FT64v7

Preface

Who This Book is For

This book is for the FPGA enthusiast who's interested in instruction set architecture (ISA). It's advisable that one have a fairly good background in digital electronics, computer systems and networking before attempting a read. Examples are provided in the Verilog language, it would be helpful to have some understanding of HDL languages. Finally, a lot about computer architecture is contained within these pages, some previous knowledge would also be helpful. If you're into electronics and computers as a hobby FPGA's can be a lot of fun.

Motivation

The author has learned a few new tricks and it was time for a new book. Initially the author was going to create one massive book covering many different cores as an extension of the Table888 book. Rather than create a massive revision of the book, a new book has been created that focuses on a single component. This book is another book about the development of a 64-bit homebrew processor, more specifically it's ISA. This time the processing core is much more in line with contemporary processors and has an improved ISA. Once again one has to be somewhat nuts to consider it. It takes a lot of time and dedication to develop a good ISA; perhaps too much for a hobbyist. It is less work to develop cores against and already established ISA.

If you seek to be an expert on the personal computer or laptop sitting on your desk, there's nothing like trying to develop your own system to learn things. It's possible these days to develop something simple and rudimentary using a small FPGA board available from several different vendors. One can get started working with FPGA's for well under $100; with free toolsets available it's not an expensive hobby. It's no more expensive than a good video game and can provide a lot of entertainment for the money. For an outlay of a few hundred dollars one can begin to become a real expert on home-grown processors, including some of the more advanced aspects of processor such as memory management and data protection. FPGA stands for 'Field Programmable Gate Array', which is a chip with lots of small memories interconnected with a connection network. I’m currently using the Nexys4Video board from Digilent. I’ve upgraded several times, to more memory and more logic cells. I’ve used boards from Terasic and BurchEd in the past. Of course, it’s also possible to make your own board if you have the skills. The first board I used was one I wired up myself but it didn’t work very reliably. Be sure to recycle the boards appropriately; I sell my older boards on Ebay to budding students.

The ISA and core presented here aren’t necessarily the best available for a given system. The processor isn’t the smallest or fastest RISC processor. The core presented here is also not a simple beginner’s example. Those weren’t my goals. Instead they offer reasonable size and performance and hopefully are balanced to the available hardware.

There’s lots of room for expansion in the future. I chose 64 bits in part anticipating more than 4GB of memory available sometime down the road. A 64-bit architecture is doable in FPGA’s today, although it uses double or more the resources that a 32 bit design would.

What this book is about

This book is an outgrowth of the Table888 book and covers one aspect of core design – a general purpose ISA for a 64-bit machine. It is centric to the author and a record of his work. The book shows how flexible FPGA’s can be. A single FPGA board is used for several different systems. The processing core could be incorporated into a single chip (called an SoC System-On-Chip). Although the book is primarily about the ISA some notes on an implementation of FT64 are included. The author continues to learn about computing systems.

About the Author

First a warning: I’m an enthusiastic hobbyist like yourself, with a ton of experience. I’ve spent a lot of time at home doing research and implementing several soft-core processors, almost maniacally. One of the first cores I worked on was a 6502 emulation. I then went on to develop the Butterfly32 core. Later the Raptor64. I have about 20 years professional experience working on banking applications at a variety of language levels including assembler. So I have some real world experience developing complex applications. I also have a diploma (in what is now a degree course) in electronics engineering technology. Some of the cores I work on these days are really too complex and too large to do at home on an inexpensive FPGA. I await bigger, better, faster boards yet to come.

Status

FT64 is still a project in the works. The ISA has become somewhat stable and is unlikely to undergo major changes. The basic instruction formats are unlikely to change. A lot of work has been put into implementing the ISA as a two-way superscalar core, but it is far from a finished project. The author has begun work on a new version of the processor so maybe it’s time to spit out the book for the current version. The most recent evolution of FT64 (FT64v3) is as a barrel processor.

Overview

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

- 64 register sets
- 32 general purpose scalar registers
- 32 general purpose floating-point registers
- 32 general purpose vector registers, length 63
- register renaming
- speculative loading
- 32-bit fixed instruction format
- 64-bit data width
- powerful branch prediction with target buffer (BTB)
- return address prediction (RSB)
- bus interface unit
- instruction and data caches
- Vector and SIMD operations
- fine-grained simultaneous multi-threading (SMT)
- dual ALU's, one flow control unit,
- one memory unit, one to three read channels
- optional write buffering
- zero to two floating point units
- bus randomization on missed speculative execution
- ability to disable branch prediction and the data cache
- wide variety of memory management options:
- 1) none
- 2) {reserved}
- 3) paging
- 4) software managed tlb
- 5) inverted page table

History

FT64v7 is a work-in-progress beginning in October 2017. FT64 originated from RiSC-16 by Dr. Bruce Jacob. RiSC-16 evolved from the Little Computer (LC-896) developed by Peter Chen at the University of Michigan. See the comment in FT64.v. FT64 is the author's fifth attempt at a 64 bit ISA. Other attempts including Raptor64, Thor, FISA64, and DSD9. The author has tried to be innovative with this design borrowing ideas from a number of other processing cores. Berkeley's RiSC-V has had an influence on this core.

Goals

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

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

Easy implementation of a compiler.

Eventual implementation as a four-way superscalar processing core.

Core Features

SMT

The core is capable of fine-grained SMT (simultaneous multi-threading) operation. With SMT there are two possible threads of execution each of which operates at about ½ the performance of a single thread. For some applications it may be desirable to use SMT in order to increase the overall performance of the system. The core fetches from two different execution threads simultaneously. When enabled the core’s program counters operate independently. One half of the fetch buffers are used for each of two possible threads of execution.

The availability of SMT operation does not affect the core’s ISA. The same instructions are executed on either thread, there are no instructions specific to SMT.

SMT Granularity

There are two basic granularities to SMT, fine-grained and coarse-grained. This core is capable of fine-grained SMT meaning instructions for more than one thread are fetched and executed per clock cycle. In coarse-grained SMT the same thread may run for several clock cycles before the thread alternates to another thread.

Notes:

For simplicity, on a branch miss the entire fetch buffer is flushed and reloaded with instructions from the target address. This includes instructions for both threads of execution. Both threads may miss at the same time and the fetch buffer will only be reloaded once.

External interrupts are processed only by the even numbered thread to prevent an interrupt from being processed twice. Other exceptions may occur on either thread.

There is a bit in the machine status register to indicate which thread is running when the bit is checked. Testing this bit makes it possible to branch to different code for each thread.

When SMT is turned on, the program counter for the second thread will be pointing to the next instruction after the current program counter. Also, the current program counter may increment by eight for one cycle. Because the exact value of the program counters may not be known a ramp of harmless instructions needs to be performed when SMT is turned on. the following code show an example of turning on SMT.

```
        ldi      r1,#$10000          ; turn on SMT use $10000
        csrrs    r0,#0,r1
        add      r0,r0,#0       ; fetch adjustment ramp
        add      r0,r0,#0
        add      r0,r0,#0
        add      r0,r0,#0
        add      r0,r0,#0
        add      r0,r0,#0
        add      r0,r0,#0
        add      r0,r0,#0
        add      r0,r0,#0
        add         r0,r0,#0
        csrrd       r1,#$044,r0          ; which thread is running ?
        bfextu      r1,r1,#24,#24
        bne         r1,r0,.st2
```

Turning SMT off is equally as tricky. It is best to turn off SMT when the location of both sets of program counters are known. Turning SMT off will cause the second program counter to follow the first one again.

Vectors

The core has standard support for vector operations. Support includes vector length and vector mask registers. Although the vector registers are currently limited to 63 elements there is no reason why the number of vector elements can't be increased in the future.

Vector Chaining

The core is capable of limited chaining of vector instruction; starting a second vector instruction before the first one is complete.

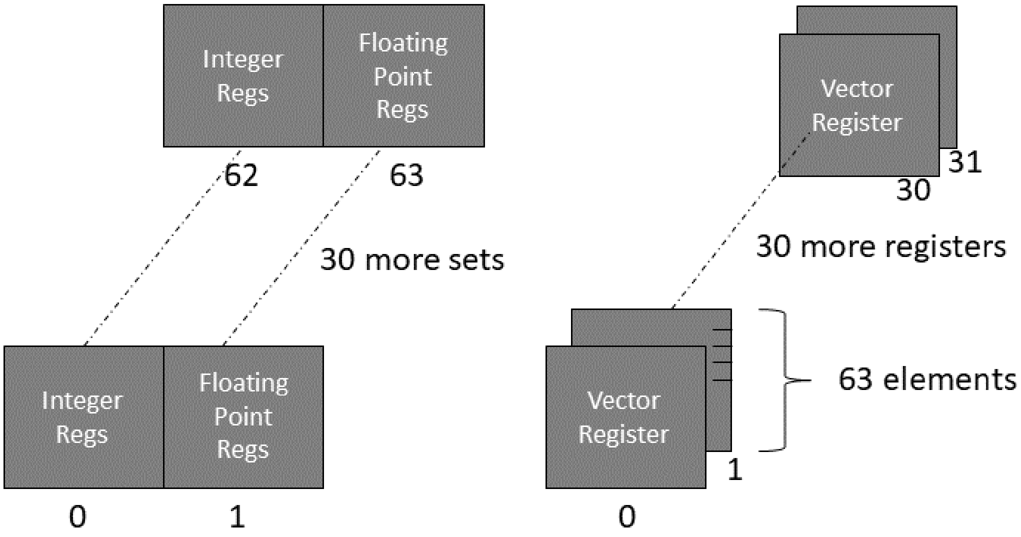
The vector chain bit in control register #0 controls the priority of queueing vector instructions when there are two vector instructions available to queue. If vector chaining is on, then one element from each vector instruction will queue. If vector chaining is off, then two elements from the first vector instruction will queue. Vector chaining may improve performance depending on the instruction mix. For instance, if there is a multiply followed by an add under normal circumstances multiplication of the next vector element can't proceed until the instruction is finished. Without vector chaining the add can't proceed until the multiply is done. With vector chaining the add can be performed at the same time as the multiply, hiding some of the latency of the multiply operation.

Pieces

This section covers various pieces that make up the FT64 core and influence it’s ISA.

Register File

The processor caches 64 sets of 32 registers in the architecture. A set may be selected for use as either general purpose registers or floating-point registers. It is recommended that odd numbered register sets be used for floating point values, while an even numbered register set is used for general purpose registers. This is because the bypass logic in the core only detects two bits of the register set selection. There is a single set of vector registers. The following is an illustration of register set usage.



On reset register set #0 is selected to be the operating register set. Register sets beginning with #1 to #7 is reserved for interrupt processing. The register set is automatically switched on interrupt and restored at interrupt return.

Machine State	Register Set Reservation
BRK / RESET	0
IRQ 1	1
IRQ 2	2
IRQ 3	3
IRQ 4	4
IRQ 5	5
IRQ 6	6
IRQ 7	7
Normal Operations	according to rs field in control reg #0

Wired Register Sets

Register sets are associated with threads of execution. There is a cache of 64 register sets including caching of base and bounds registers. The register sets may be “wired” meaning they are always present in the processor. Non-wired threads may have their associated register sets evicted from the core during a thread switch.

The first eight register sets are always wired. By default, the next eight register sets are also wired. Which registers sets are wired is controlled by the wired thread register (WTR). The number of wired registers always starts with register set zero and proceeds towards larger numbered register sets.

Register Usage Convention

The register usage convention probably has more to do with software than hardware. Excepting a couple of special cases, the registers are general purpose in nature.

R0 always has the value zero in all register sets. r29 is the link register used implicitly by the call instruction.

Register	Description / Suggested Usage	Saver
r0	always reads as zero	
r1-r2	return values / exception	caller
r3-r9	temporaries	caller
r10	exception SP offset	
r11-r17	register variables	callee
r18-r22	function arguments	caller
r23	assembler usage	
r24	type number / function argument	caller
r25	class pointer / function argument	caller
r26	thread pointer	callee
r27	global pointer	
r28	exception link register	callee
r29	return address / link register	callee
r30	base / frame pointer	callee
r31	stack pointer (hardware)	callee

The ISA supports up to 32 vector registers of length 63. There is only a single set of vector registers.

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

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

Notes:

The register set is implemented with block ram resources in the FPGA. It was desired to have 64 element vector registers, since the general register file is also contained in the same block rams, this resulted in 64 available register sets.

The register set currently selected is determined by the rs field in the machine status register (0x044). Note that the register sets selected for SMT operation should have different bit settings for bit 6, and 7 of the register code in order to allow bypassing logic to work correctly.

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

11	6	5	4	0
Register Set	Fp	0	General Purpose Register number	

Vector element	1	Vector register number

To conserve hardware which would otherwise be quite large, the bypassing logic looks at only the six least significant bits, plus bits 6, and 7 of the register code for bypassing purposes. This allows it to differentiate between different general-purpose registers, floating-point, thread 0 and thread 1 registers, and vector registers. This meets bypass logic requirements in most circumstances.

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

Similarly, the core does not provide bypassing between register sets of the general-purpose registers outside of checking thread register pairs. Switching the register set should be followed by a synchronization operation to ensure contents of the previous instructions are updated before the new use. Note that the break and return from interrupt operations automatically synchronize the processor so that register sets remain valid.

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

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

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

On reset register set #0 is selected.

Program Counter

The program counter identifies which instruction to execute. The program counter normally increments by the size of the instruction (2, 4, or 6). The increment may be overridden using one of the flow control instructions. The program counter usually addresses 16-bit instruction parcels. The program counter register is also split into two sections. Branch instructions only update the lower 32-bits.

63 32	31 0
PC High$_{32}$	PC Low$_{[39..0]}$

There are actually multiple program counters in use by the core, one for each thread.

Register Zero

Register zero – r0 – always reads as zero.

Stack and Frame Pointers

Although the stack and frame pointer registers may be used with any instruction the core has special hardware to detect stack bounds violations by either the stack pointer or frame pointer. The stack and frame pointer registers should be kept aligned on whole word boundaries. That is, they should be a multiple of eight, which has the three least significant bits as zero. There is currently no hardware in the core to enforce a word alignment.

Control and Status Registers

Control Register Zero (CSR #000)

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

Bit		Description
0	Pe	Protected Mode Enable: 1 = enabled, 0 = disabled
8 to 13	cmpgrp	compressed instruction group (0-7 valid)
16	SMT	simultaneous multi-threading enable 1 = enabled, 0 = disabled (0 default).
30	DCE	data cache enable: 1=enabled, 0 = disabled
32	BPE	branch predictor enable: 1=enabled, 0=disabled
34	WBM	write buffer merging enable: 1 = enabled, 0 = disabled
35	SPLE	speculative load enable (1 = enable, 0 = disable) (0 default)
36		
63	D	debug mode status. this bit is set during an interrupt routine if the processor was in debug mode when the interrupt occurred.

This register supports bit set / clear CSR instructions.

DCE

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

BPE

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

WBM bit

Merging of values stored to memory may be disabled by setting this bit. On reset write buffer merging is disabled because it is likely desirable to setup I/O devices. Many I/O devices require updates to individual bytes by separate store instructions.

SPLE

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

HARTID (0x001)

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

TICK (0x002)

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

PCR Paging Control (CSR 0x003)

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

AEC Arithmetic Exception Control (CSR 0x004)

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

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

PMR Power Management Register (CSR 0x005)

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

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

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

CAUSE (0x006)

This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code in order to determine what to do. Only the low order 16 bits are implemented. The high order bits read as zero and are not updateable. The low order eight bits are loaded from the cause field of the BRK instruction. The next eight bits are loaded from the user$_6$ field of the break instruction. Bit 7 of the cause is set if a hardware interrupt was the source of the break.

BADADDR (CSR 0x007)

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

PCR2 Paging Control (CSR 0x008)

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

Scratch (CSR 0x009)

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

WBRCD (CSR 0x00A)

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

BAD_INSTR (CSR 0x00B)

This register contains a copy of the exceptioned instruction.

SEMA (CSR 0x00C) Semaphores

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

Semaphore	Usage Convention
0	LWR / SWC status bit
1	system garbage collection protector
2	system
3	input / output focus list
4	keyboard
5	system busy
6	memory management

VM_SEMA (CSR 0x00D) Semaphores

This register is available for system semaphore or flag use.

KEYS – (CSR 0x00E)

This register contains the collection of keys associated with the process for the memory lot system. Each key is ten bits in size. The register contains six keys.

63 60	59 50	49 40	39 30	29 20	19 10	9 0
$\sim_4$	key6	key5	key4	key3	key2	key1

TCB (CSR 0x010)

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

WRS (CSR 0x011)

This register indicates which register sets are wired, or perpetually present in the core. Register sets beginning with register set #0 and progressing to the value supplied in this register are wired.

There are only 64 register sets in the machine. Wiring too many register sets may make it difficult for the OS to switch threads.

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

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

Bit		Symbol	Description
63:44	~		reserved
43:40	**PRC**		Default precision
39:32	**RGS**		Register set
31:29	**RM**	rm	rounding mode
28	**E5**	inexe	- inexact exception enable
27	**E4**	dbzxe	- divide by zero exception enable
26	**E3**	underxe	- underflow exception enable
25	**E2**	overxe	- overflow exception enable
24	**E1**	invopxe	- invalid operation exception enable
23	**NS**	ns	- non standard floating point indicator
Result Status			
22		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
21	**RA**	rawayz	rounded away from zero (fraction incremented)
20	**SC**	C	denormalized, negative zero, or quiet NaN
19	**SL**	neg <	the result is negative (and not zero)
18	**SG**	pos >	the result is positive (and not zero)
17	**SE**	zero =	the result is zero (negative or positive)
16	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
15	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
14	**X5**	inerx	- inexact result exception occurred (sticky)
13	**X4**	dbzx	- divide by zero exception occurred
12	**X3**	underx	- underflow exception occurred
11	**X2**	overx	- overflow exception occurred
10	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
9	**GX**	gx	- global exception indicator – set if any enabled exception has happened
8	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software
Exception Type Resolution			
7	**X1T**	cvt	- attempt to convert NaN or too large to integer
6	**X1T**	sqrtx	- square root of non-zero negative
5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

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

These registers contain addresses of instruction or data breakpoints.

63	0
Address $_{63..0}$	

DBCR (CSR 0x01C) Debug Control Register

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

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

DBSR (CSR 0x01D) - Debug Status Register

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

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

CAS (CSR 0x02C) Compare and Swap

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

63	0
Value $_{63..0}$	

TVEC (0x030 to 0x033)

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

IM_STACK (0x040)

This register contains the interrupt mask stack. When an exception or interrupt occurs, this register is shifted to the left and the low order bits are set to all ones, when an RTI instruction is executed this register is shifted to the right and the status bits copied from the low order bits of the register. On RTI the last stack entry is set to fifteen masking all interrupts on stack underflow. The low order four bits represent the current interrupt mask level. Only the low order 32 bits of the register are implemented.

OL_STACK (0x041)

This register contains the operating and data level stack. When an exception or interrupt occurs, this register is shifted to the left and the current status copied to the low order bits, when an RTI instruction is executed this register is shifted to the right and the status bits copied from the low order bits of the register. On RTI the last stack entry is set to zero which will select the machine operating level on stack underflow. The low order 16 bits are the code/stack operating level, bits 32 to 47 are the data operating level. Note there is extra unused space in this register to accommodate a change to more threads or greater stack depth. Bit 0,1 represent the current operating level. Bits 32,33 represent the current data level.

PL_STACK (0x042)

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

RS_STACK (0x043)

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

STATUS (0x044)

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

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

VM$_5$

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

VE_HOLD (0x045)

This register contains the currently executing vector element number for fetch buffers #0 and #1. Source and target element numbers are stored independently. Normally the source and target elements are the same, however they may be different if a vector compress instruction is executing. If the vector register set is switched during exception processing this register should be saved and restored.

63 54	53 48	47 38	37 32	31 22	21 16	15 6	5 0
~	vet1	~	ves1	~	vet0	~	ves0

BRS_STACK (0x046)

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

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

EPC (0x048 to 0x4F)

This sets of registers contains the interrupt or exception stack of the program counter register. The top of the stack is register 0x48. When an interrupt or exception occurs register 0x48 to 0x4E are copied to the next register and the program counter is placed into register 0x48. When an RTI instruction is executed the program counter is loaded from register 0x048 and registers 0x048 to 0x047 are loaded with the next register. Register 0x04F is loaded with the address of the break handler so that in the event of an underflow the break handler will be executed.

CODEBUF (0x080 to 0x0BF)

This register range is for access to 64 adaptable code buffers. The code buffers are used by the EXEC instruction to execute code which may change at run-time.

IQ CTR (0x3C0)

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

BM_CTR (0x3C1)

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

IRQ_CTR (0x3C3)

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

BR_CTR (0x3C4)

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

TIME (0x3E0)

The TIME register corresponds to the wall clock real time. This register can be used to compute the current time based on a known reference point. The register value will typically be a fixed number of seconds offset from the real wall clock time. The lower 32 bits of the register are driven by the tm_clk_i clock time base input which is independent of the cpu clock. The tm_clk_i input is a fixed frequency used for timing that cannot be less than 10MHz. The low order 32 bits represent the fraction of one second. The upper 32 bits represent seconds passed. For example, if the tm_clk_i frequency is 100MHz the low order 32 bits should count from 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 of the register is incremented. The upper 32 bits of the register represent the number of seconds passed since an arbitrary point in the past.

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

INSTRET (0x3E1)

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

INFO (0x3F0 to 0x3FF)

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

Caches

Overview

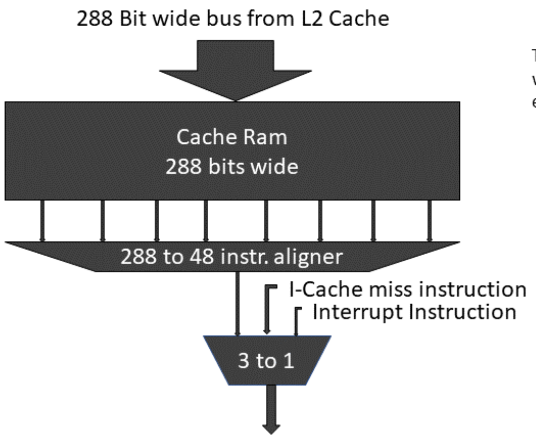
The core has both instruction and data caches to improve performance. The instruction cache is a two-level cache (L1, L2) allowing better performance. The first level cache is four-way set associative, the second level cache is also four-way set associative. The author initially had the first level cache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

Instructions

Since the instruction format affects the cache design it is mentioned here. For this design instructions are of three different sizes (16, 32 or 48 bits). Specific formats are listed under the instruction set description section of this book. Because instructions vary in size the number of instructions fitting onto a cache line may not work out evenly. For this reason, there is an overflow area of 32-bits for each cache line. The overflow area stores the instruction bits remaining from the next cache line.

