by the comparison of two general-purpose registers. Two sets of conditional branches are supported. One set for integers and a second set for floating-point values.

### Conditional Branch Format

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| Displacement17 | Cnd3 | 0 | Ra6 | 0 | Rb6 | Opcode6 |

### Branch Conditions

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ▼ |  |  |  |  | ▼ |
| 39 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| Displacement17 | Cnd3 | 0 | Ra6 | 0 | Rb6 | Opcode6 |

|  |  |  |
| --- | --- | --- |
| Cnd3 | Integer Comparison Test | Float |
| 0 | signed less than | less than |
| 1 | signed greater or equal | greater than or equal |
| 2 | unsigned less than | less than or equal |
| 3 | unsigned greater than or equal | greater than |
| 4 | reserved | reserved |
| 5 | reserved | reserved |
| 6 | equal | equal |
| 7 | not equal | not equal |

### BEQ – Branch if Equal

**Description**:

This instruction branches to the target address if the contents of the Ra equals the contents of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: Bcc

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| Displacement17 | 63 | 0 | Ra6 | 0 | Rb6 | 286 |

**Operation:**

If (Ra==Rb)

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### BGE – Branch if Greater than or Equal

**Description**:

This instruction branches to the target address if the contents of the Ra is greater than or equal to the contents of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: Bcc

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| Displacement17 | 13 | 0 | Ra6 | 0 | Rb6 | 286 |

**Operation:**

If (Ra >= Rb)

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### BLT – Branch if Less Than

**Description**:

This instruction branches to the target address if the contents of the Ra is less than the contents of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: Bcc

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| Displacement17 | 03 | 0 | Ra6 | 0 | Rb6 | 286 |

**Operation:**

If (Ra < Rb)

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### BNE – Branch if Not Equal

**Description**:

This instruction branches to the target address if the contents of the Ra does not equal the contents of Rb, otherwise program execution continues with the next instruction. The values are treated as signed integers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: Bcc

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| Displacement17 | 73 | 0 | Ra6 | 0 | Rb6 | 286 |

**Operation:**

If (Ra<>Rb)

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### FBEQ – Float Branch if Equal

**Description**:

This instruction branches to the target address if the contents of the Ra equals the contents of Rb, otherwise program execution continues with the next instruction. The values are treated as single precision floating-point numbers. For a further description see [Branch Instructions](#_Branch_Instructions).

**Formats Supported**: Bcc

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 39 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| Displacement17 | 03 | 0 | Ra6 | 0 | Rb6 | 296 |

**Operation:**

If (Ra==Rb)

IP = IP + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

For a floating-point comparison positive and negative zero are considered equal.

### JMP – Jump to Address

**Description**:

This instruction always jumps to the target address. The target address range is 4GB.

**Formats Supported**: JMP

|  |  |  |
| --- | --- | --- |
| 39 8 | 7 6 | 5 0 |
| Target32 | 02 | 246 |

**Operation:**

IP = Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### JSR – Jump to Subroutine

**Description**:

This instruction always jumps to the target address. The address of the next instruction is stored in a link register. The target address range is 4GB.

**Formats Supported**: JMP, JMPR

|  |  |  |
| --- | --- | --- |
| 39 8 | 7 6 | 5 0 |
| Target32 | Rt2 | 246 |

**Operation:**

Lk = next IP

IP = Constant

**Format**: JMPR

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 266 |

**Operation:**

Rt = next IP

IP = Ra + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

### RTS – Return from Subroutine

**Description**:

This instruction always jumps to the target address. The address of the next instruction is stored in a link register. The target address range is 4GB. A return is achieved by specifying one of the return address registers for Ra. RTS mnemonic assumes Rt is zero and Ra is 41.

**Format**: JMPR

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | Immediate16 | Mask3 | Ta | Ra6 | Tt | Rt6 | 266 |

**Operation:**

Rt = next IP

IP = Ra + Constant

**Execution Units**: Branch

**Exceptions**: none

**Notes:**

## System Instructions

### CSRx – Control and Special / Status Access

**Description**:

The CSR instruction group provides access to control and special or status registers in the core. For the read operation the current value of the CSR is placed in the target register Rt.

**Instruction Format**: CSR

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 38 37 | 36 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | O2 | Immediate14 | mask3 | Ta | Ra6 | Tt | Rt6 | 76 |

|  |  |  |
| --- | --- | --- |
| O3 |  | Operation |
| 0 | CSRRD | Only read the CSR, no update takes place, Ra should be x0. |
| 1 | CSRRW | Read/Write to CSR |
| 2 | CSRRS | Read/Set CSR bits |
| 3 | CSRRC | Read/Clear CSR bits |

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

The Regno[13..12] field is reserved to specify the operating mode. Note that registers cannot be accessed by a lower operating mode.

**Clock Cycles**: 1

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

### REX – Redirect Exception

**Description**:

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

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

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

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

The privilege level is set to the value in Ra.

