FT64

Overview

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

- 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 floating point unit

History

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

Goals

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

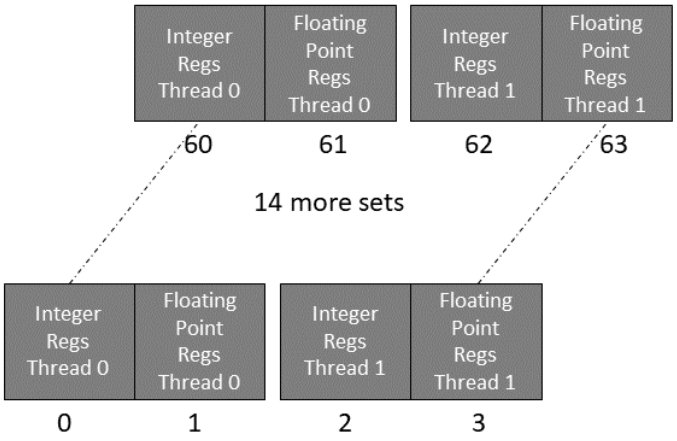
Implementing many features of the Thor core using a fixed 32 bit instruction set.

Easy implementation of a compiler.

Eventual implementation as a four-way superscalar processing core.

Register Sets

There are 64 sets of 32 general purpose registers in the architecture. The odd registers sets may be used as floating-point registers for the even register set. When SMT is turned on register sets are used in pairs. The following is an illustration of register set usage.



On reset register set #0 is selected to be the operating register set. On interrupt every fourth register sets #4 to #28 will be selected according to the level of the interrupt.

Machine State	Register Set Selected
BRK / RESET	0
IRQ 1	4
IRQ 2	8
IRQ 3	12
IRQ 4	16
IRQ 5	20
IRQ 6	24
IRQ 7	28
Normal Operations	according to rs field in control reg #0

There is just a single set of vector registers.

Register Usage Convention

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-r10	temporaries	caller
r11-r17	register variables	callee
r18-r22	function arguments	caller
r23	assembler usage	
r24	type number / function argument	caller
r25	class pointer / function argument	caller
r26	thread pointer	callee
r27	global pointer	
r28	exception link register	caller
r29	return address / link register	caller
r30	base / frame pointer	callee
r31	stack pointer (hardware)	callee

The ISA supports up to 32 vector registers of length 63. 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. In order to get byte write strobes for the registers it was possible to accommodate a large number of registers. Elucidating, the block rams in use provided 4096 eight-bit wide registers per block ram. Regardless of the number of registers actually used there was still a provision for 4096. All these available registers were put to good use as multiple register sets and vectored registers.

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

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

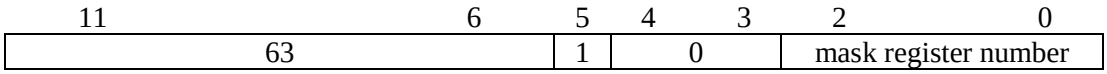
11 6	5	4 0
Register Set	0	General Purpose Register number
Vector element	1	Vector register number

To conserve hardware which would otherwise be quite large, the bypassing logic looks at only the six least significant bits, 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.

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

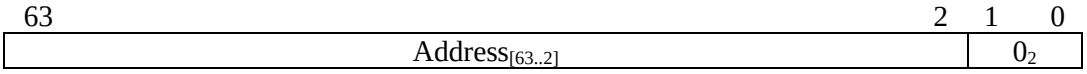


This is hidden from the ISA and may be implemented differently in the future.

On reset register set #0 is selected.

Program Counter

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



Notes:

There are actually two program counters in use by the core, one for each fetch buffer, and each one normally increments by eight. The second program counter always follows the first one, incremented by four, so that it addresses the next instruction word. There are a couple of reasons to use two counters. One is to avoid an adder delay that would be present on the output of a single counter if only one counter were used. A second reason for two counters is that they may be used independently for simultaneous multi-threading (SMT). When SMT is on each program counter operates independently and increments by four instead of eight.

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.

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.

Vector Chaining

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.

Caches

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

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

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

Uncached Data Area

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

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.

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. A statically predicted branch does not use the branch predictor instead the prediction is based on the setting of the prediction bits in the branch instruction.