L1 Instruction Cache

L1 is 2.25kB in size and made from distributed ram to get single cycle read performance. L1 is organized as 64 lines of 36-bytes. Note that there is a separate copy of the L1 cache for each way parallel of the design. Separate copies are used to support SMT in addition to a wider instruction fetch. The following illustration shows the L1 cache organization for FT64.



There are multiple copies of the L1 cache corresponding to the number of ways parallel of the core. Each copy may operate independently if SMT is enabled or in unison with other copies for a wider instruction fetch.

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

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

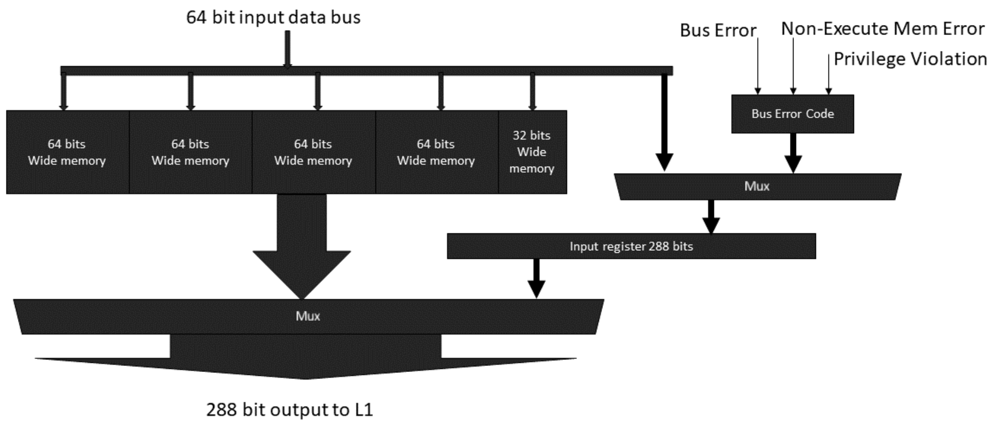
While the L1 cache has single cycle reads it requires two clock cycles to update (write) the cache. The cache line to update needs to be provided by the tag memory which is unknown until after the tag updates.

L2 Instruction Cache

L2 is 18kB in size implemented with block ram. L2 is organized as 512 lines of 36 bytes. Unlike the L1 cache there is only a single L2 cache. There is more flexibility in the design of the L2 cache since it’s made up of block ram components. Once again, the cache is too small in the author’s opinion, but it represents a trade-off in use of block ram resources for the cpu core versus using the block ram for other purposes such as memory management or data cache. There are only so many block rams in the FPGA.

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

L2 Cache Organization



A cache load reads five words, four words for the cache line and a fifth word which is the first word of the next cache line, and stores four and a half words in the cache. The extra half-word is to accommodate 32 and 48 bit instructions that don’t fit evenly into 256 bits and would overflow a 256 bit cache line. Cache line size is 36 bytes.

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

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

The L2 cache is 18kB (512 rows * 36 bytes) composed of block ram which has a three cycle read latency. The L2 cache is four-way set associative.

The L2 cache reads five words from memory on a cache line load, the fifth word read is the first word of the next cache line which is also stored in the current cache line. The reason to do this is instructions may not fit evenly into a 32-byte cache line so there is a four-byte overflow area. While data is loading into the L2 cache an input register is also loaded with the data, the input register is transferred to the L1 cache, this is done so that the L1 cache update doesn’t have to wait for the three cycle read latency of the L2 cache. Also fed into the input register are instructions for error processing should an error occur during the cache load.

Data Cache

The data cache organization is somewhat simpler than that of the instruction cache. Data is cached with a single level cache because it's not critical that the data be available within a single clock cycle at least not for the hobby design. Some of the latency of the data cache can be hidden by the presence of non-memory operating instructions in the instruction queue and the ability of the core to overlap data fetches.

The data cache is organized as 256 lines of 64 bytes (16kB) and implemented with block ram. Access to the data cache is multicycle. The data cache has three read ports allowing three load operations to be in progress at the same time. The policy for stores is write-through. Stores always write through to memory. Since stores follow a write-through policy the latency of the store operation depends on the external memory system. It isn't critical that the cache be able to update in single cycle as external memory access is bound to take many more cycles than a cache update. There is only a single write port on the data cache.

Cache Enables

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

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

Cache Validation

A cache line is automatically marked as valid when loaded. The entire cache may be invalidated using the CACHE instruction. Invalidating a single line of the cache is not currently supported, but it is supported by the ISA.

Uncached Data Area

The address range $F…FDxxxxx is an uncached 1MB data area. This area is reserved for I/O devices. The data cache may also be disabled in control register zero. There is also a set of load instructions that bypass the data cache. These are called load volatile (LVx) instructions.

Write Buffering

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

Fetch Buffers

There are two fetch buffers each of which holds a pair of instructions. When a fetch buffer becomes empty it is loaded with new instructions from the cache. While the processor is working with instructions from one fetch buffer, the other fetch buffer can be loading more instructions. In the case of a cache miss or interrupt a special instruction is loaded into the fetch buffer rather than the instruction output by the cache. For a cache miss this is the NOP instruction. For an interrupt this is the BRK instruction. The program counter increment is suppressed during a cache miss.

The program counters are located in the fetch buffer component.

When SMT is enabled one half of the fetch buffers is used for each thread.

Fetch Rate

The fetch rate is two instructions per clock cycle. When SMT is on one instruction is fetched for each thread. This is fine-grained SMT.

Return Address Stack Predictor (RSB)

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

Branch Predictor

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

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

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

Sequence Numbers

Sequence numbers are mentioned here because they are used by branch instructions to determine what to invalidate.

The core assigns a sequence number to each instruction as it enters the instruction queue. The purpose of the sequence number is to allow the core to determine which instructions should be invalidated because of a branch miss. All the instructions in the queue with a sequence number coming after the branch instruction’s sequence number will be invalidated. The sequence number assigned is the next highest number above that which is already in the queue. As instructions commit to the machine state, the sequence numbers of following instructions are decremented by the value of the sequence number of the committing instruction. This keeps the sequence numbers within range of the number of queue entries while maintaining the ordering relationship between the numbers.
The sequence number mechanism allows the core to speculate across any number of branches that might be in the instruction queue.

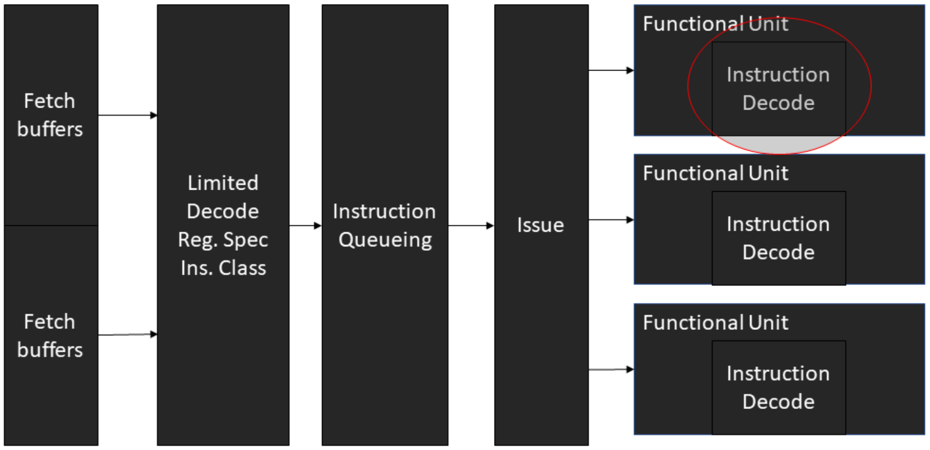
Branch Target Buffer (BTB)

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

Decode Logic

Instruction decode is performed primarily in two places a) in the instruction decode and register fetch stage of the core, and b) within the core's functional units. Broad classes of instructions are decoded for the benefit of issue logic along with register specifications prior to instruction enqueue. Most of the decodes are done with functions defined early in FT64.v because decoding typically involves reducing a wide input into a smaller number of output signals. Other decodes are done at instruction execution time with case statements.

Placement of Instruction Decode



Limited decode takes place between fetch and queue. Between fetch and queue register specifications are decoded along with general instruction classes for the benefit of issue. A handful of additional signals (like sync) that control the overall operation of the core are also decoded. Much of the instruction decode is actually done in the functional unit. The instruction register is passed right through to the functional units in the core.

A sample decode of the register Rc field is shown below. It shows that there is additional logic required to insert the register set selection and vector element selection into the final register select output bits. It also shows that the least significant bits of the register Rc field of the instruction are simply copied directly from the instruction.

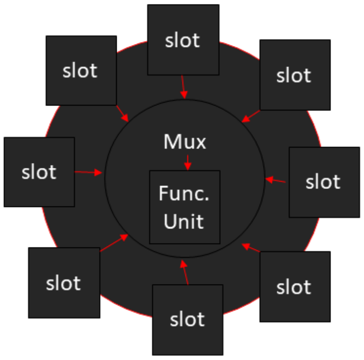
```
function [RBIT:0] fnRc;
input [31:0] isn;
input [5:0] vqei;
input thrd;
case(isn[`INSTRUCTION_OP])
`RR:      case(isn[`INSTRUCTION_S2])
        `SVX:     fnRc = {vqei,1'b1,isn[`INSTRUCTION_RC]};
                `SBX,`SCX,`SHX,`SWX,`SWCX,`CACHEX:
                    fnRc = {rgs,1'b0,isn[`INSTRUCTION_RC]};
                `CMOVEQ,`CMOVNE,`MAJ:
                    fnRc = {rgs,1'b0,isn[`INSTRUCTION_RC]};
        default:   fnRc = {rgs,1'b0,isn[`INSTRUCTION_RC]};
        endcase
`VECTOR:
                            case(isn[`INSTRUCTION_S2])
        `VSxx,`VSxxS,`VSxxU,`VSxxSU:   fnRc = {6'h3F,1'b1,2'b0,isn[18:16]};
        default:   fnRc = {vqei,1'b1,isn[`INSTRUCTION_RC]};
        endcase
```

```
`FLOAT:           fnRc = {rgs[5:1],1'b1,1'b0,isn[`INSTRUCTION_RC]};
`BccR:            fnRc = {rgs,1'b0,isn[`INSTRUCTION_RC]};
default:    fnRc = {rgs,1'b0,isn[`INSTRUCTION_RC]};
endcase
endfunction
```

Instruction Queue (ROB)

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

Instruction Queue – Re-order Buffer



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

Queue Rate

Up to two instructions may queue during the same clock cycle depending on the availability of queue slots. For a vector instruction up to two elements of the vector may queue during a clock cycle, or if vector chaining is on single elements from both the current and following vector instruction may queue during the same clock cycle.

Sequence Numbers

The queue maintains a 32-bit instruction sequence number which gives other operations in the core a clue as to the order of instructions. The sequence number is assigned when an instruction queues. Branch instructions need to know when the next instruction has queued in order to detect branch misses. A separate sequence number is maintained for each hardware thread. The program counter cannot be used to determine the instruction sequence because there may be a software loop at work which causes the program counter to cycle backwards even though it's really the next instruction executing.

Queueing of Flow Control Operations

Flow control operations are not done until sometime after the next instruction queues. This is necessary to determine address miss-predicts during the flow control operation. Waiting until the next instruction queues avoids the problem of false mis-predictions. A consequence of waiting for the next instruction to queue is that flow control operations may only issue from one of the first seven queue slots relative to the head of the queue. Note however that if the instruction queue is full the flow control operation will issue anyway otherwise the core could become deadlocked. When the core issues a flow control operation because the queue is full it will most likely cause a branch-miss state, which may reduce performance.

Issue Logic

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

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

Issue Rate

The functional units of FT64 include two alu's, a floating-point unit, a memory unit, and a flow control unit (branch unit). Instructions may be issued to any and all functional units during a single clock cycle. The memory unit can handle three requests at the same time. As a result, up to seven instructions may be issued in a single clock cycle. In practice fewer instructions will be ready to be issued. The author has noted, with limited testing, that issuing a third memory operation in the same clock cycle is rarely done. The memory unit is typically 2/3 occupied or less. The design could likely be reduced in size by omitting the third cache read port and support for a third memory operation in the issue logic with little loss of performance.

Execute Logic

Instructions are executed on functional units after they have been issued to the unit. The execution logic for FT64 consists of two alu's, a floating-point unit, a flow control unit and a memory unit.

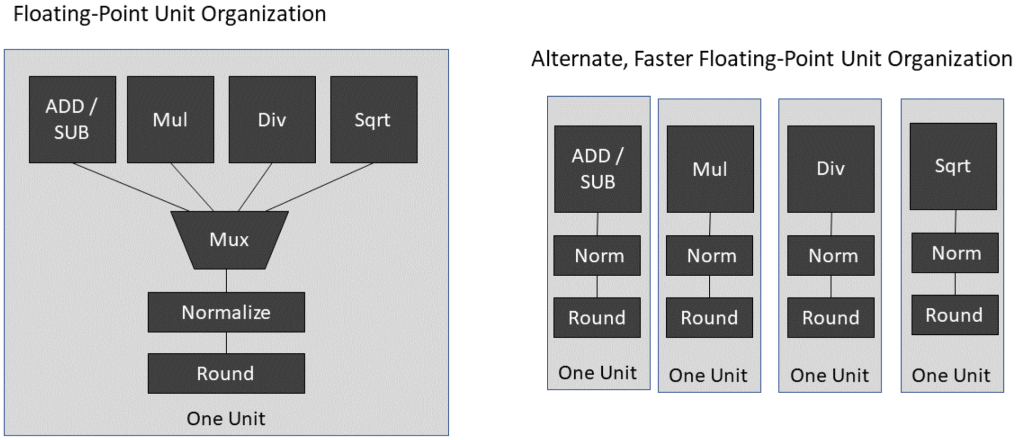
ALU's

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

Floating Point Unit

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

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



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

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

Flow Control Unit

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

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

The flow control unit has a small alu to calculate register increment or decrement, stack pointer adjustment, and return addresses. The fcu's alu is called FT64_fcu_calc.v to avoid confusion with FT64's master alu.

Memory Unit

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

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

Operating Levels

The core has four operating levels. The highest operating level is operating level zero which is called the machine operating level. Operating level zero has complete access to the machine. Other operating levels may have more restricted access. When an interrupt occurs, the operating level is set to the machine level. The core vectors to an address depending on the current operating level.

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

Switching Operating Levels

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

Privilege Levels

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

Exceptions

External Interrupts

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

If running with SMT then an external interrupt will be processed only by the even numbered thread to prevent the same interrupt from being processed twice. Other exceptions may occur on either thread.

Polling for Interrupts

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

Effect on Machine Status

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

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

Exception Stack

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

Exception Vectoring

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

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

Reset

The core begins executing instructions at address $FFFFFFFFFFFC0100. All registers are in an undefined state. Register set #0 is selected.

Precision

Exceptions in FT64 are precise. They are processed according to program order of the instructions. If an exception occurs during the execution of an instruction, then an exception field is set in the reorder buffer. The exception is processed when the instruction commits which happens in program order. If the instruction was executed in a speculative fashion, then no exception processing will be invoked unless the instruction makes it to the commit stage.

Exception Cause Codes

The following table outlines the cause code for a given purpose. These codes are specific to FT64. Under the HW column an ‘x’ indicates that the exception is internally generated by the processor; the cause code is hard-wired to that use. An ‘e’ indicates an externally generated interrupt, the usage may vary depending on the system.

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

58	DPF	x	data page fault	
60	DBE	x	data bus error	
61				
62	NMI	x	Non-maskable interrupt	
230	RT	x	return selector	
231	LDCS	x	load code selector	
232	ZS LD	x	segment load exception	
233	DS LD	x		
234	ES LD	x		
235	FS LD	x		
236	GS LD	x		
237	HS LD	x		
238	SS LD	x		
239	CS LD	x		
240	SYS		Call operating system (FMTK)	
241			FMTK Schedule interrupt	
255	PFI		reserved for poll-for-interrupt instruction	

DBG

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

IADR

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

UNIMP

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

OFL

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

FLT

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

DRF, DWF, EXF

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

CPF, DPF

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

PRIV

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

STK

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

DBE

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

IBE

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

NMI

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

IPT – The Inverted Page Table

Overview

A page table stores address translations from virtual to physical addresses. The inverted page table stores the virtual address in the table at an offset that corresponds to a physical page in memory. Thus, a translation can just use the index into the inverted page table as the physical page. This mechanism requires searching for the virtual page number in the table in order to determine the index. One might think this would be time consuming, but usually a hash function is used to get a good guess at the entry desired.

Memory Usage

An inverted page table manages memory using only a single entry for each physical page of memory. As such it requires less memory than a paged memory management system would.

Hash Function

A hash function is used to turn a virtual address into an inverted page table index. The hash function effectively reduces the number the number of bits in the virtual address to a number that corresponds to the table size. Hash function inputs are the virtual page number and the system’s current randomization key.

Randomization Key

As an aid to system security and integrity memory pages are allocated in a randomized fashion. Each time the system is restarted pages will be allocated in a different order. This is accomplished using a random key value in the hash function used to find virtual to physical address translations. the randomization key is determined from a random source in the system.

TLB

Usually an inverted page table is also used with a tlb to store address translations for faster access. In the case of the FT64v7SoC system-on-chip however there is no caching of address translations. Instead the entire inverted page table is stored in block ram resources within the FPGA. Translation will typically require only a small number of accesses to the block ram memory. This is contrasted with access to main memory which has a higher latency.

TLB – The Translation Lookaside Buffer

Overview

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

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

The TLB is manipulated with the TLB instruction.

Size / Organization

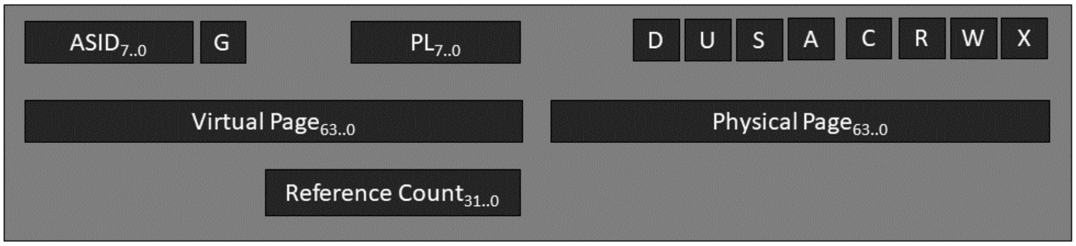
The core uses a 256 entry TLB (translation look-aside buffer) to support virtual memory. The TLB supports variable page sizes from 8kB to 2MB. The TLB is organized as a sixteen-way sixteen-set cache. The TLB processes all addresses leaving the core including both code and data addresses.

Updating the TLB

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

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

TLB Entries



G = Global

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

ASID = address space identifier

The ASID field in the TLB entry must match the processor’s current ASID value for the translation to be considered valid, unless the G bit is set. If the G bit is set in the TLB entry, then the ASID field is ignored during the address comparison. The processor’s current ASID is located in the machine status register.

C = cache-ability bits

If the cache-ability bits are set to 001_b then the page is un-cached, otherwise the page is cached.

D = dirty bit

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

A = accessed bit

This bit is set when the page is accessed.

U = undefined usage

This bit is available for OS usage

S = address shortcut

R = read bit

This bit indicates that the page is readable.

W = write bit

This bit indicates that the page is writeable

X = execute bit

This bit indicates that the page contains executable code

R,W,X = valid bit

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

Page Table Entry

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

31 16	15 8	7	6	5	4	3	2	1	0
$\sim_{16}$	PL	D	U	S	A	C	R	W	X
Reference Counter$_{32}$									
Page Number$_{31..0}$									
Page Number$_{63..32}$									

Bit			
0	X	1 = executable	Together these three fields combined indicate if the page is present. It must be at least one of readable, writeable, or executable.
1	W	1 = page writeable	
2	R	1 = readable	
3	C	1 = cache disabled	
4	A	1 = accessed	
5	S	1 = shortcut translation	Translation shortcut bit eg (8MiB pages)
6	U	undefined usage	available to be used by OS
7	D	1=dirty	
8 to 15	PL	Privilege level	
16 to 31	PN	Memory Page Number	High order bits of the page number may not be required and should be set to zero.
32 to 50			

Page Directory Entry

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

TLB Registers

TLBWired (#0h)

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

TLBIndex (#1h)

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

TLBRandom (#2h)

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

TLBPageSize (#3h)

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

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

TLBPhysPage (#5h)

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

<table>
<tr><td>63</td><td align="right">0</td></tr>
<tr><td colspan="2" align="center">Physical Page Number</td></tr>
</table>

TLBVirtPage (#4h)

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

63	0
Virtual Page Number	

TLBASID (#7h)

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

63 32	31 24	23 16	13 11	10	9	8	7	6	5 3	2	1	0
-----	PL	ASID	PgSz	G	D	U	S	A	C	R	W	X

TLBAFC (#12) – Aging Frequency Control

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

Simplified Paged Memory Management Unit

Overview

One option for memory management is a simplified paged memory management unit. Memory management by the MMU includes virtual to physical address mapping and read/write/execute permissions. The MMU divides memory into 8kiB or 512kiB pages depending on the setting in PCR2. The FPGA board in use has 512MiB of ram onboard. The simplified MMU is setup to map only this amount of ram.

8kiB pages

Processor address bits 13 to 22 are used as a ten-bit index into a mapping table to find the physical page. The MMU remaps the ten address bits into a sixteen-bit value used as address bits 13 to 28 when accessing a physical address. The lower thirteen bits of the address pass through the MMU unchanged. The maximum amount of memory that may be mapped in the MMU is 8MiB per map out of a pool of 512MiB. Only addresses with the most significant three bits set to zero are mapped.