**Integer Instruction Format: REX**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 8 | 7 6 | 5 0 |
| m | ~2 | ~ | 16 | Tb | 266 | Mask3 | Ta | Ra6 | 0 | Irq4 | Tm2 | 26 |

|  |  |
| --- | --- |
| Tm2 |  |
| 0 | redirect to user mode |
| 1 | redirect to supervisor mode |
| 2 | redirect to hypervisor mode |
| 3 | reserved |

|  |  |
| --- | --- |
| Irq4 |  |
| 0xxx | Do not set IRQ mask |
| 1nnn | Set Irq mask to nnn |

**Clock Cycles**: 4

Execution Units: Branch

Example:

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

**Notes**:

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

### RTI – Return from Interrupt or Exception

**Description**:

Restore the previous interrupt enable setting, operating mode, and privilege level and transfer program execution back to the address in the exception address register, vector register #63.

This is a special form of the VSRLVI instruction.

**Instruction Format:** R1

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | ~2 | ~ | 336 | Tb | 16 | Mask3 | 1 | 636 | 1 | 636 | 26 |

**Flags Affected**: none

**Operation:**

PMSTACK = PMSTACK >> 8

PLSTACK = PLSTACK >> 8

IP = v63[0]

**Execution Units**: Branch

**Clock Cycles**:

**Exceptions**: none

**Notes**:

### TLBRD – Read TLB

**Description**:

This instruction reads the TLB. Which translation entry to read comes from the value in Ra. The read value is transferred to vector register Rt. A description of the PTE and PMTE formats is in the section of the text describing the TLB.

The entry number for Ra comes from virtual address bits 14 to 23. The low order bits of the Ra value determine which way to update in the TLB if the algorithm is a fixed or LRU way algorithm. Otherwise, a way to update will be selected randomly. When the LRU algorithm is active the most recently used entry is placed in way #0. It may be desirable to bump out entry #3 and replace it with the new entry for LRU operation.

Page numbers are in terms of a 16kB page size.

**Instruction Format**: R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | 02 | ~ | 76 | 0 | ~6 | Mask3 | 0 | Ra6 | 1 | Rt6 | 26 |

Op2: 00=read TLB,01=write TLB

Ra Value:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 16 | 15 | 14 5 | 43 | 2 0 |
|  | 0 | Entry Num10 | ~2 | way3 |

Rt Value: - unused fields should be zero

|  |  |
| --- | --- |
| Element | 31 0 |
| 0 | PTE31..0 |
| 1 | PTE63..32 |
| 2 | PTE95..64 |
| 3 | PTE127..96 |
| 4 | PMTE31..0 |
| 5 | PMTE63..32 |
| 6 | PMTE95..64 |
| 7 | PMTE127..96 |
| 8 | PTE Address31..0 |
| 9 | PTE Address63..32 |
| 10 | PMTE Address31..0 |
| 11 | PMTE Address63..32 |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |

**Exceptions:** none

### TLBRW – Read / Write TLB

**Description**:

This instruction both reads and writes the TLB. Vector registers are used to transfer data to or from the TLB. Which translation entry to update comes from the value in Ra. The update value comes from the value in Rb. Rb contains the PTE and PMTE and addresses of the entries. The current value of the entry selected by Ra is copied to Rt. A description of the PTE and PMTE formats is in the section of the text describing the TLB.

The entry number for Ra comes from virtual address bits 14 to 23. The low order bits of the Ra value determine which way to update in the TLB if the algorithm is a fixed or LRU way algorithm. Otherwise, a way to update will be selected randomly. When the LRU algorithm is active the most recently used entry is placed in way #0. It may be desirable to bump out entry #3 and replace it with the new entry for LRU operation.

Page numbers are in terms of a 16kB page size.

**Instruction Format**: R2

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 39 | 3837 | 36 | 25 30 | 29 | 28 23 | 22 20 | 19 | 18 13 | 12 | 11 6 | 5 0 |
| m | 12 | ~ | 76 | 0 | Rb6 | Mask3 | 0 | Ra6 | 1 | Rt6 | 26 |

Op2: 00=read TLB,01=write TLB

Ra Value:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 16 | 15 | 14 5 | 43 | 2 0 |
|  | 0 | Entry Num10 | ~2 | way3 |

Rb Value: - unused fields should be zero

|  |  |
| --- | --- |
| Element | 31 0 |
| 0 | PTE31..0 |
| 1 | PTE63..32 |
| 2 | PTE95..64 |
| 3 | PTE127..96 |
| 4 | PMT31..0 |
| 5 | PMT63..32 |
| 6 | PMT95..64 |
| 7 | PMT127..96 |
| 8 | PTE Address31..0 |
| 9 | PTE Address63..32 |
| 10 | PMT Address31..0 |
| 11 | PMT Address63..32 |
| 12 |  |
| 13 |  |
| 14 |  |
| 15 |  |

**Exceptions:** none