The CC64 compiler has a notation for representing static branch predictions in high level code. Refer to the CC64 compiler documentation for the exact notation used.

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 single cycle operation. During execution of the branch instruction the address of the following instruction queued is checked against the address depending on the branch outcome. If the address does not match what is expected then the queue will be flushed and new instructions loaded from the correct program path.

Branch Target Buffer (BTB)

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

Return Address Stack Predictor (RSB)

There is an address predictor for return addresses which can in some cases can eliminate the flushing of the instruction queue when a return instruction is executed. The RET instruction is detected in the fetch stage of the core and a predicted return address used to fetch instructions following the return. JAL instructions using the link register as the source are also treated as return instructions. The return address stack predictor has a stack depth of 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.

Instruction Queue

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 core will not enqueue an instruction unless there is room for two or more instructions in the queue. It will not enqueue two instructions unless there is room for three or more instructions in the queue. The core waits for an additional queue slot to become available in order to prevent the core from becoming deadlocked by a flow control instruction which waits until the next instruction queues before being issued.

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.

Operating Levels

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

Operating Level	Privilege Level	Moniker

7	7 to 255	user
6	6	supervisor
5	5	supervisor
4	4	supervisor
3	3	supervisor
2	2	supervisor
1	1	hypervisor
0	0	machine

Switching Operating Levels

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

Privilege Levels

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

Control and Status Registers

Control Register Zero (CSR #000)

This register contains a bit to enable protected mode.

63	62			33	32	30	17	16	1514	13 8	7 1	0
D	~			~	bpe	dce	SNR	SMT	0	~		Pe

D: debug mode status. this bit is set during an interrupt routine if the processor was in debug mode when the interrupt occurred.
PE: Protected Mode enable: 1 = enabled, 0 = disabled.
DCE: data cache enable: 1=enabled, 0 = disabled
bpe: branch predictor enable: 1=enabled, 0=disabled
SMT: simultaneous multi-threading enable 1 = enabled, 0 = disabled (0 default).
SNR: sequence number reset, 1 = reset, automatically clears

Disabling the data cache is useful for some codes with large data sets to prevent cache loading of values that are used infrequently. The instruction cache may not be disabled.
Disabling branch prediction will significantly affect the cores performance, but may be useful for debugging. Disabling branch prediction causes all branches to be predicted as not-taken (unless determined otherwise by the instruction). No entries will be updated in the branch history table if the branch predictor is disabled.
This register supports bit set / clear CSR instructions.

HARTID (0x001)

This register contains a number that is externally supplied on the hartid_i input bus to represent the hardware thread id or the core number.

TICK (0x002)

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

PCR Paging Control (CSR 0x003)

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

AEC Arithmetic Exception Control (CSR 0x004)

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

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

CAUSE (0x006)

This register contains a code indicating the cause of an exception or interrupt. The break handler will examine this code in order to determine what to do. Only the low order 16 bits are implemented. The high order bits read as zero and are not updateable.

BADADDR (CSR 0x007)

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

PCR2 Paging Control (CSR 0x008)

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

Scratch (CSR 0x009)

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

SEMA (CSR 0x00C) Semaphores

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

SBL (CSR 0x00E)

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

SBU (CSR 0x00F)

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

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

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

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

5	**X1T**	NaNCmp	- comparison of NaN not using unordered comparison instructions
4	**X1T**	infzero	- multiply infinity by zero
3	**X1T**	zerozero	- division of zero by zero
2	**X1T**	infdiv	- division of infinities
1	**X1T**	subinfx	- subtraction of infinities
0	**X1T**	snanx	- signaling NaN

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

These registers contain addresses of instruction or data breakpoints.

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

DBCR (CSR 0x01C) Debug Control Register

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

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

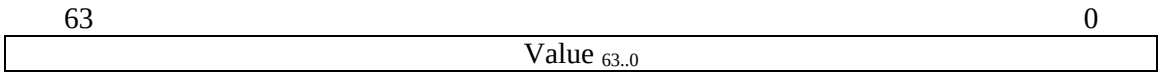
DBSR (CSR 0x01D) - Debug Status Register

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

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

CAS (CSR 0x02C) Compare and Swap

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



TVEC (0x030 to 0x037)

These registers contain the address of the exception handling routine for a given operating level. TVEC[0] (0x030) is used directly by hardware to form an address of the interrupt routine. The lower eight bits of TVEC[0] are not used. The lower bits of the interrupt address are determined from the operating level. For the other registers the two low order bits of the address must be zero in order to keep the program counter aligned on a half-word address. TVEC[1] to TVEC[7] are used by the REX instruction.