512kiB pages

Some tasks require a lot of memory and an 8MiB map isn't sufficient. For instance, while in machine mode the core requires access to the entire address range. A memory page size of 512kiB may be selected by setting the bit corresponding to the memory map in PCR2.

Processor address bits 19 to 28 are used as a ten-bit index into a mapping table to find the physical page. The MMU remaps the ten address bits into a ten-bit value used as address bits 19 to 28 when accessing a physical address. The lower 19 bits of the address pass through the MMU unchanged. The maximum amount of memory that may be mapped in the MMU is 512MiB per map out of a pool of 512MiB. Only addresses with the most significant three bits set to zero are mapped.

Map Tables

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

Map Caching / TLB

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

Operate Key

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

Access Key

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

Address Pass-through

Addresses pass through the MMU unaltered until the mapping enable bit is set. Until mapping is enabled, the physical address will match the virtual address. Additionally address bits 0 to 12 pass through the MMU unaltered.

Mapping Table Layout

	D20	D19	D18	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
000	S1	S0	W	R	X	PA28	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	PA15	PA14	PA13
004	S1	S0	W	R	X	PA28	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	PA15	PA14	PA13
			…																		
FFC	S1	S0	W	R	X	PA28	PA27	PA26	PA25	PA24	PA23	PA22	PA21	PA20	PA19	PA18	PA17	PA16	PA15	PA14	PA13

PAnn = physical address bit
X = executable page indicator.
W = writeable data page indicator.
R = readable data page indicator.
Note the low order six bits are not used for 512kiB pages.
S1,S0 = two bits for program use

PCR- Paging Control Register Layout

31	30 14	13 8	7 6	5 0
PE	~$_{18}$	AKey$_6$	~	OKey$_6$

PE = Paging Enable (1=enabled, 0 = disabled)
AKey = Access Key
OKey = Operate Key

PCR2 – Page Size

This register controls the memory page size. Each bit in the register corresponds to a memory map. Memory may be paged in either 8kiB or 512kiB pages. All pages in a map have the same size.

Latency

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

Telescopic Memory

Telescopic memory is used to increase the performance of garbage collection routines and other similar software.

Telescopic memory stores pointer values at progressively lower address resolutions. When a pointer store to main memory occurs, the address is shifted right eleven times and a single bit is set in the telescopic memory to indicate that a pointer store occurred. At the same time the address is also shifted right twenty-two times and a single bit is set in the telescopic memory to indicate a pointer store occurred. This generates two, bit tables indicating where pointer stores occurred. Three write accesses to update the telescopic memory occur in parallel to main memory access. A single store pointer (sptr) instruction triggers all three accesses.

For instance, with a 256MB memory, the first level table contains 128kB (16k words). The resolution of the address of the stored pointer is 256 bytes. The second level table contains 32 bytes or four words. The resolution of the address of the stored pointer is 1MB.

Instruction Set Description

Formats

Most instructions have a fixed 32 bit format. There are only a handful of different instruction formats. The opcode, register read Ra, Rb, Rc, and Rt fields always occur in the same place in an instruction to simplify decoding and keep the register read address which is needed prior to enqueue at a fixed decoding location.

<table>
<tr><td colspan="3"></td><td colspan="10">Immed$_{14}$</td><td colspan="3">Rt$_5$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>RI</td></tr>
<tr><td colspan="3"></td><td colspan="10">Immed$_{14}$</td><td colspan="3">Rt$_5$</td><td>Imm$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>LUI</td></tr>
<tr><td colspan="3"></td><td colspan="9">Disp$_{13..6}$</td><td>Rb$_5$</td><td colspan="3">Disp$_{4..0}$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>MS</td></tr>
<tr><td colspan="3"></td><td colspan="6">Funct$_6$</td><td colspan="3">Sz$_3$</td><td>Rb$_5$</td><td colspan="3">Rt$_5$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>RR</td></tr>
<tr><td colspan="3"></td><td colspan="6">01$_6$</td><td colspan="3">Sz$_3$</td><td>Funct$_5$</td><td colspan="3">Rt$_5$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>R1</td></tr>
<tr><td colspan="3"></td><td colspan="6">Funct$_6$</td><td colspan="3">Funct$_3$</td><td>Rb$_5$</td><td colspan="3">Rt$_5$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>SR</td></tr>
<tr><td colspan="3"></td><td colspan="6">Funct$_6$</td><td colspan="3">Funct$_3$</td><td>Immed$_5$</td><td colspan="3">Rt$_5$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>SI</td></tr>
<tr><td>Funct$_4$</td><td>Rg$_3$</td><td colspan="2">~$_3$</td><td colspan="2">Bw</td><td colspan="2">Bo</td><td colspan="4">Rc$_5$</td><td>Bw$_5$</td><td colspan="3">Rt$_5$</td><td>Bo$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>BF</td></tr>
<tr><td colspan="3"></td><td colspan="9">Disp$_{11..3}$</td><td>Rb$_5$</td><td>D$_{2..1}$</td><td colspan="2">Cond$_3$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>BD</td></tr>
<tr><td colspan="3"></td><td colspan="9">Disp$_{11..3}$</td><td>Bitno$_{5..1}$</td><td>D$_{2..1}$</td><td>B$_0$</td><td>Cnd$_2$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>BB</td></tr>
<tr><td colspan="3"></td><td colspan="9">Disp$_{11..3}$</td><td>Imm$_{7..3}$</td><td>D$_{2..1}$</td><td colspan="2">Imm$_{2..0}$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>BE</td></tr>
<tr><td colspan="3"></td><td colspan="5">~$_8$</td><td colspan="4">Rc$_5$</td><td>Rb$_5$</td><td colspan="3">Cond$_5$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>BR</td></tr>
<tr><td colspan="3"></td><td>0</td><td colspan="4">Funct$_3$</td><td colspan="2">Fn$_2$</td><td>~</td><td>Sc$_2$</td><td>Rb$_5$</td><td colspan="3">Rt$_5$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>MXL</td></tr>
<tr><td colspan="3"></td><td>1</td><td colspan="4">Funct$_3$</td><td colspan="4">Rc$_5$</td><td>Rb$_5$</td><td>Fn$_2$</td><td>~</td><td>Sc$_2$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>MXS</td></tr>
<tr><td colspan="3"></td><td colspan="2">Op$_2$</td><td colspan="2">OL$_2$</td><td colspan="6">Regno$_{10}$</td><td colspan="3">Rt$_5$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>CSR</td></tr>
<tr><td colspan="3"></td><td colspan="5">Funct$_5$</td><td colspan="2">Prec$_2$</td><td colspan="2">Rm$_3$</td><td>Rb$_5$</td><td colspan="3">Rt$_5$</td><td>Ra$_5$</td><td>L$_2$</td><td>Opcode$_6$</td><td>FLT</td></tr>
<tr><td colspan="3"></td><td colspan="14">Address$_{24}$</td><td>L$_2$</td><td>Opcode$_6$</td><td>JC</td></tr>
</table>

There are a handful of additional formats primarily for control type instructions. See the instruction details for the exact format used and additional information.

Format	Instruction Group
RI	register-immediate and load with displacement
LUI	load upper immediate
MS	memory store with displacement
RR	register-register, two source registers
R1	single source register
SR	shift register-register
SI	shift register-immediate
BF	bitfield
BD	branch with displacement
BB	branch on bit set / clear
BE	branch on equal immediate
BR	branch to register
MXL	memory indexed load
MXS	memory indexed store
CSR	control and status register access
JC	jump and call
FLT	floating-point

L_2	Instruction Length
0	32 bit instruction
1	48 bit instruction
2	16 bit compressed instruction
3	16 bit compressed instruction

Compressed Instruction Formats

15 12	11 8, 5		76	4 0		
0000	00_5		10	0_5		NOP
0000	Amt$_{[7..3]}$		10	31_5		ADDISP
0000	Amt$_5$		10	Ra/Rt$_5$		ADDI
0001	Amt$_5$		10	Rt$_5$		LDI / SYS (SYS if Rt = 0)
0010	Amt$_5$		10	Ra/Rt$_5$		RET / ANDI (RET if Ra=0)
0011	Amt$_5$		10	Rt$_5$		SHLI
0100	Amt$_5$		10	00	Ra/Rt'$_3$	SHRI
0100	Amt$_5$		10	01	Ra/Rt'$_3$	ASRI
0100	Amt$_5$		10	10	Ra/Rt'$_3$	ORI
0100	00	Rb'$_3$	10	11	Ra/Rt'$_3$	SUB
0100	01	Rb'$_3$	10	11	Ra/Rt'$_3$	AND
0100	10	Rb'$_3$	10	11	Ra/Rt'$_3$	OR
0100	11	Rb'$_3$	10	11	Ra/Rt'$_3$	XOR
0101	Address$_{9..5}$		10	Address$_{4..0}$		CALL
0110			10			reserved
0111	Disp$_{9..5}$		10	Disp$_{4..0}$		BRA
10	Disp$_7$		10	Ra$_5$		BEQZ
11	Disp$_7$		10	Ra$_5$		BNEZ
0000	Rt$_5$		11	Ra$_5$		MOV
0001	Rb$_5$		11	Ra/Rt$_5$		ADD
0010	Rt$_5$		11	Ra$_5$		JALR
0011	??????		11	Ra$_5$		PUSH / CS / DS / ES / SS / FS / GS
The following two instructions have SP as an implied register read						
0100	Disp$_{6..2}$		11	Rt$_5$		LH Rt,d[SP]
0101	Disp$_{7..3}$		11	Rt$_5$		LW Rt,d[SP]
The following two instructions have FP as an implied register read						
0110	Disp$_{6..2}$		11	Rt$_5$		LH Rt,d[FP]
0111	Disp$_{7..3}$		11	Rt$_5$		LW Rt,d[FP]
The following two instructions have SP as an implied register read						
1000	Disp$_{6..2}$		11	Rb$_5$		SH Rb,d[SP]
1001	Disp$_{7..3}$		11	Rb$_5$		SW Rb,d[SP]
The following two instructions have FP as an implied register read						
1010	Disp$_{6..2}$		11	Rb$_5$		SH Rb,d[FP]
1011	Disp$_{7..3}$		11	Rb$_5$		SW Rb,d[FP]
1100	d$_{5..4}$	Rt'$_3$	11	d$_{3..2}$	Ra'$_3$	LH Rt,d[Ra]
1101	d$_{6..5}$	Rt'$_3$	11	d$_{4..3}$	Ra'$_3$	LW Rt,d[Ra]
111?	d$_{5..4}$	Rb'$_3$	11	d$_{3..2}$	Ra'$_3$	SH Rb,d[Ra]
1111	d$_{6..5}$	Rb'$_3$	11	d$_{4..3}$	Ra'$_3$	SW Rb,d[Ra]

Operation Sizes

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

SIMD

Single instruction multiple data operations treat the 64 bit operands as multiple independent lanes of data depending on the size selected. For a half-word size the operands are treated as two independent 32 bit operands. For a character size the operands are treated as four independent 16 bit operands. SIMD operations are selected by setting the parallel operation bit in the instruction (the most significant bit of the size field).

Arithmetic Operations

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

Relational Operations

There are handful of instructions for determining the relationship between values. These include an assortment of set instructions. The set instruction group is asymmetrical because there are means to implement set instructions with other already existing instructions. For instance, there are no set greater than or set greater than or equal set instructions that take register operands because the less than versions of the set instructions can be used to the same effect by swapping operands around. There is no set if equal or set if not equal because the xnor and xor functions perform much the same operation.

Logical Operations

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

Shift Operations

There is a full complement of shift operations including left and right unsigned and signed shifts, and left and right rotate instructions. Shift instructions are only supported on alu #0.

Bitfield Operations

The CC64 compiler has direct support for bitfields and bitfield instructions support these operations. Bitfield operations include insert, extract, set, change and clear operations.

Memory Operations

Memory operations include loads and stores of bytes, words or half-words. There isn't yet a full complement of memory operations in order to keep the size of the core smaller. The core can perform loads and stores using indexed addressing.

Loads

Loads may execute speculatively. They may occur out of program order. A load will be issued provided there is no address overlap with a previous memory operation.

Stores

Stores will not be issued by the core until it is known that the store can be guaranteed to execute. Unlike a load, a store cannot be executed speculatively. This means no prior instruction will exception and no change of control flow will take place before the store. Stores always write through to memory. A store instruction can't be committed to the machine state until exceptions are checked for during the store operation. Until the operation to memory is complete the store can't commit. However, the store operation is marked as "done" as soon as it's issued so that other instructions may continue to execute. Much of the latency of a store operation is then hidden.

AMO

There is a set of AMO memory operations (atomic memory operations). These operations use a read-modify-write cycle to modify the memory location. There is a small ALU associated with the AMO operations that allows some basic functions to be performed on the data between the read and write cycle.

Control Flow Instructions

Control flow instructions include call, return, jumps and branches, breakpoint and return instructions. All controls transfers take place at the fetch stage of the processor and if a predicted fetch direction turns out to be incorrect it is corrected during the execution stage of the instruction. Instructions which use calculated addresses unknown until run-time make use of the branch target buffer to predict the address.

Jump

There is a single jump instruction which modifies the low order 28 bits of the program counter, allowing a jump within the same 256MB region of memory. This range is probably sufficient for most applications when an mmu is present.

Call

There is a single call instruction which modifies the low order 28 bits of the program counter. Call instruction flow transfer takes place immediately in the fetch stage of the core. The call return address is pushed onto the return address stack predictor. When the call instruction executes, the return address is stored in the return address register. The JAL instruction may also be used to call subroutines and allows a register indirect call to be performed.

Return

Return instructions are predicted during the fetch stage of the core using a return address predictor. The return instruction is also capable of adjusting the stack pointer.

Conditional Branches

Conditional branches are predicted using a (2,2) correlating branch predictor.

Breakpoint

Breakpoint instructions cause some of the cores state to be stored on internal stacks. The stored state includes the program counter, interrupt mask, privilege level, and operating level. The internal stacks are eight entries deep; this is the maximum amount of nesting that can occur. The breakpoint instruction specifies the number of instruction words to skip over to determine point of return.

Exception (breakpoint) Return

The exception return instruction unstacks the state previously stacked by a breakpoint instruction.

Clock cycles

The clock cycles indicated are only approximate. An attempt has been made to give a relative indication between instructions of the clocks required. The core hasn’t under gone significant timing measurements. Many common instructions which can execute in only ½ of a clock cycle, for example add and subtract, indicate a clock cycle time of 1. A number of instructions have single cycle execution times because they may only execute on ALU #0.

ABS – Absolute Value

Description:

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

Instruction Format:

01_6	Sz_3	4_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
If Ra < 0
   Rt = -Ra
else
   Rt = Ra
```

Exceptions: none

Notes:

Sz_3	
0	reserved
1	reserved
2	reserved
3	reserved
4	Byte Parallel
5	Char Parallel
6	Half Parallel
7	Word

ADD - Addition

Description:

Add two values. The first operand must be in a register. The second operand may be in a register or may be an immediate value specified in the instruction.

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	04h$_6$

L$_2$: 0 = 14 bit constant, 1 = 30 bit constant

04$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions: none

Notes:

For parallel operation forms the registers are treated as if they were a group of registers corresponding to the size selected. And the same operation is performed on each part of the register. For parallel forms the entire register is updated.

Sz$_3$	
0	reserved
1	reserved
2	reserved
3	reserved
4	Byte Parallel
5	Char Parallel
6	Half Parallel
7	Word

ADD3 - Addition

Description:

Add three values. All operands must be in registers.

Instruction Format:

04_6	$\sim_{14}$	Rc_5	Rb_5	Rt_5	Ra_5	1_2	$02h_6$

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions: none

ADDV – Addition with Overflow Detection

Description:

Add two values. Both operands must be in registers. If an overflow occurs an overflow exception may be generated if enabled in the arithmetic exception control register.

Instruction Format:

02_6	$\sim_3$	Rt_5	Rb_5	Ra_5	L_2	$02h_6$

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions:

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

Notes:

AMO – Atomic Memory Operation

Description:

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

Instruction Format:

Funct$_4$	A	R	Sz$_3$	Rt$_5$	Rb$_5$	Ra$_5$	0$_2$	2Fh$_6$

Instruction Format (immediate operand):

Funct$_4$	A	R	Sz$_3$	Imm$_5$	Rt$_5$	Ra$_5$	0$_2$	3Fh$_6$

Funct$_4$	Mnemonic	Operation Performed	
00	swap	swap	memory[Ra] = Rb
01	add	addition	memory[Ra] = memory[Ra] + Rb
02	and	bitwise and	memory[Ra] = memory[Ra] & Rb
03	or	bitwise or	memory[Ra] = memory[Ra] | Rb
04	xor	bitwise exclusive or	memory[Ra] = memory[Ra] ^ Rb
05	shl	shift left	memory[Ra] = memory[Ra] << Rb
06	shr	shift right	memory[Ra] = memory[Ra] >> Rb
07	min	minimum	memory[Ra] = memory[Ra] < Rb ? memory[Ra] : Rb
08	max	maximum	memory[Ra] = memory[Ra] >Rb ? memory[Ra] : Rb
09	minu	minimum unsigned	memory[Ra] = memory[Ra] < Rb ? memory[Ra] : Rb
0A	maxu	maximum unsigned	memory[Ra] = memory[Ra] > Rb ? memory[Ra] : Rb

Sz$_2$	
0	Byte
1	Char
2	Half
3	Word

Acquire and release bits determine the ordering of memory operations.

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

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

All combinations of A, R are allowed.

AND – Bitwise And

Description:

Perform a bitwise ‘and’ operation between operands. The immediate constant is sign extended before use.

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	08h$_6$

L$_2$: 0 = 14 bit constant, 1 = 30 bit constant

08$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 0.5

Execution Units: All ALUs

Exceptions: none

ASL – Arithmetic Shift Left

Description:

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

Instruction Format:

$2Fh_6$	2_3	Rb_5	Rt_5	Ra_5	0_2	$02h_6$

Immediate, shift count 0 to 31

$0Fh_6$	2_3	Imm_5	Rt_5	Ra_5	0_2	$02h_6$

Immediate, shift count 32 to 63

$1Fh_6$	2_3	Imm_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions:

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

ASR – Arithmetic Shift Right

Description:

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

Instruction Formats:

Register

2Fh$_6$	3$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Immediate, shift count 0 to 31

0Fh$_6$	3$_3$	Imm$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Immediate, shift count 32 to 63

1Fh$_6$	3$_3$	Imm$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

AUIPC – Add Upper Immediate to PC

Description:

This instruction forms the sum of the program counter and an immediate value shifted left 30 times. The result is then placed in the target register. The low order 30 bits of the target register are zeroed out. When long forms of immediates are used, a full 64-bit address may be formed.

Instruction Format:

Immed$_{18..5}$	Rt$_{5}$	Imm$_{4..0}$	L$_{2}$	03h$_{6}$

L$_{2}$: 0 = 19 bit constant, 1 = 35 bit constant

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions: none

Notes:

BBC –Branch if Bit Clear

Description:

If the specified bit in a register is clear, the target address is computed and loaded into the program counter. The branch is relative to the address of the branch instruction. If the branch branches back to itself a branch exception will be generated.

Instruction Format:

31 23	22 18	1716	15	1413	12 8	76	5 0	
Disp$_{11..3}$	Bitno$_{5..1}$	D$_{2..1}$	B$_{0}$	1$_{2}$	Ra$_{5}$	L$_{2}$	30h$_{6}$	BB

Operation:

```
if (Ra[bitno]=0)
        pc[31:1] = pc[31:1] + displacement
```

Clock Cycles: 2 with accurate prediction, otherwise 8 or more

Execution Units: FCU Only

Exceptions: branch target address

BBS –Branch if Bit Set

Description:

If the specified bit in a register is set, the target address is computed and loaded into the program counter. The branch is relative to the address of the branch instruction. If the branch branches back to itself a branch exception will be generated.

Instruction Format:

31 23	22 18	1716	15	1413	12 8	76	5 0	
Disp$_{11..3}$	Bitno$_{5..1}$	D$_{2..1}$	B$_0$	0$_2$	Ra$_5$	L$_2$	30h$_6$	BB

Operation:

```
if (Ra[bitno]=1)
        pc[31:1] = pc[31:1] + displacement
```

Clock Cycles: 2 with accurate prediction, otherwise 8 or more

Execution Units: FCU Only

Exceptions: branch target address

Bcc – Conditional Branch

Description:

If the branch condition is true, the target address is computed and loaded into the program counter. The branch is relative to the address of the branch instruction. If the branch branches back to itself a branch exception will be generated.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
Disp$_{11..3}$	Rb$_5$	D$_{2..1}$	Cond$_3$	Ra$_5$	L$_2$	30h$_6$	BD

Opcode$_6$	Cond$_2$	Mne.	
30h	0	BEQ	Ra = Rb signed
	1	BNE	Ra <> Rb
	2	BLT	Ra < Rb
	3	BGE	Ra >= Rb
	4	BAND	Ra && Rb
	5	BOR	Ra || Rb
	6	BLTU	Ra < Rb (unsigned)
	7	BGEU	Ra >= Rb (unsigned)

8	FBEQ	Ra = Rb (floating point)
9	FBNE	Ra != Rb (floating point)
10	FBLT	Ra < Rb (floating point)
11	FBGE	Ra >= Rb (floating point)
12		reserved
13		reserved
14		reserved
15	FBUN	register Ra contains unordered floating point constant

Clock Cycles:

Typically, 2 with correct branch outcome and target prediction.

Execution Units: FCU Only

Exceptions: branch target

BCDADD - Register-Register

Description:

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

Instruction Format:

00_6	0_3	Rb_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra + Rb

Exceptions: none

BCDMUL - Register-Register

Description:

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

Instruction Format:

00_6	2_3	Rb_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Operation:

Rt = Ra * Rb

Exceptions: none

BCDSUB - Register-Register

Description:

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

Instruction Format:

00_6	1_3	Rb_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra - Rb

Exceptions: none

BEQ –Branch if Equal

Description:

If two registers are equal, an eleven-bit sign extended displacement is added to the program counter. The branch is relative to the address of the branch instruction. If the branch branches back to itself a branch exception will be generated.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
Disp$_{11..3}$	Rb$_5$	D$_{2..1}$	0$_3$	Ra$_5$	L$_2$	30h$_6$	BD

Operation:

```
if (Ra = Rb)
        pc[31:1] = pc[31:1] + displacement
```

Clock Cycles: Typically, 2 with correct branch outcome and target prediction.

Execution Units: FCU Only

Exceptions: branch target

BEQI –Branch if Equal Immediate

Description:

If a register is equal to an eight-bit sign extended value, then a target address is loaded into the program counter. The branch is relative to the address of the branch instruction. This instruction is useful for implementing case statements based on small values.

Target Address Computation: A sign extended displacement value is added to bits 1 to 31 of the program counter.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
$Disp_{11..3}$	$Imm_{7..3}$	$D_{2..1}$	$Imm_{2..0}$	Ra_5	L_2	$32h_6$	BE

Operation:

```
if (Ra = Immediate)
        pc[31:1] = pc[31:1] + displacement
```

Clock Cycles: Typically, 2 with correct branch outcome and target prediction.

Execution Units: FCU Only

Exceptions: branch target

BFCHG – Bitfield Change

Description:

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

Instruction Format:

47 44	43 33	32 30	29	28	27 23	22 18	17 13	12 8	76	5 0
2_4	Da_{11}	Rg_3	Bw	Bo	Rt_5	Da_5	Bw_5	Bo_5	1_2	$22h_6$

Rg_3 Bit	
0	1= Bo is a register spec, 0 = Bo is a six bit immediate
1	1 = Bw is a register spec, 0 = Bw is a six bit immediate
2	1 = Da is a register spec, 0 = Da is an sixteen bit immediate

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

BFCLR – Bitfield Clear

Description:

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

Instruction Format:

47 44	43 33	32 30	29	28	27 23	22 18	17 13	12 8	76	5 0
1_4	Da_{11}	Rg_3	Bw	Bo	Rt_5	Da_5	Bw_5	Bo_5	1_2	$22h_6$

Rg_3 Bit	
0	1= Bo is a register spec, 0 = Bo is a six bit immediate
1	1 = Bw is a register spec, 0 = Bw is a six bit immediate
2	1 = Da is a register spec, 0 = Da is an sixteen bit immediate

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

Notes:

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

BFEXT – Bitfield Extract

Description:

A bitfield is extracted from the source register Da by shifting to the right and ‘and’ masking. The result is sign extended to the width of the machine. This instruction may be used to sign extend a value from an arbitrary bit position.

Instruction Format:

47 44	43 33	32 30	29	28	27 23	22 18	17 13	12 8	76	5 0
5_4	Da_{11}	Rg_3	Bw	Bo	Rt_5	Da_5	Bw_5	Bo_5	1_2	$22h_6$

Rg_3 Bit	
0	1= Bo is a register spec, 0 = Bo is a six bit immediate
1	1 = Bw is a register spec, 0 = Bw is a six bit immediate
2	1 = Da is a register spec, 0 = Da is an sixteen bit immediate

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

Notes:

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

BFEXTU – Bitfield Extract

Description:

A btifield is extracted from the source register Da by shifting to the right and ‘and’ masking. The result is zero extended to the width of the machine. This instruction may be used to zero extend a value from an arbitrary bit position.

Instruction Format:

47 44	43 33	32 30	29	28	27 23	22 18	17 13	12 8	76	5 0
6_4	Da_{11}	Rg_3	Bw	Bo	Rt_5	Da_5	Bw_5	Bo_5	1_2	$22h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

BFFFO – Bitfield Find First One

Description:

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

Instruction Format:

47 44	43 33	32 30	29	28	27 23	22 18	17 13	12 8	76	5 0
8_4	Da$_{11}$	Rgs$_3$	Bw	Bo	Rt$_5$	Da$_5$	Bw$_5$	Bo$_5$	L$_2$	Opcode$_6$

Rg$_3$ Bit	
0	1= Bo is a register spec, 0 = Bo is a six bit immediate
1	1 = Bw is a register spec, 0 = Bw is a six bit immediate
2	1 = Da is a register spec, 0 = Da is an sixteen bit immediate

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

BFINS – Bitfield Insert

Description:

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

Instruction Format:

3_4	Rg_3	Da_3	Bw	Bo	Rt_5	Da_5	Bw_5	Bo_5	L_2	$22h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

BFINSI – Bitfield Insert Immediate

Description:

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

Instruction Format:

4_4	Rg_3	Da_3	Bw	Bo	Rt_5	Da_5	Bw_5	Bo_5	L_2	$22h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

BGE –Branch if Greater or Equal

Description:

If register Ra is greater than or equal to register Rb then the target address is computed and loaded into the program counter. Values in registers are treated as signed values. The branch is relative to the address of the branch instruction. This instruction may also be used to branch on less than or equal by swapping the registers around.

Instruction Format:

31 23	22 18	17 16	15 13	12 8	7 6	5 0	
Disp$_{11..3}$	Rb$_5$	D$_{2..1}$	3$_3$	Ra$_5$	L$_2$	30h$_6$	BD

Operation:

```
if (Ra < Rb)
        pc[31:1] = pc[31:1] + displacement
```

Clock Cycles: Typically, 2 with correct branch outcome and target prediction.

Execution Units: FCU Only

Exceptions: branch target address

BGEU –Branch if Greater or Equal Unsigned

Description:

If register Ra is greater than or equal to register Rb then the target address is computed and loaded into the program counter. Values in registers are treated as unsigned values. The branch is relative to the address of the branch instruction. This instruction may also be used to branch on less than or equal by swapping the registers around.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
Disp$_{11..3}$	Rb$_5$	D$_{2..1}$	7$_3$	Ra$_5$	L$_2$	30h$_6$	BD

Operation:

```
if (Ra >= Rb)
        pc[31:1] = pc[31:1] + displacement
```

Clock Cycles: Typically, 2 with correct branch outcome and target prediction.

Execution Units: FCU Only

Exceptions: branch target address

BLT –Branch if Less Than

Description:

If register Ra is less than register Rb then the target address is computed and loaded into the program counter. Values in registers are treated as signed values. The branch is relative to the address of the branch instruction. This instruction may also be used to branch on greater than by swapping the registers around.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
Disp$_{11..3}$	Rb$_5$	D$_{2..1}$	2$_3$	Ra$_5$	L$_2$	30h$_6$	BD

Operation:

```
if (Ra < Rb)
        pc[31:1] = pc[31:1] + displacement
```

BLTU –Branch if Less Than Unsigned

Description:

If register Ra is less than register Rb then the target address is computed and loaded into the program counter. Values in registers are treated as unsigned values. The branch is relative to the address of the branch instruction. This instruction may also be used to branch on greater than by swapping the registers around.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
$\text{Disp}_{11..3}$	Rb_5	$\text{D}_{2..1}$	6_3	Ra_5	L_2	30h_6	BD

Operation:

```
if (Ra < 0)
        pc[31:1] = pc[31:1] + displacement
```

BMM – Bit Matrix Multiply

BMM Rt, Ra, Rb

Description:

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

Instruction Format:

03$_6$	~	Fn$_2$	Rt$_5$	Rb$_5$	Ra$_5$	0$_2$	02h$_6$

Fn$_2$	Function
0	MOR
1	MXOR
2	MORT (MOR transpose)
3	MXORT (MXOR transpose)

Operation:

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

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions: none

Notes:

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

BNAND –Branch on Logical Nand

Description:

If the logical nand of two registers is true, an eleven-bit sign extended displacement is added to the program counter. The branch is relative to the address of the branch instruction. If the branch branches back to itself a branch exception will be generated.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
Disp$_{11..3}$	Rb$_5$	D$_{2..1}$	4$_3$	Ra$_5$	L$_2$	10h$_6$	BD

Operation:

```
if (!(Ra && Rb))
        pc[31:1] = pc[31:1] + displacement
```

Clock Cycles: Typically, 2 with correct branch outcome and target prediction.

Execution Units: FCU Only

Exceptions: branch target

BNE –Branch if Not Equal

Description:

If the two registers are unequal, the target address is computed and loaded into the program counter. The branch is relative to the address of the branch instruction. If the branch branches back to itself a branch exception will be generated.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
Disp$_{11..3}$	Rb$_5$	D$_{2..1}$	1$_3$	Ra$_5$	L$_2$	30h$_6$	BD

Operation:

```
if (Ra <> 0)
        pc = pc + displacement
```

BNEI –Branch if Not Equal Immediate

Description:

If a register is not equal to an eight-bit sign extended value, then a target address is loaded into the program counter. The branch is relative to the address of the branch instruction.

Target Address Computation: A sign extended displacement value is added to bits 1 to 31 of the program counter.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
Disp$_{11..3}$	Imm$_{7..3}$	D$_{2..1}$	Imm$_{2..0}$	Ra$_{5}$	L$_{2}$	12h$_{6}$	BE

Operation:

```
if (Ra = Immediate)
        pc[31:1] = pc[31:1] + displacement
```

Clock Cycles: Typically, 2 with correct branch outcome and target prediction.

Execution Units: FCU Only

Exceptions: branch target

BNOR –Branch on Logical Nnor

Description:

If the logical nor of two registers is true, an eleven-bit sign extended displacement is added to the program counter. The branch is relative to the address of the branch instruction. If the branch branches back to itself a branch exception will be generated.

Instruction Format:

31 23	22 18	1716	15 13	12 8	76	5 0	
Disp$_{11..3}$	Rb$_5$	D$_{2..1}$	5_3	Ra$_5$	L$_2$	10h$_6$	BD

Operation:

```
if (!(Ra || Rb))
        pc[31:1] = pc[31:1] + displacement
```

Clock Cycles: Typically, 2 with correct branch outcome and target prediction.

Execution Units: FCU Only

Exceptions: branch target

BRK – Hardware / Software Breakpoint

Description:

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

For hardware interrupts a register set is selected automatically according to the hardware interrupt level (1 to 15). For a software interrupt register set #0 is selected. Registers from alternate register sets are available with the MOV instruction.

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

Item Stacked	CSR reg	
program counter	pc_stack	
operating level	ol_stack	available as a single CSR
privilege level	pl_stack	available as a single CSR
interrupt mask	im_stack	available as a single CSR
register set	rs_stack	available as a single CSR

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

On stack underflow a break exception is triggered.

Instruction Format:

31 26	25 21	20 17	16	15 8	7 6	5 0
User$_6$	S$_5$	IL$_4$	0	Cause Code$_8$	L$_2$	00h$_6$

S = skip 2 = software interrupt – return address is next instruction

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

IL$_4$ = the priority level of the hardware interrupt, the priority level at time of interrupt is recorded in the instruction, the interrupt mask will be set to this level when the instruction commits. This field is not used for software interrupts and should be zero.

Cause Code = numeric code associated with the cause of the interrupt.

The User$_6$ field may be used to pass constant data to the break handler.

Instruction Format:

31 26	25 21	20 17	16	15 13	12 8	7 6	5 0
User$_6$	S$_5$	IL$_4$	1	~$_3$	Ra$_5$	L$_2$	00h$_6$

S = word skip 2 = software interrupt – return address is next instruction

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

IL$_4$ = the priority level of the hardware interrupt, the priority level at time of interrupt is recorded in the instruction, the interrupt mask will be set to this level when the instruction commits. This field is not used for software interrupts and should be zero.

[Ra$_5$] = Cause Code = numeric code associated with the cause of the interrupt.

The User$_6$ field may be used to pass constant data to the break handler.

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

Compressed Instruction Format:

The compressed instruction format for the BRK instruction provides a short-form which may conserve code space when using a frequently occurring break instruction. Only cause codes 32-62 are supported for the short form. Cause code 63 is reserved for the interrupt polling instruction (PFI). The short form BRK is used only for software interrupts and the return address is the address of the instruction following the BRK. (Skip is implied = 1).

15 12	11 8	7 6	5	4 0
01h$_4$	Cause Code$_{4..1}$	10$_2$	C$_0$	00h$_5$

CACHE – Cache Command

CACHEX –

CACHE Cmd, d(Rn)
CACHE Cmd, d(Ra + Rc * scale)

Description:

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

Instruction Formats:

Displacement$_{14}$					Cmd$_5$	Ra$_5$	L$_2$	1Eh$_6$	CACHE Cmd,d16(Rn)
7h$_4$	Rc$_5$	2$_2$	~	Sc$_2$	Cmd$_5$	Ra$_5$	L$_2$	16h$_6$	CACHE Cmd,d(Ra+Rc*sc)

Commands:

Cmd$_5$	Mne.	Operation
00h		reserved
01h		reserved
02h	inviline	invalidate instruction cache line
03h	invic	invalidate entire instruction cache (address is ignored)
10h	disabledc	disable data cache
11h	enabledc	enable data cache
12h		invalidate data cache line
13h	invdc	invalidate entire data cache (address is ignored)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

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

Notes:

The displacement constant may be extended up to 64 bits.

Sc$_2$ Code	Multiply By
0	1
1	2
2	4

3	8

CALL – Call Method

Description:

Instruction Format:

This instruction loads the program counter with a constant value specified in the instruction. In addition, the address of the instruction following the CALL is stored in the return address register. The current code segment selector value is stored in the RS register (the upper 24 bits of the link register). This instruction may be used to implement subroutine calls.

This format has a 32MB range and is limited to the current code segment.

Immed$_{24}$	0$_{2}$	19h$_{6}$

This format has a 256GB range and allows a call into a different code segment.

Seg$_{2}$	Immed$_{38}$	1$_{2}$	19h$_{6}$

If an address range greater than 38 bits is required, then the JAL instruction must be used.

Seg$_{2}$	
0	ZS
1	ES
2	HS
3	CS

Software Sample:

```
mov2seg     hs,#$001234        ; load the hs with the target segment
call        hs:some_function
<…> other
call        hs:another_func
```

Execution Units: FCU

Clock Cycles:

CAS – Compare and Swap

Description:

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

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

Instruction Format:

Disp$_{16}$	Rst$_5$	Ra$_5$	25h$_6$

Operation:

```
if memory[Ra+displacement] = casreg
        memory[Ra + displacement] = Rst
        Rst = 1
else
        casreg = memory [Ra + displacement]
         Rst = 0
```

Assembler:

CAS Rt, displacement[Ra]

CHK – Check Register Against Bounds

Description:

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

Instruction Format:

31 18	17 13	12 8	76	5 0
Immediate_{14}	Rb_5	Ra_5	0_2	34h_6

31 26	25 23	22 18	17 13	12 8	76	5 0
34h_6	$\sim_3$	Rc_5	Rb_5	Ra_5	0_2	02h_6

Clock Cycles: 1

Exceptions: bounds check

Notes:

CLI – Clear Interrupt Mask

Description:

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

Instruction Format:

30_6	$\sim_5$	0_3	$\sim_5$	0_5	0_2	$02h_6$

Clock Cycles: 0.5

CMOVEZ – Conditional Move If Zero

Description:

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

Instruction Format:

28h$_6$	Rt$_5$	Rc$_5$	Rb$_5$	Ra$_5$	02h$_6$

Clock Cycles: 0.5

CxxxNZ – Conditional Op If Non-Zero

Description:

The conditional move if not zero instruction moves the contents of register Rb to the target register Rt if Ra is non-zero. Otherwise the contents of register Rc or a signed extended immediate value are moved to the target register.

Instruction Format:

47 42	4139	38 23	22 18	17 13	12 8	7 6	5 0
29h$_6$	Op$_3$	Immed$_{15..0}$	Rb$_5$	Rt$_5$	Ra$_5$	1$_2$	02h$_6$

Op$_3$	
0	CMOVNZ
1	CADDNZ
2	
3	
4	CMOVNZI
5	CADDNZI
6	
7	

Clock Cycles: 0.5

CNTLO – Count Leading Ones

Description:

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

Instruction Format:

01_6	Sz_3	1_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

Sz_3	
0	Byte
1	Char
2	Half
3	Word

CNTLZ – Count Leading Zeros

Description:

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

Instruction Format:

01_6	Sz_3	0_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

Sz_3	
0	Byte
1	Char
2	Half
3	Word

CNTPOP – Count Population

Description:

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

Instruction Format:

01_6	Sz_3	2_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

Sz_3	
0	Byte
1	Char
2	Half
3	Word

CSR – Control and Status Access

Description:

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

Instruction Format:

Op$_2$	OL$_2$	Regno$_{10}$	Rt$_5$	Ra$_5$	0$_2$	05h$_6$

Op$_2$		Operation
0	CSRRD	Only read the CSR, no update takes place, Ra should be R0.
1	CSRRW	Both read and write the CSR
2	CSRRS	Read CSR then set CSR bits
3	CSRRC	Read CSR then clear CSR bits

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

The OL$_2$ field is reserved to specify the operating level. Note that registers cannot be accessed by a lower operating level.

Regno$_{10}$		Access	Description
001	HARTID	R	hardware thread identifier (core number)
002	TICK	R	tick count, counts every cycle from reset
030-037	TVEC	RW	trap vector handler address
040	EPC	RW	exceptioned pc, pc value at point of exception
044	STATUSL	RWSC	status register, contains interrupt mask, operating level
045	STATUSH	RW	status register bits 64 to 127
080-0BF	CODE	RW	code buffers
3F0	INFO	R	Manufacturer name
3F1	“	R	“
3F2	“	R	cpu class
3F3	“	R	“
3F4	“	R	cpu name
3F5	“	R	“
3F6	“	R	model number
3F7	“	R	serial number
3F8	“	R	cache sizes instruction (bits 32 to 63), data (bits 0 to 31)

Clock Cycles: 0.5

DIV – Signed Division

Description:

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

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	3Eh$_6$

Return quotient

3Eh$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

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

Execution Units: ALU #0 Only

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

DIVSU – Signed-Unsigned Division

Description:

Compute the quotient value. Both operands must be in registers. The first operand is treated as a signed value. The second operand is an unsigned value. The result is a signed result.

Instruction Format:

Return quotient

3Dh$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

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

Execution Units: ALU #0 Only

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

DIVU – Unsigned Division

Description:

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

Comment:

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

Instruction Format:

Immed$_{14}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	3Ch$_{6}$

Return quotient

3Ch$_{6}$	Sz$_{3}$	Rb$_{5}$	Rt$_{5}$	Ra$_{5}$	0$_{2}$	02h$_{6}$

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

Execution Units: ALU #0 Only

Exceptions: none

EXEC – Execute Code Buffer

Description:

Execute code from code buffer. The N_6 field specifies the code buffer to use. Code buffers allow code to be adapted at run-time. This is useful as an alternative to self-modifying code when code has to change at runtime.

Instruction Format:

01_6	$\sim_3$	$13h_5$	$\sim_4$	N_6	0_2	$02h_6$

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

FXADD – Fixed Point Addition

Description:

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

Instruction Format:

04_6	Sz_3	Rt_5	Rb_5	Ra_5	0_2	$02h_6$

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions: none

FXMUL – Fixed Point Multiply

Description:

This instruction multiplies Ra by Rb. Ra and Rb are fixed point numbers with whole and binary point places (32.32) or (16.16). The result is a fixed-point number.

Instruction Format:

3Bh$_6$	Sz$_3$	Rt$_5$	Rb$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 19

ALU Support: All

FXSUB – Fixed Point Subtraction

Description:

Subtract two values assuming operands are fixed point numbers. Both operands must be in a register. This instruction is an alternate mnemonic for the SUB instruction.

Instruction Format:

05_6	Sz_3	Rt_5	Rb_5	Ra_5	0_2	$02h_6$

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions: none

INC – Increment Word (64 bits)

Description:

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

Instruction Format:

Immed$_{14}$	Amt$_{5}$	Ra$_{5}$	L$_{2}$	1Ah$_{6}$

1Ah$_{6}$	~	Sc$_{2}$	Amt$_{5}$	Rb$_{5}$	Ra$_{5}$	L$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

ISNULL – Is Null Pointer

Description:

This instruction detects if the value in a register is a null pointer and places the result in a target register. A null pointer value is indicated if the top 20 bits of the value are equal to $FFF01 and the remaining bits are all zero OR if all bits of the register are zero.

Instruction Format:

01_6	0_3	$6h_5$	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
If Ra = $FFF0100000000000 or Ra = 0
   Rt = 1
else
   Rt = 0
```

Exceptions: none

Notes:

ISPTR – Is Pointer

Description:

This instruction detects if the value in a register is a pointer and places the result in a target register. A pointer value is indicated if the top 20 bits of the value are equal to $FFF01.