IM_STACK (0x040)

This register contains the interrupt mask stack. When an exception or interrupt occurs this register is shifted to the left and the current status copied to the low order bits, when an RTI instruction is executed this register is shifted to the right and the status bits copied from the low order bits of the register. On RTI the last stack entry is set to seven masking all interrupts on stack underflow. Only the low order 24 bits of the register are implemented.

OL_STACK (0x041)

This register contains the operating level stack. When an exception or interrupt occurs this register is shifted to the left and the current status copied to the low order bits, when an RTI instruction is executed this register is shifted to the right and the status bits copied from the low order bits of the register. On RTI the last stack entry is set to zero which will select the machine operating level on stack underflow. Only the low order 24 bits of the register are implemented.

PL_STACK (0x042)

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

RS_STACK (0x043)

This register contains the register set selection stack. When an exception or interrupt occurs this register is shifted to the left and the current status copied to the low order bits, when an RTI instruction is executed this register is shifted to the right and the status bits copied from the low

order bits of the register. On RTI the last stack entry will be set to eight which will select register set #8 on stack underflow.

STATUS (0x044)

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

63	6261	60 56	55	5452	5150	4948	47 32	27 24	2320	19 14	13 6	5 3	2 0
SD_1	$\sim_2$	VM_5	$MPRV_1$	$\sim_3$	XS_2	FS_2	$\sim_{16}$	$Thrd_1$	$\sim_4$	RS_6	PL_8	OL_3	IM_3

VM_5

These bits control virtual memory options. Note that multiple options may be present at the same time. At reset all the bits are set to zero.

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

MPRV

This bit when true (1) causes memory operations to use the first stack privilege level when evaluating privilege and protection rules. (Bits 0 to 13 in the status reg).

FS_2

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

XS_2

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

$Thrd_1$

The currently executing hardware thread.

IRQ[42..40]

The level of interrupt that caused the hardware BRK.

VCA

(bit 32) This bit indicates that vector chaining was active prior to an exception.

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

EPC (0x048 to 0x4F)

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

CODEBUF (0x080 to 0x0BF)

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

TIME (0x7E0)

The TIME register corresponds to the wall clock real time. This register can be used to compute the current time based on a known reference point. The register value will typically be a fixed number of seconds offset from the real wall clock time. The lower 32 bits of the register are driven by the tm_clk_i clock time base input which is independent of the cpu clock. The tm_clk_i input is a fixed frequency used for timing that cannot be less than 10MHz. The low order 32 bits represent the fraction of one second. The upper 32 bits represent seconds passed. For example if the tm_clk_i frequency is 100MHz the low order 32 bits should count from 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 (0x7E1)

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

INFO (0x7F0 to 0x7FF)

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

Exceptions

External Interrupts

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

Effect on Machine Status

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

For a hardware interrupt the register set is set to the level of the hardware interrupt (0 to 7). For a software exception register set #8 is selected. Individual registers from alternate register sets may be selected with the 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 eight entries, one for each operating level the core may be running at. The location of the vector table is determined by TVEC[0]. If the core is operating at level four for instance and an interrupt occurs vector table address number four is used for the interrupt handler. Note that the interrupt automatically switches the core to operating level zero, privilege level zero. An exception handler at the machine level may redirect exceptions to a lower level handler identified in one of the vector registers. More specific exception information is supplied in the cause register.

Operating Level	Address (If TVEC[0] contains $FFFC0000)	
0	$FFFC0000	Handler for operating level zero interrupt
1	$FFFC0020	
2	$FFFC0040	
3	$FFFC0060	
4	$FFFC0080	
5	$FFFC00A0	
6	$FFFC00C0	
7	$FFFC00E0	handler for operating level seven interrupt

Reset

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

Exception Cause Codes

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

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

Simplified Paged Memory Management Unit

Overview

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 64kB or 4MiB pages depending on the setting in PCR2.