Instruction Format:

01_6	1_3	6_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
If $Ra_{[63..44]}$ = $FFF01
   Rt = 1
else
   Rt = 0
```

Exceptions: none

Notes:

JAL – Jump-And-Link

Description:

Instruction Format:

This instruction loads the program counter with the sum of a register and a constant value specified in the instruction. In addition, the address of the instruction following the JALR is stored in the specified target register. This instruction may be used to implement subroutine calls and returns.

Immed$_{14}$	Rt$_5$	Ra$_5$	0$_2$	18h$_6$

Instruction Format:

This instruction loads the program counter with the sum of a register and a constant value specified in the instruction. The code segment is set to the specified segment register. In addition, the address of the instruction following the JALR is stored in the specified target register. This instruction may be used to implement far subroutine calls.

Seg$_2$	Immed$_{28}$	Rt$_5$	Ra$_5$	1$_2$	18h$_6$

Seg$_2$	
0	ZS
1	ES
2	HS
3	CS

Compressed Instruction Format:

For the compressed format, the constant value is zero. The address of a frequently used subroutine call may be loaded into a register as a compiler optimization, then the compressed JAL instruction used.

2$_4$	Rt$_{4..1}$	3$_2$	Rt$_0$	Ra$_{4..0}$

Execution Units: FCU

Clock Cycles:

JMP – Jump to Address

Description:

A jump is made to the address specified in the instruction. The jump may be within the current code segment or to another code segment.

Instruction Format:

The format modifies only PC bits 0 to 23. The high order PC bits are not affected. This allows accessing code within a 16MB region of memory. Note that with the use of a mmu this address range is often sufficient.

Immed$_{24}$	L$_2$	28h$_6$

Instruction Format:

The format modifies only PC bits 0 to 37. The high order PC bits are not affected. This allows accessing code within a 256GB region of memory. This format allows jumping to a different code segment.

Seg$_2$	Immed$_{38}$	1$_2$	28h$_6$

Seg$_2$	
0	ZS
1	ES
2	HS
3	CS

Execution Units: FCU

Clock Cycles: 1

Exceptions: none

Notes:

If an address range larger than 37 bits is required then the value must be loaded into a register and the JAL instruction used.

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

LB – Load Byte

Description:

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

Instruction Formats:

Immed$_{14}$	Rt$_5$	Ra$_5$	0$_2$	13h$_6$

Immed$_{30}$	Rt$_5$	Ra$_5$	1$_2$	13h$_6$

4h$_4$	Rc$_5$	3$_2$	~	Sc$_2$	Rt$_5$	Ra$_5$	0$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

LBU – Load Unsigned Byte

Description:

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

Instruction Formats:

Immed$_{14}$	Rt$_5$	Ra$_5$	0$_2$	23h$_6$

Immed$_{30}$	Rt$_5$	Ra$_5$	1$_2$	23h$_6$

2h$_4$	Rc$_5$	2$_2$	~	Sc$_2$	Rt$_5$	Ra$_5$	0$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

LC – Load Char (16 bits)

Description:

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

Instruction Format:

Immed$_{13}$	1	Rt$_5$	Ra$_5$	0$_2$	20h$_6$

Immed$_{29}$	1	Rt$_5$	Ra$_5$	1$_2$	20h$_6$

2h$_4$	Rc$_5$	0$_2$	~	Sc$_2$	Rt$_5$	Ra$_5$	0$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

LCU – Load Unsigned Char (16 bits)

Description:

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

For the short (32 bit) forms of the instruction the processor will use the stack segment for references involving the stack or frame pointer, otherwise the data segment will be used.

For the long (48 bit) forms of the instruction the segment used is specified in the instruction. The assembler will default this field to the stack segment for stack pointer or frame pointer references, or the data segment otherwise. The default assignment may be overridden with a segment prefix indicator.

Instruction Format:

Immed$_{13}$	1	Rt$_5$	Ra$_5$	0$_2$	21h$_6$

Immed$_{29}$	1	Rt$_5$	Ra$_5$	1$_2$	21h$_6$

2h$_4$	Rc$_5$	1$_2$	~	Sc$_2$	Rt$_5$	Ra$_5$	0$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

LD – Load Double-Word (128 bits)

Description:

This instruction loads a word (128 bit) value from memory. The memory address must be double-word aligned.

Instruction Format:

Immed_{10}	8_4	Rt_5	Ra_5	L_2	20h_6

Clock Cycles: 4 minimum depending on memory access time

LDI – Load Immediate

Description:

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

Instruction Format:

Immed_{14}	Rt_5	0_5	L_2	$08h_6$

L_2: 0 = 14 bit constant, 1 = 30 bit constant

Clock Cycles: 0.5

LEA – Load Effective Address

Description:

This instruction loads an address value into a register. The upper 20 bits of the register are automatically set to $FFF01 to indicate a pointer value is in the register.

Instruction Format:.

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	0Eh$_6$

L: 0 = 14 bit constant, 1 = 30 bit constant

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

6h$_4$	Rc$_5$	0$_2$	~	Sc$_2$	Rt$_5$	Ra$_5$	L$_2$	16h$_6$

Clock Cycles: 0.5

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

LFD – Load Float Double

Description:

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

Instruction Format:

Immed$_{11}$	4$_{3}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	1Bh$_{6}$

7h$_{4}$	Rc$_{5}$	1$_{2}$	~	Sc$_{2}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

LFQ – Load Float Quad

Description:

This instruction stores a word (128 bit) value to memory. The memory address must be double-word aligned.

Instruction Format:

Immed$_{10}$	8$_4$	Rt$_5$	Ra$_5$	L$_2$	1Bh$_6$

7h$_4$	Rc$_5$	3$_2$	~	Sc$_2$	Rt$_5$	Ra$_5$	L$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

LFS – Load Float Single

Description:

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

Instruction Format:

Immed$_{12}$	2_2	Rt$_5$	Ra$_5$	L$_2$	1Bh$_6$

7h$_4$	Rc$_5$	0_2	~	Sc$_2$	Rt$_5$	Ra$_5$	L$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

LH – Load Half-Word (32 bits)

Description:

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

Instruction Format:

Immed$_{12}$	2$_{2}$	Rt$_{5}$	Ra$_{5}$	0$_{2}$	20h$_{6}$

Immed$_{28}$	2$_{2}$	Rt$_{5}$	Ra$_{5}$	1$_{2}$	20h$_{6}$

4h$_{4}$	Rc$_{5}$	0$_{2}$	~	Sc$_{2}$	Rt$_{5}$	Ra$_{5}$	0$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

LHU – Load Half-Word (32 bits)

Description:

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

Instruction Format:

Immed$_{12}$	2$_{2}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	21h$_{6}$

4h$_{4}$	Rc$_{5}$	1$_{2}$	~	Sc$_{2}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

LUI – Load Upper Immediate

Description:

This instruction loads an immediate value shifted left 30 times into a target register. The low order 30 bits of the target register are zeroed out. When long forms of immediate are used, a full 64-bit immediate may be formed.

Instruction Format:

Immed$_{18..5}$	Rt$_5$	Imm$_{4..0}$	L$_2$	27h$_6$

L$_2$: 0 = 19 bit constant, 1 = 35 bit constant

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions:

Notes:

LVB – Load Volatile Byte (8 bits)

Description:

This instruction loads a byte (8 bit) value from memory. This load instruction bypasses the data cache and loads directly from memory. There is only an indexed form of this instruction.

Instruction Format:

$0h_4$	Rc_5	0_2	~	Sc_2	Rt_5	Ra_5	L_2	$16h_6$

Clock Cycles: 4 minimum depending on memory access time

Sc_2	Scale Rb By
0	1
1	2
2	4
3	8

LVBU – Load Volatile Unsigned Byte (8 bits)

Description:

This instruction loads a byte (8 bit) value from memory. This load instruction bypasses the data cache and loads directly from memory. There is only an indexed form of this instruction.

Instruction Format:

$0h_4$	Rc_5	1_2	~	Sc_2	Rt_5	Ra_5	L_2	$16h_6$

Clock Cycles: 4 minimum depending on memory access time

Sc_2	Scale Rb By
0	1
1	2
2	4
3	8

LVC – Load Volatile Char (16 bits)

Description:

This instruction loads a char (16 bit) value from memory. This load instruction bypasses the data cache and loads directly from memory.

Instruction Format:

Immed$_{13}$	1	Rt$_5$	Ra$_5$	L$_2$	3Bh$_6$

Instruction Format:

0h$_4$	Rc$_5$	2$_2$	~	Sc$_2$	Rt$_5$	Ra$_5$	L$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

LVCU – Load Volatile Unsigned Char (16 bits)

Description:

This instruction loads a char (16 bit) value from memory. This load instruction bypasses the data cache and loads directly from memory. The value is zero extended to 64-bits in the register.

Instruction Format:

Immed$_{13}$	1	Rt$_{5}$	Ra$_{5}$	L$_{2}$	11h$_{6}$

Instruction Format:

0h$_{4}$	Rc$_{5}$	3$_{2}$	~	Sc$_{2}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

LVH – Load Volatile Half-word (32 bits)

Description:

This instruction loads a char (32 bit) value from memory. This load instruction bypasses the data cache and loads directly from memory. The value is sign extended to 64-bits in the register.

Instruction Format:

Immed$_{12}$	2$_{2}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	3Bh$_{6}$

Instruction Format:

1h$_{4}$	Rc$_{5}$	0$_{2}$	~	Sc$_{2}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

LVHU – Load Volatile Unsigned Half-word (32 bits)

Description:

This instruction loads a half-word (32 bit) value from memory. This load instruction bypasses the data cache and loads directly from memory. The value is zero extended to 64-bits in the register.

Instruction Format:

Immed_{12}	2_2	Rt_5	Ra_5	L_2	11h_6

Instruction Format:

1h_4	Rc_5	1_2	~	Sc_2	Rt_5	Ra_5	L_2	16h_6

Clock Cycles: 4 minimum depending on memory access time

Sc_2	Scale Rb By
0	1
1	2
2	4
3	8

LVW – Load Volatile Word (64 bits)

Description:

This instruction loads a word (64 bit) value from memory. The memory address must be word aligned. This load instruction bypasses the data cache and loads directly from memory.

Instruction Format:

Immed$_{11}$	4$_{3}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	3Bh$_{6}$

Instruction Format:

1h$_{4}$	Rc$_{5}$	2$_{2}$	~	Sc$_{2}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

LW – Load Word (64 bits)

Description:

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

Instruction Format:

Immed$_{14}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	33h$_{6}$

4h$_{4}$	Rc$_{5}$	2$_{2}$	~	Sc$_{2}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

LWR – Load Word and Reserve Address

Description:

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

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	1Dh$_6$

5h$_4$	Rc$_5$	0$_2$	~	Sc$_2$	Rt$_5$	Ra$_5$	L$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

Acquire and release bits determine the ordering of memory operations.

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

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

All combinations of A, R are allowed.

MAJ – Majority Logic

Description:

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

Instruction Format:

2Eh$_6$	~$_{14}$	Rc$_5$	Rb$_5$	Rt$_5$	Ra$_5$	1$_2$	02h$_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

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

MAX – Maximum Value

Description:

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

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

Instruction Format:

2Dh$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Maximum of three values:

2Dh$_6$	~$_{11}$	Sz$_3$	Rc$_5$	Rb$_5$	Rt$_5$	Ra$_5$	1$_2$	02h$_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
IF Ra > Rb
        Rt = Ra
else
        Rt = Rb
```

MEMDB –Memory Data Barrier

Description:

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

Instruction Format:

$01h_6$	$\sim_5$	$10h_5$	$\sim_5$	$\sim_5$	$02h_6$

Clock Cycles: varies depending on queue contents

MEMSB –Memory Synchronization Barrier

Description:

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

Instruction Format:

$01h_6$	$\sim_5$	$11h_5$	$\sim_5$	$\sim_5$	$02h_6$

Clock Cycles: varies depending on queue contents

MIN – Minimum Value

Description:

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

Instruction Format:

2Ch$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Minimum of three values:

2Ch$_6$	~$_{11}$	Sz$_3$	Rc$_5$	Rb$_5$	Rt$_5$	Ra$_5$	1$_2$	02h$_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

```
IF Ra < Rb
    Rt = Ra
else
    Rt = Rb
```

MOD – Signed Modulus

Description:

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

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	2Eh$_6$

Return remainder

16h$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

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

Execution Units: ALU #0 Only

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

MODSU – Signed-Unsigned Modulus

Description:

Compute the modulus (remainder) value. Both operands must be in registers. The first operand is treated as a signed value. The second operand is an unsigned value. The result is a signed result. There is no immediate form for this instruction.

Instruction Format:

Return remainder

$15h_6$	Sz_3	Rb_5	Rt_5	Ra_5	0_2	$02h_6$

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

ALU Support: ALU #0 Only

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

MODU – Unsigned Modulus

Description:

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

Instruction Format:

Return remainder

$14h_6$	Sz_3	Rb_5	Rt_5	Ra_5	0_2	$02h_6$

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

ALU Support: ALU #0 Only

Exceptions: none

MOV – Move register to register

Description:

This instruction moves one general purpose register to another including between different register sets. This instruction may be used to move between the integer and floating-point registers or between normal and excepted register sets.

Note that one does not normally want a value moved directly between integer and floating-point registers. Instead usually a conversion is desired (ftoi - double to integer or itof - integer to double) for example.

Instruction Format:

Register sets 0 to 31

$22h_6$	D_3	Rgs_5	Rt_5	Ra_5	0_2	$02h_6$

Register sets 32 to 63

$23h_6$	D_3	Rgs_5	Rt_5	Ra_5	0_2	$02h_6$

Compressed Instruction Format:

For the compressed format, moves take place only for general purpose registers in the current register set.

0_4	$Rt_{4..1}$	3_2	Rt_0	$Ra_{4..0}$

D_3	Asm Sample	Operation
0	mov r6:1,r1	move from current Ra to Rt in register set Rgs
1	mov r1,r6:1	move from Ra in register set Rgs to Rt in current register set
2	mov r7:x,r2	move from current Ra to Rt in excepted register set (Rgs is ignored).
3	mov r7,r2:x	move from Ra in excepted register to Rt in current register set.
4	mov fp8,r3	move from Ra in current register set to Rt in floating point register set
5	mov r3,fp9	move from floating point to general register file in current register set
6	mov fp1,fp2	move from current floating-point to current floating-point register
7	mov r15,r23	move from current Ra to current Rt (rgs ignored).

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions: none

Notes:

The exceptioned register set referred to by the instruction is the one identified by the top stack element of the rs_stack.

MUL – Signed Multiply

Description:

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

Instruction Format:

Immed$_{14}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	3Ah$_{6}$

Multiply, return low order product

3Ah$_{6}$	Sz$_{3}$	Rb$_{5}$	Rt$_{5}$	Ra$_{5}$	0$_{2}$	02h$_{6}$

Clock Cycles: 19

Exceptions: multiply overflow, if enabled

MULF – Fast Unsigned Multiply

Description:

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

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	2Ah$_6$

2Ah$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	L$_2$	02h$_6$

Clock Cycles: 0.5

Exceptions: none

MULH – Signed Multiply

Description:

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

Instruction Format:

Multiply, return high order product

26h$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 19

MULSU – Signed-Unsigned Multiply

Description:

Multiply two values. Both operands must be in registers. The first operand is treated as a signed value. The second operand is treated as an unsigned value. The result is a signed result. There is no immediate form for this instruction.

Instruction Format:

39h$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 19

Exceptions: none

MULSUH – Signed-Unsigned Multiply

Description:

Multiply two values. Both operands must be in registers. The first operand is treated as a signed value. The second operand is treated as an unsigned value. The result is a signed result. There is no immediate form for this instruction. The high order bits of the product are returned.

Instruction Format:

Multiply, return high order product

25h$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 19

Exceptions: none

MULU – Unsigned Multiply

Description:

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

Comment:

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

Instruction Format:

Immed$_{14}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	38h$_{6}$

Multiply, return low order product

38h$_{6}$	Sz$_{3}$	Rb$_{5}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	02h$_{6}$

Clock Cycles: 20

Exceptions: none

MULUH – Unsigned Multiply

Description:

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

Instruction Format:

Multiply, return high order product

$24h_6$	Sz_3	Rb_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 20

Exceptions: none

MUX – Multiplex

Description:

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

Instruction Format:

$1Bh_6$	$\sim_{14}$	Rc_5	Rb_5	Rt_5	Ra_5	1_2	$02h_6$

Clock Cycles: 0.5

Exceptions: none

NAND – Bitwise Nand

Description:

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

Instruction Format:

0Ch$_6$	~$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 0.5

Execution Units: All ALUs

Exceptions: none

NEG - Negate

Description:

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

Instruction Format:

05_6	Sz_3	Rb_5	Rt_5	0_5	0_2	$02h_6$

Clock Cycles: 0.5

NOP – No Operation

Description:

The NOP instruction doesn't perform any operation. NOP's are detected in the instruction fetch stage of the core and are not enqueued by the core. They do not occupy queue slots. Because NOPs don't occupy queue slots they may not be used to synchronize operations between instructions. Note that a compressed NOP is really an add instruction and hence occupies space in the instruction queue.

Instruction Format:

Immediate$_{26}$	L$_{2}$	3Dh$_{6}$

Compressed Instruction Format:

00h$_{8}$	80h$_{2}$

Clock Cycles: 0.5

Execution Units: trapped at IF stage

Exceptions: none

NOR – Bitwise Nor

Description:

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

Instruction Format:

0Dh$_6$	~$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 0.5

Execution Units: All ALUs

Exceptions: none

NOT – Logical Not

Description:

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

Instruction Format:

01_6	Sz_3	5_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = !Ra

Exceptions: none

Notes:

Sz_3	
0	reserved
1	reserved
2	reserved
3	reserved
4	Byte Parallel
5	Char Parallel
6	Half Parallel
7	Word

OR – Bitwise Or

Description:

Perform a bitwise or operation between operands.

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	09h$_6$

L$_2$: 0 = 14 bit constant, 1 = 30 bit constant

09$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Triple Operand

09$_6$	~$_3$	Sz$_3$	Rt$_5$	Rc$_5$	Rb$_5$	Ra$_5$	1$_2$	02h$_6$

Clock Cycles: 0.5

Execution Units: All ALUs

Exceptions: none

PFI – Poll for Interrupt

Description:

The poll for interrupt instruction polls the interrupt status lines and performs an interrupt service if an interrupt is present. Otherwise the PFI instruction is treated as a NOP operation. The PFI instruction has a 16-bit compressed instruction format. Polling for interrupts is performed by managed code. PFI provides a means to process interrupts at specific points in running software.

Instruction Format:

31 26	25 21	20 17	16	15 8	7 6	5 0
User$_6$	2_5	0_4	0	255_8	L_2	00h$_6$

Compressed Instruction Format:

The compressed instruction format of the PFI instruction borrows one opcode from the compressed format of the BRK instruction. Cause code 63 is reserved for the interrupt polling instruction (PFI). The return address is the address of the instruction following the PFI. (Skip is implied = 1).

15 12	11 8	7 6	5	4 0
01h$_4$	Fh$_4$	10_2	1_0	00h$_5$

PTRDIF – Difference Between Pointers

Description:

Subtract two values then shift the result right. Both operands must be in a register. The top 20 bits of each value are masked off before the subtract. The right shift is provided to accommodate common object sizes. It may still be necessary to perform a divide operation after the PTRDIF to obtain an index into odd sized or large objects.

Instruction Format:

1Eh$_6$	Sc$_3$	Rb$_5$	Rt$_5$	Ra$_5$	L$_2$	02h$_6$

Operation:

Rt = (Ra – Rb) >> Sc

Clock Cycles: 0.5

Exceptions:

None.

PUSH – Push Word (64 bits)

Description:

This instruction decrements the stack pointer and stores a word (64 bit) value to stack memory. The value pushed onto the stack may come from either a register or a constant value defined in the instruction.

Instruction Format:

Ch$_4$	~$_5$	Rb$_5$	1Fh$_5$	1Fh$_5$	L$_2$	16h$_6$

Constant$_{14}$			1Fh$_5$	IFh$_5$	L$_2$	14h$_6$

Compressed Instruction Format (register only):

15 12	11 8	7 6	5	4 0
03h$_4$	0h$_4$	11$_2$	0$_0$	Rb$_5$

Operation:

Memory$_8$[SP - 8] = Rb
SP = SP – 8

Clock Cycles: 4 minimum depending on memory access time

Notes:

Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.

The instruction explicitly encodes the stack pointer register r31, if desired a different register may be chosen. For the compressed format, the stack pointer is always r31.

~~REDOR – Reduction Or~~

Description:

This instruction turns a non-zero value in a register into a single bit Boolean one. If the register is zero, the target is set to zero, otherwise the target is set to one.

Instruction Format:

01_6	Sz_3	6_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: All ALU’s

Operation:

```
If Ra <> 0
   Rt = 1
else
   Rt = 0
```

Exceptions: none

Notes:

Sz_3	
0	reserved
1	reserved
2	reserved
3	reserved
4	Byte Parallel
5	Char Parallel
6	Half Parallel
7	Word

RET – Return from Subroutine

Description:

This instruction performs a subroutine return by loading the program counter with the contents of the return address register (r29). Additionally, the stack pointer is adjusted by a constant supplied in the instruction. The immediate constant is a multiple of eight to keep the stack word aligned. The constant is also zero extended to the left.

Instruction Format:

Immed$_9$	IDh$_5$	1Fh$_5$	1Fh$_5$	L$_2$	29h$_6$

PC = RA
SP = SP + Immediate * 8

Clock Cycles: 1 (more if predicted incorrectly).

Exceptions: none

Notes:

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

REX – Redirect Exception

Description:

This instruction redirects an exception from an operating level to a lower operating level and privilege level. If the target operating level is hypervisor then the hypervisor privilege level (1) is set. If the target operating level is supervisor, then one of the supervisor privilege levels must be chosen (2 to 6). This instruction if successful jumps to the target exception handler and does not return. If this instruction fails execution will continue with the next instruction.

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

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

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

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

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

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

Instruction Format:

31 29	28 26	25 18	1715	14 13	12 8	76	5 0
$\sim_3$	IM_3	PL_8	$\sim_3$	Tgt_2	Ra_5	0_2	$0Dh_6$

Tgt_2	
0	not used
1	redirect to hypervisor level
2	redirect to supervisor level
3	not used

Clock Cycles: 3

Example:

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

Notes:

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

ROL – Rotate Left

Description:

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

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

Instruction Format:

2Fh$_6$	4$_3$	Rt$_5$	Rb$_5$	Ra$_5$	0$_2$	02h$_6$

Immediate, shift count 0 to 31

0Fh$_6$	4$_3$	Imm$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Immediate, shift count 32 to 63

1Fh$_6$	4$_3$	Imm$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

ROR – Rotate Right

Description:

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

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

Instruction Format:

$2Fh_6$	5_3	Rt_5	Rb_5	Ra_5	0_2	$02h_6$

Immediate, shift count 0 to 31

$0Fh_6$	5_3	Imm_5	Rt_5	Ra_5	0_2	$02h_6$

Immediate, shift count 32 to 63

$1Fh_6$	5_3	Imm_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

RTI – Return from Interrupt

Description:

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

Instruction Format:

32h$_6$	~$_2$	Sema$_6$	~$_5$	Ra$_5$	0$_2$	02h$_6$

Semaphore[Sema$_6$|[Ra]] = 0
Semaphore[0] = 0

Clock Cycles: 8 minimum

Execution Units: Flow Control Unit

RTE – Return from Exception

Description:

This is an alternate mnemonic for the RTI instruction.

Instruction Format:

$32h_6$	$\sim_2$	$Sema_6$	$\sim_5$	Ra_5	0_2	$02h_6$

Semaphore[$Sema_6$|[Ra]] = 0
Semaphore[0] = 0

Clock Cycles: 8 minimum

Execution Units: Flow Control Unit

RTOP – Runtime Operation

Description:

Perform an operation that is determined at run-time by the contents of a register. The contents of register Rc determine the instruction passed to the ALU. The register / constant fields of the instruction will be ignored. Instead operands will come from registers Ra and Rb.

Instruction Format:

$00h_6$	$\sim_6$	$\sim_5$	Sz_3	Rt_5	Rc_5	Rb_5	Ra_5	1_2	$02h_6$

Clock Cycles: 0.5

Execution Units: All ALUs

Exceptions: none

SB – Store Byte (8 bits)

Description:

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

Instruction Format:

Immed$_{13...5}$	Rb$_5$	Imm$_{4..0}$	Ra$_5$	0$_2$	15h$_6$

Immed$_{29...5}$	Rb$_5$	Imm$_{4..0}$	Ra$_5$	1$_2$	15h$_6$

8h$_4$	Rc$_5$	Rb$_5$	0$_2$	~	Sc$_2$	Ra$_5$	0$_2$	16h$_6$

Operation:

Memory$_8$[Ra + immediate] = Rb

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

Notes:

Stores always write through to memory and therefore take a significant number of clock cycles before they are ready to be committed. Exceptions are checked for during the execution of a store operation.

SC – Store Char (16 bits)

Description:

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

Instruction Format:

Immed$_{13…5}$	Rb$_5$	Im$_{4..0}$	Ra$_5$	L$_2$	24h$_6$

9h$_4$	Rc$_5$	Rb$_5$	0$_2$	~	Sc$_2$	Ra$_5$	L$_2$	16h$_6$

Operation:

Memory$_{16}$[Ra + immediate] = Rb

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

SEI – Set Interrupt Mask

SEI #3
SEI $v0,#7

Description:

The interrupt level mask is set to the value specified by the instruction. The value used is the bitwise or of the contents of register Ra and an immediate (M_4) supplied in the instruction. The assembler assumes a mask value of fifteen, masking all interrupts, if no mask value is specified. Usually either M_4 or Ra should be zero. The previous setting of the interrupt mask is stored in Rt.

Instruction Format:

$30h_6$	$\sim_2$	$\sim_2$	M_4	Rt_5	Ra_5	0_2	$02h_6$

Operation:

Rt = im
im = M_3 \| Ra

SETWB -Set Write Barrier

Description:

This instruction sets the write barrier indicator for the current register set in the WBRCD CSR. This instruction is output by the compiler whenever a register is loaded with a pointer value. Rn_5 identifies the register containing a pointer.

Instruction Format:

$01h_6$	0_3	$16h_5$	$\sim_5$	Rn_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: All ALU’s

Notes:

SFD – Store Float Double

Description:

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

Instruction Format:

Immed$_{11}$	4_3	Rb$_5$	Ra$_5$	L$_2$	2Bh$_6$

Bh$_4$	Rc$_5$	Rb$_5$	1_2	~	Sc$_2$	Ra$_5$	L$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

SFQ – Store Float Quad

Description:

This instruction stores a quad precision floating-point (128 bit) value to memory. The memory address must be double-word aligned.

Instruction Format:

Immed$_{10}$	8$_{4}$	Rb$_{5}$	Ra$_{5}$	L$_{2}$	2Bh$_{6}$

Bh$_{4}$	Rc$_{5}$	Rb$_{5}$	3$_{2}$	~	Sc$_{2}$	Ra$_{5}$	L$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

SFS – Store Float Single

Description:

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

Instruction Format:

Immed$_{12}$	2$_{2}$	Rb$_{5}$	Ra$_{5}$	L$_{2}$	2Bh$_{6}$

Bh$_{4}$	Rc$_{5}$	Rb$_{5}$	0$_{2}$	~	Sc$_{2}$	Ra$_{5}$	L$_{2}$	16h$_{6}$

Clock Cycles: 4 minimum depending on memory access time

Sc$_{2}$	Scale Rb By
0	1
1	2
2	4
3	8

SGN – Get Sign

Description:

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

Instruction Format:

06_6	0_3	0_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 0.5

SH – Store Half-Word (32 bits)

Description:

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

Instruction Format:

Immed$_{13…5}$	Rb$_5$	Im$_{4..0}$	Ra$_5$	L$_2$	3Bh$_6$

8h$_4$	Rc$_5$	Rb$_5$	1$_2$	~	Sc$_2$	Ra$_5$	L$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

SHL – Shift Left

Description:

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

Instruction Format:

2Fh$_6$	0$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Immediate, shift count 0 to 31

0Fh$_6$	0$_3$	Imm$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Immediate, shift count 32 to 63

1Fh$_6$	0$_3$	Imm$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Compressed Instruction Format:

3$_4$	Imm$_{[4..1]}$	2$_2$	Imm$_{[0]}$	Rt$_5$

Clock Cycles: 0.5

ALU Support: All

Exceptions: none

SHL:[op] – Shift Left and Op

SHL:ADD
SHL:OR

Description:

> Bits from the source register Ra are shifted left by the amount in register Rb or an immediate value. Zeros are shifted into the least significant bits. A second operation is performed between the first result and register Rc. The second operation must be one of: add, sub, and, or, or xor.

Double Op Instruction Format:

47 42	41 36	3533	32 30	29	28	27 23	22 18	17 13	12 8	76	5 0
2Fh$_6$	Op2$_6$	0$_3$	Sz$_3$	R	I	Rc$_5$	Rb$_5$	Rt$_5$	Ra$_5$	1$_2$	02h$_6$

R: determines if Rb is a register (0) or an immediate value (1)
I: the high order bit of the immediate value

Clock Cycles: 2

ALU Support: ALU #0 Only

Exceptions: none

SHR – Shift Right

Description:

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

Instruction Format:

2Fh$_6$	1$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Immediate, shift count 0 to 31

0Fh$_6$	1$_3$	Imm$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Immediate, shift count 32 to 63

1Fh$_6$	1$_3$	Imm$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

SHR:[op] – Shift Right and Op

SHR:ADD
SHR:OR

Description:

> Bits from the source register Ra are shifted left by the amount in register Rb or an immediate value. Zeros are shifted into the least significant bits. A second operation is performed between the first result and register Rc. The second operation must be one of: add, sub, and, or, or xor.

Double Op Instruction Format:

47 42	41 36	3533	32 30	29	28	27 23	22 18	17 13	12 8	76	5 0
2Fh$_6$	Op2$_6$	1$_3$	Sz$_3$	R	I	Rt$_5$	Rc$_5$	Rb$_5$	Ra$_5$	1$_2$	02h$_6$

Clock Cycles: 2

ALU Support: ALU #0 Only

Exceptions: none

SGT – Set if Greater Than

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than a second operand in either a register (Rb) or a constant supplied in the instruction, then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as signed values.

The register form of the instruction is just the register form of the SLT instruction with the operands switched.

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	2Ch$_6$

06h$_6$	Sz$_3$	Ra$_5$	Rt$_5$	Rb$_5$	0$_2$	02h$_6$

Clock Cycles: 0.5

SGTU – Set if Greater Than Unsigned

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is greater than a second operand in either a register (Rb) or a constant supplied in the instruction, then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as unsigned values.

The register form of the instruction is just the register form of the SLTU instruction with the operands switched.

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	1Ch$_6$

07h$_6$	Sz$_3$	Ra$_5$	Rt$_5$	Rb$_5$	0$_2$	02h$_6$

Clock Cycles: 0.5

SLE – Set if Less Than or Equal

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than or equal to a second operand in a register (Rb), then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as signed values.

The instruction may also be used to test for greater than or equal by swapping the operands around.

There is no immediate form for this instruction.

Instruction Format:

$28h_6$	Sz_3	Rb_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 0.5

SLEU – Set if Less Than or Equal Unsigned

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than or equal to a second operand in a register (Rb), then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as unsigned values.

The instruction may also be used to test for greater than or equal by swapping the operands around.

There is no immediate form for this instruction.

Instruction Format:

$29h_6$	Sz_3	Rb_5	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 0.5

SLT – Set if Less Than

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than a second operand in either a register (Rb) or a constant supplied in the instruction, then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as signed values.

The register form of the instruction may also be used to test for greater than by swapping the operands around.

Instruction Format:

Immed$_{14}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	06h$_{6}$

06h$_{6}$	Sz$_{3}$	Rb$_{5}$	Rt$_{5}$	Ra$_{5}$	0$_{2}$	02h$_{6}$

Clock Cycles: 0.5

SLTU – Set if Less Than Unsigned

Description:

The set instruction places a 1 or 0 in the target register based on the relationship between the two source operands. If operand Ra is less than a second operand in either a register (Rb) or a constant supplied in the instruction, then the target register is set to a one, otherwise the target register is set to a zero. The operands are treated as unsigned values.

Instruction Format:

Immed$_{14}$	Rt$_{5}$	Ra$_{5}$	L$_{2}$	07h$_{6}$

07h$_{6}$	Sz$_{3}$	Rb$_{5}$	Rt$_{5}$	Ra$_{5}$	0$_{2}$	02h$_{6}$

Clock Cycles: 0.5

SUB - Subtract

Description:

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

Instruction Format:

05_6	Sz_3	Rb_5	Rt_5	Ra_5	L_2	$02h_6$

Clock Cycles: 0.5

Exceptions:

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

SUBV – Subtraction with Overflow Detection

Description:

Subtract two values. Both operands must be in registers. If an overflow occurs an overflow exception may be generated if enabled in the arithmetic exception control register.

Instruction Format:

0Bh$_6$	~$_3$	Rt$_5$	Rb$_5$	Ra$_5$	L$_2$	02h$_6$

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions:

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

Notes:

SW – Store Word (64 bits)

Description:

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

Instruction Format:

Immed$_{13\ldots5}$	Rb$_5$	I$_{4..0}$	Ra$_5$	L$_2$	35h$_6$

8h$_4$	Rc$_5$	Rb$_5$	2$_2$	~	Sc$_2$	Ra$_5$	L$_2$	16h$_6$

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

SWC – Store Word and Clear Reservation

Description:

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

Instruction Format:

Immed$_{13…5}$	Rb$_5$	Imm$_{4..0}$	Ra$_5$	L$_2$	17h$_6$

8h$_4$	Rc$_5$	Rb$_5$	3$_2$	~	Sc$_2$	Ra$_5$	L$_2$	16h$_6$

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

Clock Cycles: 4 minimum depending on memory access time

Sc$_2$	Scale Rb By
0	1
1	2
2	4
3	8

Acquire and release bits determine the ordering of memory operations.

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

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

All combinations of A, R are allowed.

SYNC -Synchronize

Description:

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

Instruction Format:

$01h_6$	0_3	$12h_5$	$\sim_5$	$\sim_5$	0_2	$02h_6$

Clock Cycles: 1 *varies depending on queue contents

Execution Units: All ALU’s

Notes:

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

SXH – Sign Extend Half-Word

Description:

Sign extend a half word as an signed value to the full width of the machine.

Instruction Format:

01_6	$\sim_3$	$18h_5$	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

THRD – Conditional Thread Switch

Description:

> This instruction performs a subroutine call to the thread switch routine or to the garbage collection routine if one of the signals for those routines is active. The garbage collection routine address is stored in the GCA CSR register. The thread switch routine address is stored in the TSA CSR register. The THRD instruction should be placed at thread-safe points.

Instruction Format:

Execution Units: FCU

Clock Cycles:

TLB – TLB Command

Description:

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

This instruction is only available at the machine operating level.

Instruction Format:

3Fh$_6$	Cmd$_4$	Tn$_4$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 3

Tn$_4$ – This field identifies which TLB register is being read or written.

Reg no.		Assembler
0	Wired	Wired
1	Index	Index
2	Random	Random
3	Page Size	PageSize
4	Virtual page	VirtPage
5	Physical page	PhysPage
7	ASID	ASID
8	Miss address	MA
9	reserved	
10	Page Table Address	PTA
11	Page Table Control	PTC
12	Aging frequency control	AFC

TLB Commands

Cmd$_4$	Description	Assembler
0	No operation	
1	Probe TLB entry	TLBPB
2	Read TLB entry	TLBRD
3	Write TLB entry corresponding to random register	TLBWR
4	Write TLB entry corresponding to index register	TLBWI
5	Enable TLB	TLBEN
6	Disable TLB	TLBDIS
7	Read register	TLBRDREG
8	Write register	TLBWRREG

9	Invalidate all entries	TLBINV
10	Get age count	TLBRDAGE
11	Set age count	TLBWRAGE

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

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

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

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

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

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

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

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

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

WAIT – Wait For Signal

Description:

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

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

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

Instruction Formats:

31_6	$\sim_3$	Imm_5	Rb_5	Ra_5	0_2	$02h_6$

Operation:

```
if (no signal)
   delay instruction
else
   mark instruction done
```

Notes:

This instruction waits for a signal to occur before proceeding.

WTME – Write Tag Memory Enable

Description:

This instruction updates a bit enabling or disabling the tagged memory system for a given memory page. The access key determining which map is updated must be set in the pcr CSR register. Register Ra contains a page number (0 to 1023) to enable or disable tagged memory on, and Rb contains the value to set the bit to. (1 = enable tagged memory, 0 = disable).

Instruction Format:

01_6	$\sim_3$	$1Dh_5$	Rb_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Exceptions: none

Notes:

XNOR – Bitwise Exclusive Nor

Description:

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

Instruction Format:

0Eh$_6$	~$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 0.5

Execution Units: All ALUs

Exceptions: none

XOR – Bitwise Exclusive Or

Description:

Perform a bitwise exclusive or operation between operands.

Instruction Format:

Immed$_{14}$	Rt$_5$	Ra$_5$	L$_2$	0Ah$_6$

L$_2$: 0 = 14 bit constant, 1 = 30 bit constant

0Ah$_6$	Sz$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	02h$_6$

Double Operations

0Ah$_6$	Op2$_6$	~$_5$	Sz$_3$	Rc$_5$	Rb$_5$	Rt$_5$	Ra$_5$	1$_2$	02h$_6$

Clock Cycles: 0.5

Execution Units: All ALUs

Exceptions: none

ZXB – Zero Extend Byte

Description:

Zero extend a byte as an unsigned value to the full width of the machine.

Instruction Format:

01_6	$\sim_3$	$0Ah_5$	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

ZXC – Zero Extend Char

Description:

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

Instruction Format:

01_6	$\sim_3$	$09h_5$	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

ZXH – Zero Extend Half-Word

Description:

Zero extend a half word as an unsigned value to the full width of the machine.

Instruction Format:

01_6	$\sim_3$	08_5	Rt_5	Ra_5	0_2	02h_6

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

Floating Point

Overview

The floating-point unit provides basic floating-point operations including addition, subtraction, multiplication, division, square root, and float to integer and integer to float conversions. The core contains only a single floating-point unit. Only double precision floating point operations are supported. The ISA itself supports single, double, and quad precision. The core uses the register set specified in the floating point CSR (FSTAT) register sets for the floating-point registers.

The precision field ($prec_2$) should be set to 1.

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

Representation

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

Double Precision Format:

63	62	61 52	51 0
S_M	S_E	Exponent	Mantissa

S_M – sign of mantissa
S_E – sign of exponent

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

S_eEEEEEEEEEE	
11111111111	Maximum exponent
….	
01111111111	exponent of zero
….	
00000000000	Minimum exponent

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

Instruction Format (Short – two source operand)

The assumed precision is double precision.

31 26	25 23	22 18	17 13	12 8	7 6	5 0
Func$_6$	Rm$_3$	Rt$_5$	Rb$_5$	Ra$_5$	0$_2$	0Fh$_6$

Not all instructions required the Rb$_5$ field. If not required Rb should be set to zero.

Instruction Format (Short – one source operand)

31 26	25 23	22 18	17	16 13	12 8	7 6	5 0
Func$_6$	Rm$_3$	Rt$_5$	0$_1$	Prc$_4$	Ra$_5$	0$_2$	0Fh$_6$

Instruction Format (Long)

The long instruction format adds a precision field and additional register spec field.

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
Func$_6$	~$_7$	Prc$_4$	Rm$_3$	Rt$_5$	Rc$_5$	Rb$_5$	Ra$_5$	1$_2$	0Fh$_6$

Prc$_4$	Precision
0	Half (16 bit)
1	Single (32 bit)
2	Double (64 bit)
3	Triple (96 bit)
4	Quad (128 bit)
5 to 7	reserved

Prc$_{[3]}$ = parallel operation bit (1=SIMD)

FABS – Floating Absolute Value

Description:

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

Instruction Format:

31 26	25 23	22	21 18	17 13	12 8	7 6	5 0
15h$_6$	Rm$_3$	0$_1$	Prc$_4$	Rt$_5$	Ra$_5$	0$_2$	0Fh$_6$

Clock Cycles: 2

Execution Units: Floating Point

FADD – Floating point addition

Description:

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

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
$04h_6$	Rm_3	Rb_5	Rt_5	Ra_5	0_2	$0Fh_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
$04h_6$	$\sim_7$	Prc_4	Rm_3	$\sim_5$	Rb_5	Rt_5	Ra_5	1_2	$0Fh_6$

Clock Cycles: 10

Execution Units: Floating Point

FBcc – Conditional Branch

Description:

If the branch condition is true, an eleven-bit sign extended value is shift left once and added to the program counter. The branch is relative to the address of the instruction directly following the branch. If the branch branches back to itself a branch exception will be generated.

Instruction Format:

31 23	22 18	17 16	15 13	12 8	7 6	5 0
Displacement$_{11..3}$	Rb$_5$	D$_{2..1}$	Cond$_3$	Ra$_5$	0$_2$	05h$_6$

Opcode$_6$	Cond$_2$	Mne.	
30h	0	FBEQ	Ra = Rb
	1	FBNE	Ra <> Rb
	2	FBLT	Ra < Rb
	3	FBGE	Ra >= Rb
	4		reserved
	5		reserved
	6		
	7	BUN	unordered

Clock Cycles:

Typically 1 with correct branch outcome and target prediction.

Execution Units: FCU Only

Exceptions: branch displacement

FCMP - Float Compare

Description:

The register compare instruction compares two registers as floating-point values and sets the flags in the target register as a result. The source operands are from the register set specified in the FSTAT CSR. The target register is from the register set specified in the MSTATUS CSR.

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
$06h_6$	Rm_3	Rb_5	Rt_5	Ra_5	0_2	$0Fh_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
$06h_6$	$\sim_7$	Prc_4	Rm_3	$\sim_5$	Rb_5	Rt_5	Ra_5	1_2	$0Fh_6$

Clock Cycles: 2

Execution Units: FPU

Operation:

```
if Ra < Rb
        Rt[1]= true
else
        Rt[1] = false
if mag Ra < mag Rb
        Rt[2] = true
else
        Rt[2] = false
if Ra = Rb
        Rt[0] = true
else
        Rt[0] = false
if Ra <= Rb
        Rt[3] = true
else
        Rt[3] = false
if unordered
        Rt[4] = true
else
        Rt[4] = false
```

FCVTSD – Convert Single to Double

Description:

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

Instruction Format:

31 26	25 24	23 21	20 16	15 11	10 6	5 0
19h$_6$	Prec$_2$	Rm$_3$	~$_5$	Rt$_5$	Ra$_5$	0Fh$_6$

Clock Cycles: 3

Execution Units: Floating Point

FDIV – Floating point divide

Description:

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

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
09h$_6$	Rm$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	0Fh$_6$

Clock Cycles: 115

Execution Units: Floating Point

FCX – Clear Floating Point Exceptions

Description:

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

Instruction Format:

31 26	25 24	23 18	17 13	12 8	7 6	5 0
$21h_6$	$\sim_2$	Imm_6	0_5	Ra_5	0_2	$0Fh_6$

Execution Units: All Floating Point

Operation:

Exceptions:

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

FDX – Floating Disable Exceptions

Description:

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

Instruction Format:

31 26	25 24	23 18	17 13	12 8	7 6	5 0
23h$_6$	~$_2$	Imm$_6$	0$_5$	Ra$_5$	0$_2$	0Fh$_6$

Clock Cycles: 2

Execution Units: Floating Point

FEX – Floating Enable Exceptions

Description:

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

Instruction Format:

31 26	25 24	23 18	17 13	12 8	7 6	5 0
$22h_6$	$\sim_2$	Imm_6	0_5	Ra_5	0_2	$0Fh_6$

Clock Cycles: 2

Execution Units: Floating Point

FMUL – Floating point multiplication

Description:

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

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
08h$_6$	Rm$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	0Fh$_6$

Clock Cycles: 12

Execution Units: Floating Point

FNABS – Floating Negative Absolute Value

Description:

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

Instruction Format:

31 26	25 23	22	21 18	17 13	12 8	7 6	5 0
18h$_6$	Rm$_3$	0$_1$	Prc$_4$	Rt$_5$	Ra$_5$	0$_2$	0Fh$_6$

Clock Cycles: 2

Execution Units: Floating Point

FNEG – Floating Negative Value

Description:

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

Instruction Format:

31 26	25 23	22 18	17	16 13	12 8	7 6	5 0
14h$_6$	Rm$_3$	Rt$_5$	0$_1$	Prc$_4$	Ra$_5$	0$_2$	0Fh$_6$

Clock Cycles: 2

Execution Units: Floating Point

FSIGN – Floating Sign

Description:

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

Instruction Format:

31 26	25 23	22	21 18	17 13	12 8	7 6	5 0
16h$_6$	Rm$_3$	0$_1$	Prc$_4$	Rt$_5$	Ra$_5$	0$_2$	0Fh$_6$

Clock Cycles: 2

Execution Units: Floating Point

FSEQ - Float Set Equal

Description:

The register compare instruction compares two registers as floating point values for equality and sets the target register as a result. The source operands are from the register set specified in the FSTAT CSR. The target register is from the register set specified in the MSTATUS CSR.

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
$3Ch_6$	Rm_3	Rb_5	Rt_5	Ra_5	0_2	$0Fh_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
$3Ch_6$	$\sim_7$	Prc_4	Rm_3	$\sim_5$	Rb_5	Rt_5	Ra_5	1_2	$0Fh_6$

Clock Cycles: 1

Execution Units: FPU

Operation:

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

FSGE - Float Set Greater Than or Equal

Description:

The register compare instruction compares two registers as floating point values for greater than or equal and sets the target register as a result. The source operands are from the register set specified in the FSTAT CSR. The target register is from the register set specified in the MSTATUS CSR.

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
$39h_6$	Rm_3	Rb_5	Rt_5	Ra_5	0_2	$0Fh_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
$39h_6$	$\sim_7$	Prc_4	Rm_3	$\sim_5$	Rb_5	Rt_5	Ra_5	1_2	$0Fh_6$

Clock Cycles: 1

Execution Units: FPU

Operation:

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

FSGT - Float Set Greater Than

Description:

The register compare instruction compares two registers as floating-point values for greater than and sets the target register as a result. The source operands are from the register set specified in the FSTAT CSR. The target register is from the register set specified in the MSTATUS CSR.

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
$3Bh_6$	Rm_3	Rb_5	Rt_5	Ra_5	0_2	$0Fh_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
$3Bh_6$	$\sim_7$	Prc_4	Rm_3	$\sim_5$	Rb_5	Rt_5	Ra_5	1_2	$0Fh_6$

Clock Cycles: 1

Execution Units: FPU

Operation:

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

FSLE - Float Set Less Than or Equal

Description:

The register compare instruction compares two registers as floating-point values for less than or equal and sets the target register as a result. The source operands are from the register set specified in the FSTAT CSR. The target register is from the register set specified in the MSTATUS CSR.

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
$3Ah_6$	Rm_3	Rb_5	Rt_5	Ra_5	0_2	$0Fh_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
$3Ah_6$	$\sim_7$	Prc_4	Rm_3	$\sim_5$	Rb_5	Rt_5	Ra_5	1_2	$0Fh_6$

Clock Cycles: 1

Execution Units: FPU

Operation:

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

FSLT - Float Set Less Than

Description:

The register compare instruction compares two registers as floating point values for less than and sets the target register as a result. The source operands are from the register set specified in the FSTAT CSR. The target register is from the register set specified in the MSTATUS CSR.

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
$38h_6$	Rm_3	Rb_5	Rt_5	Ra_5	0_2	$0Fh_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
$38h_6$	$\sim_7$	Prc_4	Rm_3	$\sim_5$	Rb_5	Rt_5	Ra_5	1_2	$0Fh_6$

Clock Cycles: 1

Execution Units: FPU

Operation:

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

FSNE - Float Set Not Equal

Description:

The register compare instruction compares two registers as floating point values for inequality and sets the target register as a result. The source operands are from the register set specified in the FSTAT CSR. The target register is from the register set specified in the MSTATUS CSR.

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
$3Dh_6$	Rm_3	Rb_5	Rt_5	Ra_5	0_2	$0Fh_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
$3Dh_6$	$\sim_7$	Prc_4	Rm_3	$\sim_5$	Rb_5	Rt_5	Ra_5	1_2	$0Fh_6$

Clock Cycles: 1

Execution Units: FPU

Operation:

```
if Ra <> Rb
        Rt = true
else
        Rt = false
```

FSUN - Float Set if Unordered

Description:

The register compare instruction compares two registers as floating point values for unorderliness and sets the target register as a result. The source operands are from the register set specified in the FSTAT CSR. The target register is from the register set specified in the MSTATUS CSR.

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
3Eh$_6$	Rm$_3$	Rb$_5$	Rt$_5$	Ra$_5$	0$_2$	0Fh$_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
3Eh$_6$	~$_7$	Prc$_4$	Rm$_3$	~$_5$	Rb$_5$	Rt$_5$	Ra$_5$	1$_2$	0Fh$_6$

Clock Cycles: 1

Execution Units: FPU

Operation:

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

FSQRT – Floating point square root

Description:

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

Instruction Format:

31 26	25 23	22	21 18	17 13	12 8	7 6	5 0
1Dh$_6$	Rm$_3$	0$_1$	Prc$_4$	Rt$_5$	Ra$_5$	0$_2$	0Fh$_6$

Clock Cycles: 110

Execution Units: Floating Point

FSUB – Floating point subtraction

Description:

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

Instruction Format:

31 26	25 23	22 18	17 13	12 8	7 6	5 0
$05h_6$	Rm_3	Rb_5	Rt_5	Ra_5	0_2	$0Fh_6$

47 42	41 35	34 31	30 28	27 23	22 18	17 13	12 8	7 6	5 0
$05h_6$	$\sim_7$	Prc_4	Rm_3	$\sim_5$	Rb_5	Rt_5	Ra_5	1_2	$0Fh_6$

Clock Cycles: 10

Execution Units: Floating Point

FSYNC -Synchronize

Description:

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

Instruction Format:

31 26	25 24	23 21	20 16	15 11	10 6	5 0
36h$_6$	~$_2$	~$_3$	~$_5$	~$_5$	~$_5$	0Fh$_6$

Clock Cycles: varies depending on queue contents

FTOI – Floating Convert to Integer

Description:

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

Instruction Format:

31 26	25 23	22	21 18	17 13	12 8	7 6	5 0
12h$_6$	Rm$_3$	0$_1$	Prc$_4$	Rt$_5$	Ra$_5$	0$_2$	0Fh$_6$

Clock Cycles: 3

Execution Units: Floating Point

FTX – Trigger Floating Point Exceptions

Description:

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

Instruction Format:

31 26	25 24	2322	21 16	15 11	10 6	5 0
20h$_6$	Prec$_2$	~$_2$	Imm$_6$	0$_5$	Ra$_5$	0Fh$_6$

Execution Units: All Floating Point

Operation:

Exceptions:

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

ITOF – Convert Integer to Float

Description:

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

Instruction Format:

31 26	25 23	22	21 18	17 13	12 8	7 6	5 0
15h$_6$	Rm$_3$	M$_1$	Prc$_4$	Rt$_5$	Ra$_5$	0$_2$	0Fh$_6$

M$_1$: Ra register set. 0 = use float register set, 1 = move from integer register set to float

Clock Cycles: 3

Execution Units: Floating Point

Vector Programming Model

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

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

Vector Length (VL register)

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

7	6	0
0	Length$_{6..0}$	

Vector Masking

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

Vector Mask (Vm registers)

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

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

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

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

Field Descriptions

The MSB of the Prc$_4$ field indicates a parallel operation (SIMD) if set to 1.

Prc$_4$	
0	16 bit – half precision
1	32 bit - single precision
2	64 bit – double precision

3	96 bit – triple precision
4	128 bit – quad precision
5	reserved
6	reserved
7	reserved

Detailed Vector Instruction Set

LV – Load Vector

Synopsis

Load vector

Description:

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

Instruction Format:

$Immed_{16}$	Vt_5	Ra_5	$36h_6$

Operation

```
for x = 0 to VL-1
    if vm[x]
        Vt[x] = $memory_{64}$[Ra + Immed + 8 * x]
    else
        NOP
```

Exceptions: DBE, DBG, LMT

LVWS – Load Vector With Stride

Synopsis

Load vector

Description:

Load a vector register from memory using indexed addressing.

Instruction Format:

$26h_6$	Vm_3	3_2	Vt_5	Rb_5	Ra_5	$16h_6$

Operation

```
for x = 0 to VL-1
    Vt[x] = $\text{memory}_{64}$[Ra+Rb*x*8]
```

Exceptions: DBE, DBG, LMT

LVX – Load Vector

Synopsis

Load vector

Description:

Load a vector register from memory using vector indexed addressing.

Instruction Format:

36h$_6$	~$_3$	3$_2$	Vt$_5$	Vb$_5$	Ra$_5$	16h$_6$

Operation

```
for x = 0 to VL-1
    Vt[x] = memory$_{64}$[Ra+Vb[x]]
```

Exceptions: DBE, DBG, LMT

SV – Store Vector

Synopsis

Load vector

Description:

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

Instruction Format:

Immed_{16}	Vb_5	Ra_5	37h_6

Operation

```
for x = 0 to VL-1
    if (vm[x])
        memory_64[Ra + Immed + 8 * x] = Vb[x]
    else
        NOP
```

Exceptions: DBE, DBG, LMT

SVWS – Store Vector With Stride

Synopsis

Store vector

Description:

Store a vector register to memory using indexed addressing.

Instruction Format:

$27h_6$	Vm_3	3_2	Vc_5	Rb_5	Ra_5	$16h_6$

Operation

```
for x = 0 to VL-1
    $memory_{64}[Ra+Rb*(x*8)] = Vc[x]$
```

Exceptions: DBE, DBG, LMT

SVX – Store Vector

Synopsis

Load vector

Description:

Store a vector register to memory using vector indexed addressing.

Instruction Format:

$37h_6$	$\sim_3$	3_2	Vc_5	Vb_5	Ra_5	$16h_6$

Operation

```
for x = 0 to VL-1
        $memory_{64}[Ra+Vb[x]] = Vc[x]$
```

Exceptions: DBE, DBG, LMT

V2BITS

Synopsis

Convert Boolean vector to bits.

$21h_6$	Vm_3	0_2	0_5	Rt_5	Va_5	$01h_6$

Description

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

Operation

```
For x = 0 to VL-1
    Rt[x] = Va[x].LSB
```

Exceptions: none

Execution Units: ALUs

VABS – Absolute value

Synopsis

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

Description

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

Instruction Format

03_6	Vm_3	T_2	Vt_5	0_5	Va_5	$01h_6$

Operation

```
        for x = 0 to VL - 1
                if (Vm[x]) Vt[x] = Va[x] < 0 ? –Va[x] : Va[x]
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VADD - Add

Synopsis

Vector register add. Vt = Va + Vb

Description

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

Float Double Instruction Format

This instruction format assumes a double precision (64 bit), with the rounding mode in use specified by the round mode field in the floating point CSR.

04_6	Vm_3	Vt_5	Vb_5	Va_5	0_2	$01h_6$

Integer Word Instruction Format

This instruction format assumes a word precision (64 bit).

04_6	Vm_3	Vt_5	Vb_5	Va_5	0_2	$31h_6$

Float Instruction Format

This instruction format allows the precision and rounding mode to be set in the instruction.

04_6	Vm_3	$\sim_9$	Prc_4	Rm_3	Vt_5	Vb_5	Va_5	1_2	$01h_6$

Integer Instruction Format

04_6	Vm_3	$\sim_9$	Prc_4	$\sim_3$	Vt_5	Vb_5	Va_5	1_2	$31h_6$

Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] + Vb[x]
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VADDS – Add Scalar

Synopsis

Vector register add. Vt = Va + Rb

Description

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

Instruction Format

$14h_6$	Vm_3	T_2	Vt_5	Rb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = Vb[x] + Rb
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VAND – Bitwise And

Synopsis

Vector register bitwise and. Vt = Va & Vb

Description

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

Instruction Format

08_6	Vm_3	$\sim_9$	Prc_4	Rm_3	Vt_5	Vb_5	Va_5	1_2	$31h_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = Va[x] & Vb[x]
```

Execution Units: ALUs

VANDS – Bitwise And with Scalar

Synopsis

Vector register bitwise and. Vt = Va & Rb

Description

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

Instruction Format

$18h_6$	Vm_3	0_2	Vt_5	Rb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = Va[x] & Rb[x]
```

VASR – Arithmetic Shift Right

Synopsis

Vector signed shift right.

$0Eh_6$	S	M_2	S	A	Vt_5	Amt_5	Va_5	$01h_6$

Description

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

Operation

```
For x = 0 to VL-1
        if (Vm[x]) Vt[x] = Va[x] >> amt
```

Exceptions: none

S_2	Amount Field	
0	general purpose register	
1	vector register	
2	immediate	
3	reserved	

VBITS2V

Synopsis

Convert bits to Boolean vector.

20h$_6$	Vm$_3$	0$_2$	Vt$_5$	0$_5$	Ra$_5$	01h$_6$

Description

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

Operation

```
For x = 0 to VL-1
    if (Vm[x]) Vt[x] = Ra[x]
```

Exceptions: none

Execution Units: ALUs

VCIDX – Compress Index

Synopsis

Vector compression.

Description

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

Instruction Format

01_6	Vm_3	0_2	Vt_5	0_5	Ra_5	$01h_6$

Operation

```
y = 0
for x = 0 to VL - 1
    if (Vm[x])
        Vt[y] = Ra * x
        y = y + 1
```

VCMPRSS – Compress Vector

Synopsis

Vector compression.

Description

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

Instruction Format

00_6	Vm_3	0_2	Vt_5	0_5	Va_5	$01h_6$

Operation

```
y = 0
for x = 0 to VL - 1
    if (Vm[x])
        Vt[y] = Va[x]
        y = y + 1
```

VCNTPOP – Population Count

Synopsis

Vector register population count. Vt = popcnt(Va)

Description

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

Instruction Format

28h$_6$	Vm$_3$	0$_2$	Vt$_5$	0$_5$	Va$_5$	01h$_6$

Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = popcnt(Va[x])
```

VDIV - Divide

Synopsis

Vector register divide. Vt = Va / Vb

Description

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

Instruction Format

3Eh$_6$	Vm$_3$	T$_2$	Vt$_5$	Vb$_5$	Va$_5$	01h$_6$

Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] / Vb[x]
```

Operand Type

T$_2$	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VDIVS – Divide by Scalar

Synopsis

Vector register divide by scalar. Vt = Va / Rb

Description

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

Instruction Format

2Eh$_6$	Vm$_3$	T$_2$	Vt$_5$	Rb$_5$	Va$_5$	01h$_6$

Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] / Rb[x]
```

Operand Type

T$_2$	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VEINS / VMOVSV – Vector Element Insert

Synopsis

Vector element insert.

$22h_6$	~	M_2	0_2	Vt_5	Rb_5	Ra_5	$01h_6$

Description

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

Operation

Vt[Ra] = Rb

Exceptions: none

VEX / VMOVS – Vector Element Extract

Synopsis

Vector element insert.

$23h_6$	~	M_2	0_2	Rt_5	Vb_5	Ra_5	$01h_6$

Description

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

Operation

Rt = Vb[Ra]

Exceptions: none

VFLT2INT – Float to Integer

Synopsis

Vector float to integer.

$24h_6$	~	M_2	T_2	Vt_5	0_5	Va_5	$01h_6$

Description

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

Operation

```
For x = 0 to [Ra]-1
    Vt[x] = (int)Va[x]
```

Exceptions: none

VINT2FLT – Integer to Float

Synopsis

Vector float to integer.

$25h_6$	~	M_2	T_2	Vt_5	0_5	Va_5	$01h_6$

Description

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

Operation

```
For x = 0 to VL-1
    Vt[x] =(float) Va[x]
```

Exceptions: none

VMAND – Bitwise Mask And

Synopsis

Vector mask register bitwise and. Vmt = Vma & Vmb

Description

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

Instruction Format

$30h_6$	0_3	0_4	Vmt_3	0_2	Vmb_3	0_2	Vma_3	$01h_6$

Operation

Vmt = Vma & Vmb

Execution Units: ALUs

VMFILL –Mask Fill

Synopsis

Fill vector mask register with bits.

Description

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

Instruction Format

$30h_6$	5_3	0_2	Vmt_5	0_5	Ra_5	$01h_6$

Operation

```
for x = 0 to VLMAX
    if (x < Ra) Vmt[x] = 1
    else Vmt[x] = 0
```

Execution Units: ALUs

VMFIRST – Find First Set Bit

Synopsis

Convert Boolean vector to bits.

30h$_6$	6$_3$	0$_2$	0$_5$	Rt$_5$	~$_2$	Vm$_3$	01h$_6$

Description

The position of the first bit set in the mask register is copied to the target register. If no bits are set the value is 64. The search begins at the least significant bit and proceeds to the most significant bit.

Operation

Rt = first set bit number of (Vm)

Exceptions: none

Execution Units: ALUs

VMLAST – Find Last Set Bit

Synopsis

Convert Boolean vector to bits.

$30h_6$	7_3	0_2	0_5	Rt_5	~$_2$	Vm_3	$01h_6$

Description

The position of the last bit set in the mask register is copied to the target register. If no bits are set the value is 64. The search begins at the most significant bit of the mask register and proceeds to the least significant bit.

Operation

Rt = first set bit number of (Vm)

Exceptions: none

Execution Units: ALUs

VMOR – Bitwise Mask Or

Synopsis

Vector mask register bitwise and. Vmt = Vma | Vmb

Description

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

Instruction Format

$30h_6$	1_3	0_4	Vmt_3	0_2	Vmb_3	0_2	Vma_3	$01h_6$

Operation

Vmt = Vma | Vmb

Operand Type

T_2	Operand Type	
0	Integer	
1	reserved	
2	reserved	
3	reserved	

Execution Units: ALUs

VMOV – Move Vector Control Register

Description:

.

Instruction Format:

33h$_6$	0$_5$		Vt$_5$	Ra$_5$	02h$_6$

Va$_5$/Vt$_5$		
0 to 7	Vector Mask	
15	Vector Length	

33h$_6$	1$_5$		Rt$_5$	Va$_5$	02h$_6$

Clock Cycles: 1

Execution Units: ALUs

VMPOP – Mask Population Count

Synopsis

Convert Boolean vector to bits.

$30h_6$	4_3	0_2	0_5	Rt_5	$\sim_2$	Vm_3	$01h_6$

Description

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

Operation

Rt = population count(Vm)

Exceptions: none

Execution Units: ALUs

VMUL - Multiply

Synopsis

Vector register multiply. Vt = Va * Vb

Description

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

Instruction Format

$3Ah_6$	Vm_3	T_2	Vt_5	Vb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] * Vb[x]
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VMULS – Multiply by Scalar

Synopsis

Vector register multiply by scalar. Vt = Va * Rb

Description

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

Instruction Format

$2Ah_6$	Vm_3	T_2	Vt_5	Rb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] * Rb
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VMXNOR – Bitwise Mask Exclusive Nor

Synopsis

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

Description

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

Instruction Format

$30h_6$	3_3	0_4	Vmt_3	0_2	Vmb_3	0_2	Vma_3	$01h_6$

Operation

Vmt = Vma ^ Vmb

Execution Units: ALUs

VMXOR – Bitwise Mask Exclusive Or

Synopsis

Vector mask register bitwise and. Vmt = Vma ^ Vmb

Description

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

Instruction Format

$30h_6$	2_3	0_4	Vmt_3	0_2	Vmb_3	0_2	Vma_3	$01h_6$

Operation

Vmt = Vma ^ Vmb

Execution Units: ALUs

VNEG – Negate

Synopsis

Vector register subtract. Vt = R0 - Va

Description

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

Instruction Format

$16h_6$	Vm_3	T_2	Vt_5	0_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = R0 - Va[x]
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VOR – Bitwise Or

Synopsis

Vector register bitwise or. Vt = Va | Vb

Description

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

Instruction Format

09_6	Vm_3	T_2	Vt_5	Vb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = Va[x] | Vb[x]
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VORS – Bitwise Or with Scalar

Synopsis

Vector register bitwise and. Vt = Va | Rb

Description

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

Instruction Format

$19h_6$	~	M_2	T_2	Vt_5	Rb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = Va[x] | Rb[x]
```

Operand Type

T_2	Operand Type	
0	Integer	
1	reserved	
2	reserved	
3	reserved	

VSxx / VSxxS

Synopsis

Vector register set. Vm = Va ? Vb

Description

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

Instruction Format

Vector-Vector Compare (VSxx)

$06_6/3F_6$	M_3	T_2	Cn_2	Vmt_3	Vb_5	Va_5	$01h_6$

Vector-Vector Unsigned Compare (VSxxU)

$27h_6/2F_6$	M_3	T_2	Cn_2	Vmt_3	Vb_5	Va_5	$01h_6$

Vector-Scalar Compare (VSxxS)

$07_6/0F_6$	M_3	T_2	Cn_2	Vmt_3	Rb_5	Va_5	$01h_6$

Vector-Scalar Unsigned Compare (VSxxSU)

$17h_6/1F_6$	M_3	T_2	Cn_2	Vmt_3	Rb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    Vt[x] = Va[x] ? Vb[x]
```

Operation:

For each vector element

```
if signed Va op signed Vb
    Vm = true
else
    Vm = false
```

Set Condition

Cn_3		
0	Equal	
1	Not Equal	
2	Less Than	

3	Greater Than or Equal	
4	Less Than or Equal	
5	Greater Than	
6	reserved	
7	unordered	

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSCAN

Synopsis

.

Description

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

Instruction Format

02_6	~	M_2	0_2	Vt_5	0_5	Ra_5	$01h_6$

Operation

```
sum = 0
for x = 0 to VL - 1
    Vt[x] = sum
    if (Vm[x])
        sum = sum + Ra
```

VSEQ – Set if Equal

Synopsis

Vector register set. Vm = Va == Vb

Description

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

Instruction Format

06_6	0	M_2	T_2	0_2	Vmt_3	Vb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    Vm[x] = Va[x] == Vb[x]
```

Operation:

For each vector element

```
if signed Va equals signed Vb
    Vm = true
else
    Vm = false
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSEQS – Set if Equal Scalar

Synopsis

Vector register set. Vm = Va == Rb

Description

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

Instruction Format

07_6	0	M_2	T_2	0_2	Vmt_3	Rb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    Vm[x] = Va[x] == Rb[x]
```

Operation:

For each vector element

```
if signed Va equals signed Rb
    Vm = true
else
    Vm = false
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSGE – Set if Greater or Equal

Synopsis

Vector register set. Vm = Va >= Vb

Description

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

Instruction Format

06_6	0	M_2	T_2	3_2	Vmt_3	Vb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    Vm[x] = Va[x] >= Vb[x]
```

Operation:

For each vector element

```
if signed Va greater than or equal signed Vb
    Vm = true
else
    Vm = false
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSGES – Set if Greater or Equal Scalar

Synopsis

Vector register set. Vm = Va >= Rb

Description

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

Instruction Format

07_6	0	M_2	T_2	3_2	Vmt_3	Rb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    Vm[x] = Va[x] >= Rb
```

Operation:

For each vector element

```
if signed Va greater than or equal signed Rb
    Vm = true
else
    Vm = false
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSHL – Shift Left

Synopsis

Vector shift left.

0Ch$_6$	S	M$_2$	S	A	Vt$_5$	Amt$_5$	Va$_5$	01h$_6$

Description

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

Operation

```
For x = 0 to VL-1
    if (Vm[x]) Vt[x] = Va[x] << amt
```

Exceptions: none

S$_2$	Amount Field	
0	general purpose register	
1	vector register	
2	immediate	
3	reserved	

VSHLV – Shift Vector Left

Synopsis

Vector shift left.

$10h_6$	~	M_2	0_2	Vt_5	Amt_5	Va_5	$01h_6$

Description

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

Operation

```
For x = VL-1 to Amt
    Vt[x] = Va[x-amt]
For x = Amt-1 to 0
    Vt[x] = 0
```

Exceptions: none

VSHR – Shift Right

Synopsis

Vector shift left.

0Dh$_6$	S	M$_2$	S	A	Vt$_5$	Amt$_5$	Va$_5$	01h$_6$

Description

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

Operation

```
For x = 0 to VL-1

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

Exceptions: none

S$_2$	Amount Field	
0	general purpose register	
1	vector register	
2	immediate	
3	reserved	

VSHRV – Shift Vector Right

Synopsis

Vector shift right.

11h$_6$	~	M$_2$	0$_2$	Vt$_5$	Amt$_5$	Va$_5$	01h$_6$

Description

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

Operation

```
For x = 0 to VL-Amt
    Vt[x] = Va[x+amt]
For x = VL-Amt +1 to VL-1
    Vt[x] = 0
```

Exceptions: none

VSIGN – Sign

Synopsis

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

Description

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

Instruction Format

$26h_6$	~	M_2	T_2	Vt_5	0_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] < 0 ? –1 : Va[x]=0 ? 0 : 1
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSLT – Set if Less Than

Synopsis

Vector register set. Vm = Va < Vb

Description

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

Instruction Format

06_6	0	M_2	T_2	2_2	Vmt_3	Vb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    Vm[x] = Va[x] < Vb[x]
```

Operation:

For each vector element

```
if signed Va less than signed Vb
    Vm = true
else
    Vm = false
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSNE – Set if Not Equal

Synopsis

Vector register set. Vm = Va != Vb

Description

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

Instruction Format

06_6	0	M_2	T_2	1_2	Vmt_3	Vb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    Vm[x] = Va[x] <> Vb[x]
```

Operation:

For each vector element

```
if signed Va not equal signed Vb
    Vm = true
else
    Vm = false
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSUB - Subtract

Synopsis

Vector register add. Vt = Va - Vb

Description

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

Instruction Format

05_6	~	M_2	T_2	Vt_5	Vb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL - 1
    if (Vm[x]) Vt[x] = Va[x] - Vb[x]
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	

2	reserved	
3	reserved	

VSUBRS – Subtract from Scalar

Synopsis

Vector register subtract. Vt = Rb - Va

Description

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

Instruction Format

$16h_6$	~	M_2	T_2	Vt_5	Rb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = Rb - Va[x]
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSUBS – Subtract Scalar

Synopsis

Vector register subtract. Vt = Va - Rb

Description

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

Instruction Format

$15h_6$	~	M_2	T_2	Vt_5	Rb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = Va[x] - Rb
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSUN – Set if Unordered

Synopsis

Vector register set. Vm = Va ? Vb

Description

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

Instruction Format

06_6	1	M_2	T_2	3_2	Vmt_3	Vb_5	Va_5	$01h_6$

Operation

```
        for x = 0 to VL-1
                Vm[x] = Va[x] ? Vb[x]
```

Operation:

For each vector element

```
if is unordered Va or Vb
        Vm = true
else
        Vm = false
```

Operand Type

T_2	Operand Type	
0	Integer	
1	Float double	
2	reserved	
3	reserved	

VSYNC -Synchronize

Description:

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

Instruction Format:

31 26	25 24	23 21	20 16	15 11	10 6	5 0
$36h_6$	$\sim_2$	$\sim_3$	$\sim_5$	$\sim_5$	$\sim_5$	$01h_6$

Clock Cycles: varies depending on queue contents

VXCHG - Exchange

Synopsis

Vector register exchange. Va = Vb;Vb= Va

Description

Exchange two vector registers (Va and Vb)

Instruction Format

$0B_6$	Vm_3	0_2	Va_5	Vb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL - 1
    if (Vm[x])
        Vb[x] = Va[x]
        Va[x] = Vb[x]
```

VXOR – Bitwise Exclusive Or

Synopsis

Vector register bitwise or. Vt = Va ^ Vb

Description

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

Instruction Format

$0Ah_6$	Vm_3	0_2	Vt_5	Vb_5	Va_5	$01h_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = Va[x] ^ Vb[x]
```

VXORS – Bitwise Exclusive Or with Scalar

Synopsis

Vector register bitwise and. Vt = Va ^ Rb

Description

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

Instruction Format

1Ah$_6$	Vm$_3$	0$_2$	Vt$_5$	Rb$_5$	Va$_5$	01h$_6$

Operation

```
for x = 0 to VL-1
    if (Vm[x]) Vt[x] = Va[x] ^ Rb[x]
```

GPU

Overview

The GPU is a 32-bit version of the FT64 instruction set.

GPU Instructions

Overview

The GPU executes a 32-bit subset of the FT64 instruction set, it has some of its own instructions as well. The GPU contains instructions specific to graphics processing. It also contains instructions to perform 32-bit fixed point arithmetic.

The GPU contains multiple dividers to increase the performance of division operations. A handle to a divider is returned by the divide instruction, a subsequent divider wait instruction must be executed to retrieve the divider results.

The GPU has its own internal call / return stack and hence does not need to access memory or use a link register to store and retrieve subroutine call and return addresses. Call and return operate differently than in the FT64 ISA. Call pushes the return address on an internal stack and RET pops the return address off the stack into the program counter. The internal stack contains 63 entries. Heavily recursive routines should not be used.

BLEND – Blend Colors

Description:

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

Instruction Format:

$\sim_4$	Rt_5	Rc_5	Rb_5	Ra_5	0_2	$0Ch_6$

Operation: Rt = (Ra * alpha) + (Rb * ~alpha)

Clock Cycles: 4

COLOR4TO8 – Convert RGB444 to RGB888

Description:

This instruction converts a sixteen-bit ZRGB4444 color value to a thirty-two-bit ZRGB8888 color. Each component of the color is extended with four zero bits.

Instruction Format:

$01h_6$	~	$\sim_2$	$0Bh_5$	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 4

COLOR8TO4 – Convert RGB888 to RGB444

Description:

This instruction converts a thirty-two-bit ZRGB8888 color value to a sixteen-bit ZRGB4444 color. Four bits are truncated off each color component.

Instruction Format:

$01h_6$	~	$~_2$	$1Bh_5$	Rt_5	Ra_5	0_2	$02h_6$

Clock Cycles: 4

FXDIV[.w] – Fixed Point Divide

Description:

This instruction divides Ra by Rb. A handle to the divider is placed in Rt as the divide result is not immediately available. The program may continue to execute after the divide is started. The handle should be used later in the program to retrieve the result value using the FXDIV.w instruction. Ra and Rb are fixed point numbers with sixteen whole and sixteen binary point places (16.16). The result is a (16.16) number.

FXDIV.w

This version of the instruction waits for the result of a divide operation to become available. A handle for the divider must be in Ra. The divider result will be placed in Rt once available. Note that this instruction waits until the divide is done which may take upwards of 70 clock cycles. However other instructions may be executed after the divide is started to hide some of the divide latency.

Instruction Format:

2Bh$_6$	W	~$_2$	Rt$_5$	Rb$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 10

FXMUL – Fixed Point Multiply

Description:

This instruction multiplies Ra by Rb. Ra and Rb are fixed point numbers with sixteen whole and sixteen binary point places (16.16). The result is a (16.16) number. Unlike divide this instruction does not return a handle.

Instruction Format:

3Bh$_6$	~	~$_2$	Rt$_5$	Rb$_5$	Ra$_5$	0$_2$	02h$_6$

Clock Cycles: 8

POPR – Pop Return Stack

Description:

This instruction pops the internal return stack. The value popped is discarded. This instruction allows performing a two-up level return from a subroutine.

Instruction Format:

$01h_6$	~	$\sim_2$	$0Eh_5$	$\sim_5$	$\sim_5$	0_2	$02h_6$

Clock Cycles: 1

TEST_CLIP

Description:

This instruction tests if the X, Y position of a point specified in the Ra and Rb registers respectively are within the clipping region. If clipping is enabled the point must be within the clip region. Whether or not clipping is enabled the point is tested to ensure it is within the target area the GPU is responsible for. A value of one indicates that the point should be clipped. A value of zero indicates the point should not be clipped. The target area and clipping region must have been previously set.

Instruction Format:

$21h_6$	~	$\sim_2$	Rt_5	Rb_5	Ra_5	0_2	$02h_6$

Clock Cycles: 4

TRANSFORM[.w]

Description:

This instruction performs a transform on a point specified in Ra, Rb, and Rc as the X, Y, and Z co-ordinates respectively. The transformation matrix must have previously been set.

If bit W of the instruction is clear, the transformation will be performed using the supplied register values and the results will be stored in a holding register. If the W bit of the instruction is set, the current transform result in the holding register will be written to the target registers Rta, Rtb and Rtc.

The TRANSFORM instruction will return immediately to continue program execution after the transform has started. The results are available after about 16 clocks. The transform writeback instruction will wait until the transform is complete, if it is not already complete, before updating the register file.

Instruction Format:

$11h_6$	W	$\sim_2$	Rc_5	Rb_5	Ra_5	0_2	$02h_6$

Instruction Format (writeback):

$11h_6$	1	$\sim_2$	Rtc_5	Rtb_5	Rta_5	0_2	$02h_6$

Clock Cycles: 18

Opcode Tables

Major Opcode (inst. bits 0 to 5)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	{FVECTOR}	{R2}	AUIPC	ADDI	CSR	SLTI	SLTUI	ANDI	ORI	XORI	SEQI	BLEND	REX	LEA	{FLOAT}
1x	BLcc	LH	BNEI#	LB	PUSHC	SB	{MNDX}	SWC	JAL	CALL	INC / DEC	LFx	SGTUI	LWR	CACHE	EXEC
2x	LC	LCU / LHU	{BITFIELD}	LBU	SC	CAS	BBC / BBS	LUI	JMP	RET	MULFI	SFx	SGTI	{CMPRSSD}	MODI	{AMO}
3x	Bcc	{IVECTOR}	BEQ#	LW	CHK	SW	LV	SV	MULUI	FXMULI	MULI	SH	DIVUI	NOP	DIVI	{AMO}

Memory Indexed (inst. bits 21,22,28 to 31) (or bits 16,17, 28 to 31 for stores) (bit 31 = 0 for loads, 1 for stores)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	LVBX	LVBUX	LVCX	LVCUX	LVHX	LVHUX	LVWX		LCX	LCUX	LBUX					
1x	LHX	LHUX	LWX	LBX	LWRX				LVWS	LVX		LFHX	LFSX	LFDX	CACHEX	LFQX
2x	SBX	SHX	SWX	SWCX	SCX	CASX		SVWS			INCX / DECX	SFHX	SFSX	SFDX		SFQX
3x				PUSH				SVX								

Major Funct (inst. bits 26 to 31)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	{BCD}	{R1}	ADDV	BMM	ADD	SUB	SLT	SLTU	AND	OR	XOR	SEQ	NAND	NOR	XNOR	{shift31}
1x		TRANSFORM			MODU	MODSU	MOD		LEAX		INCX		MOV		PTRDIF	{shift63}
2x	TESTSCN	TEST_CLIP	MOV	MOV	MULUH	MULSUH	MULH		SLE	SLEU	MULF	FXDIV	MIN	MAX	MAJ	{shiftr}
3x	SEI / CLI	WAIT	RTI	VMOV	CHK	SLE	{SEG}		MULU	MULSU	MUL	FXMUL	DIVU	DIVSU	DIV	TLB

Major Funct (inst. bits 42 to 47)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	RTOP	{R1}	ADDV		ADD	SUB			AND	OR	XOR	SUBV	NAND	NOR	XNOR	{shift31}
1x												MUX	MOV			{shift63}
2x			MOV					CMOVFNZ	CMOVEZ	CMOVNZ			MIN3	MAX3	MAJ	{shiftr}
3x									MULU	MULSU	MUL		DIVMODU	DIVMODSU	DIVMOD	

Float Funct (inst. bits 26 to 31)

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

R1 (inst. bits 21 to 25)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	CNTLZ	CNTLO	CNTPOP	COM	ABS	NOT	ISPTR	NEG	ZXH	ZXC	ZXB	4to8	2to8		POPR	
1x	MEMDB	MEMSB	SYNC	EXEC	CHAIN OFF	CHAIN ON	SETWB		SXH	SXC	SXB	8to4	8to2			RD_CMD_COUNT

Compound Opcode (inst. bits 36 to 41)

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NOP				ADD	SUB	SLT	SLTU	AND	OR	XOR	SEQ				
1x													SGTUI			
2x	COM	NOT									MULFI		SGTI	{CMPRSSD}	MODI	
3x									MULUI		MULI		DIVUI	NOP	DIVI	

Shift (inst. bits 22 to 25)

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

Vector Funct (inst. bits 26 to 31)

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

Appendix

Reducing the size of the core.

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

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

SMT support adds considerably to the size of the core. The additional logic requirements for SMT consume approximately 28,000 LUTs. The core size can be reduced significantly by building the core without SMT. This may be done by commenting out the SUPPORT_SMT configuration define.

Support for debugging logic adds to the size of the core. The core may be built without debugging logic in order to reduce the size. The configuration define for this is SUPPORT_DBG.

Some of the issue logic may be omitted in order to reduce the size of the core. However the issue logic was found not to have a large impact on core size. The configuration define is FULL_ISSUE_LOGIC.

Some of the functional units may be reduced in number. Setting the number of instruction decoders to lower number (NUM_IDU) or the number of ALU's (NUM_ALU) will reduce the size of the core. Floating-point may be removed in which case executing floating-point instructions will cause an unimplemented instruction exception. Note that removing functional units from the core may significantly impact performance.

Architectural Register vs Physical Registers

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

Register Renaming

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

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

Instruction Cache Miss

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

Branches

Branches store the target address in iqentry_a0 the immediate constant field of the queue. The target address has to be stored somewhere in the instruction queue so that it may be used to update the branch target buffer later. It can't be stored in the result field, and it can't be stored in one of the other argument fields. Arg0 is the only place it can be stored safely.

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

Configuration Defines

Q2VECTORS

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

SUPPORT_SMT

- Enables support for SMT and two threads of execution.
- increase the size of the core

SUPPORT_DBG

- - enables support for debug registers and logic

Parameters

SUP_TXE

- default 0
- enables support for the call target exception

SUP_VECTOR

- default 1
- enables support for vector instructions

-

Instructions Supported Only on ALU #0

The following less frequently used instructions are only supported on ALU #0 in order to reduce the size of the core. ALU #0 is almost double the size of ALU#1 due to its support of additional instructions.

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

Glossary

Burst Access

A burst access is a number of bus accesses that occur rapidly in a row in a known sequence. If hardware supports burst access the cycle time for access to the device is drastically reduced. For instance dynamic RAM memory access is really fast for sequential burst access, and somewhat slower for random access.

BTB

An acronym for Branch Target Buffer. The branch target buffer is used to improve the performance of a processing core. The BTB is a table that stores the branch target from previously executed branch instructions. A typical table may contain 1024 entries. The table is typically indexed by part of the branch address. Since the target address of a branch type instruction may not be known at fetch time, the address is speculated to be the address in the branch target buffer. This allows the machine to fetch instructions in a continuous fashion without pipeline bubbles. In many cases the calculated branch address from a previously executed instruction remains the same the next time the same instruction is executed. If the address from the BTB turns out to be incorrect, then the machine will have to flush the instruction queue or pipeline and begin fetching instructions from the correct address.

FPGA

An acronym for Field Programmable Gate Array. FPGA’s consist of a large number of small RAM tables, flip-flops and other logic. These are all connected together with a programmable connection network. FPGA’s are ‘in the field’ programmable, and usually re-programmable. An FPGA’s re-programmability is typically RAM based. They are often used with configuration PROM’s so they may be loaded to perform specific functions.

HDL

An acronym that stands for ‘Hardware Description Language’. A hardware description language is used to describe hardware constructs at a high level.

Instruction Bundle

A group of instructions. It is sometimes required to group instructions together into bundle. For instance all instructions in a bundle may be executed simultaneously on a processor as a unit. Instructions may also need to be grouped if they are oddball in size for example 41 bits, so that they can be fit evenly into memory. Typically a bundle has some bits that are global to the bundle, such as template bits, in addition to the encoded instructions.

ISA

An acronym for Instruction Set Architecture. The group of instructions that an architecture supports. ISA’s are sometimes categorized at extreme edges as RISC or

CISC. FT64 falls somewhere in between with features of both RISC and CISC architectures.

Linear Address

A linear address is the resulting address from a virtual address after segmentation has been applied.

Physical Address

A physical address is the final address seen by the memory system after both segmentation and paging have been applied to a virtual address. One can think of a physical address as one that is "physically" wired to the memory.

Program Counter

A processor register dedicated to addressing instructions in memory. It is also often and perhaps more aptly called an instruction pointer. The program counter got it's name because it usually increments (or counts) automatically after an instruction is fetched. In early machines in some rare cases the program counter did not count in a sequential binary fashion, but instead used other forms of a counter such as a grey counter or linear feedback shift register. In some machines the program counter addresses bundles of instructions rather than individual instructions. This is common with some stack machines where multiple instructions are packed into a memory word.

ROB

An acronym for ReOrder Buffer. The re-order buffer allows instructions to execute out of order yet update the machine's state in order by tracking instruction state and variables. In FT64 the re-order buffer is a circular queue with a head and tail pointers. Instructions at the head are committed if done to the machine's state then the head advanced. New instructions are queued at the buffer's tail as long as there is room in the queue. Instructions in the queue may be processed out of the order that they entered the queue in depending on the availability of resources (register values and functional units).

RSB

An acronym that stands for return stack buffer. A buffer of addresses used to predict the return address which increases processor performance. The RSB is usually small, typically 16 entries. When a return instruction is detected at time of fetch the RSB is accessed to determine the address of the next instruction to fetch. Predicting the return address allows the processing core to continuously fetch instructions in a speculative fashion without bubbles in the pipeline. The return address in the RSB may turn out to be detected as incorrect during execution of the return instruction, in which case the pipeline or instruction queue will need to be flushed and instructions fetched from the proper address.

SIMD

An acronym that stands for 'Single Instruction Multiple Data'. SIMD instructions are usually implemented with extra wide registers. The registers contain multiple data items, such as a 128 bit register containing four 32 bit numbers. The same instruction is applied to all the data items in the register at the same time. For some applications SIMD instructions can enhance performance considerably.

Stack Pointer

A processor register dedicated to addressing stack memory. Sometimes this register is assigned by convention from the general register pool. This register may also sometimes index into a small dedicated stack memory that is not part of the main memory system. Sometimes machines have multiple stack pointers for different purposes but they all work on the idea of a stack. For instance in Forth machines there are typically two stacks, one for data and one for return addresses.

TLB

TLB stands for translation look-aside buffer. This buffer is used to store address translations for fast memory access in a system with an mmu capable of performing address translations.

Miscellaneous

Reference Material

Below is a short list of some of the reading material I've studied. I've downloaded a fair number of documents on computer architecture from the web. Too many to list.

Modern Processor Design Fundamentals of Superscalar Processors by John Paul Shen, Mikko H. Lipasti. Waveland Press, Inc.

Computer Architecture A Quantitative Approach, Second Edition, by John L Hennessy & David Patterson, published by Morgan Kaufman Publishers, Inc. San Franciso, California is a good book on computer architecture. There is a newer edition of the book available.

Memory Systems Cache, DRAM, Disk by Bruce Jacob, Spencer W. Ng., David T. Wang, Samuel Rodriguez, Morgan Kaufman Publishers

PowerPC Microprocessor Developer's Guide, SAMS publishing. 201 West 103rd Street, Indianapolis, Indiana, 46290

80386/80486 Programming Guide by Ross P. Nelson, Microsoft Press

Programming the 286, C. Vieillefond, SYBEX, 2021 Challenger Drive #100, Alameda, CA 94501

Tech. Report UMD-SCA-2000-02 ENEE 446: Digital Computer Design — An Out-of-Order RiSC-16

Programming the 65C816, David Eyes and Ron Lichty, Western Design Centre Inc.

Microprocessor Manuals from Motorola, and Intel,

The SPARC Architecture Manual Version 8, SPARC International Inc, 535 Middlefield Road. Suite210 Menlo Park California, CA 94025

The SPARC Architecture Manual Version 9, SPARC International Inc, Sab Jose California, PTR Prentice Hall, Englewood Cliffs, New Jersey, 07632

The MMIX processor: http://mmix.cs.hm.edu/doc/instructions-en.html

RISCV 2.0 Spec, Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovi´c CS Division, EECS Department, University of California, Berkeley {waterman|yunsup|pattrsn|krste}@eecs.berkeley.edu

Trademarks

IBM® is a registered trademark of International Business Machines Corporation. Intel® is a registered trademark of Intel Corporation. HP® is a registered trademark of Hewlett-Packard Development Company. "SPARC® is a registered trademark of SPARC International, Inc.

WISHBONE Compatibility Datasheet

The FT64 core may be directly interfaced to a WISHBONE compatible bus.

WISHBONE Datasheet WISHBONE SoC Architecture Specification, Revision B.3		
Description:	Specifications:	
General Description:	Central processing unit (CPU core)	
Supported Cycles:	MASTER, READ / WRITE MASTER, READ-MODIFY-WRITE MASTER, BLOCK READ / WRITE, BURST READ (FIXED ADDRESS)	
Data port, size:	64 bit	
Data port, granularity:	8 bit	
Data port, maximum operand size:	64 bit	
Data transfer ordering:	Little Endian	
Data transfer sequencing	any (undefined)	
Clock frequency constraints:		
Supported signal list and cross reference to equivalent WISHBONE signals	Signal Name:	WISHBONE Equiv.
	ack_i	ACK_I
	adr_o(31:0)	ADR_O()
	clk_i	CLK_I
	dat_i(63:0)	DAT_I()
	dat_o(63:0)	DAT_O()
	cyc_o	CYC_O
	stb_o	STB_O
	wr_o	WE_O
	sel_o(7:0)	SEL_O
	cti_o(2:0)	CTI_O
	bte_o(1:0)	BTE_O
Special Requirements:		