64kiB pages

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

4MiB pages

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

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

Map Tables

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

Map Caching / TLB

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

Operate Key

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

Access Key

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

Address Pass-through

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

Mapping Table Layout

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

PAnn = physical address
X = executable page indicator.
W = writeable data page indicator.
R = readable data page indicator.
Note the low order six bits are not used for 4MiB pages.
S1,S0 = two bits for program use

PCR- Paging Control Register Layout

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

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

PCR2 – Page Size

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

Latency

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

Instruction Set Description

Formats

Instructions have a fixed 32 bit format. There are only a handful of different instruction formats. The opcode, Ra, Rb, and Rc 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. The Rt field is allowed to float around to make the instruction encoding easier. In a pipelined processor there is usually at least one clock cycle before Rt is used meaning it has time to be shifted around before it's use.

$Immed_{16}$				Rt_5	Ra_5	$Opcode_6$	RI
$Funct_4$	$Immed_{12}$			Rt_5	Ra_5	$Opcode_6$	RI12
$Funct_6$	$\sim_2$	Sz_3	Rt_5	Rb_5	Ra_5	$Opcode_6$	RR
01_6	$\sim_2$	Sz_3	$Funct_5$	Rt_5	Ra_5	$Opcode_6$	R1
$Funct_6$	$Funct_4$	E	Rt_5	Rb_5	Ra_5	$Opcode_6$	SR
$Funct_6$	$Funct_4$	$Immed_6$		Rt_5	Ra_5	$Opcode_6$	SI
$Funct_4$	Me_6	Mb_6		Rt_5	Ra_5	$Opcode_6$	BF
$Disp_{10}$		P_2	$Cond_4$	Rb_5	Ra_5	$Opcode_5$ D	BD
$Disp_{10}$		P_2	$Cond_3$	$Bitno_6$	Ra_5	$Opcode_5$ D	BB
$\sim_5$	P_2	$Cond_4$	Rc_5	Rb_5	Ra_5	$Opcode_6$	BR
$Funct_6$	Fn_3	Sc_2	Rt_5/Rc_5	Rb_5	Ra_5	$Opcode_6$	MX
Op_2 OL_3	$Regno_{11}$			Rt_5	Ra_5	$Opcode_6$	CSR
$Address_{26}$						$Opcode_6$	JC
$Funct_6$	$Prec_2$	Rm_3	Rt_5	Rb_5	Ra_5	$Opcode_6$	FLT

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

Format	Instruction Group
RI	register-immediate and load / store with displacement
RI12	register-immediate 12, set and load volatile instructions.
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
BR	branch to register
MX	memory indexed
CSR	control and status register access
JC	jump and call
FLT	floating-point

Operation Sizes

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

SIMD

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

Arithmetic Operations

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

Logical Operations

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

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.

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.

Call

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.

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 a 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	$\sim_2$	Sz_3	4_5	Rt_5	Ra_5	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

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

Exceptions: none

Notes:

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

ADD - Addition

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$04h_6$

04_6	$\sim_1$	Ov	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

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

Overflow works properly only on 64 bit values.

Instruction Format:

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

$Immed_{16}$	Rt_5	1_3	Q_2	$1Ah_6$

Q_2	Bits
0	0 to 15
1	16 to 31
2	32 to 47
3	48 to 63

Clock Cycles: 0.5

Execution Units: All ALU’s

Exceptions:

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

Notes:

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

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

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

AMO – Atomic Memory Operation

Description:

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

Instruction Format:

$Funct_6$	A	R	Sz_3	Rt_5	Rb_5	Ra_5	$2Fh_6$

Instruction Format (immediate operand):

$Funct_6$	A	R	Sz_3	Imm_5	Rt_5	Ra_5	$2Fh_6$

$Funct_6$	Mnemonic	Operation Performed	
01	swap	swap	memory[Ra] = Rb
04	add	addition	memory[Ra] = memory[Ra] + Rb
08	and	bitwise and	memory[Ra] = memory[Ra] & Rb
09	or	bitwise or	memory[Ra] = memory[Ra] | Rb
0A	xor	bitwise exclusive or	memory[Ra] = memory[Ra] ^ Rb
0C	shl	shift left	memory[Ra] = memory[Ra] << Rb
0D	shr	shift right	memory[Ra] = memory[Ra] >> Rb
1C	min	minimum	memory[Ra] = memory[Ra] < Rb ? memory[Ra] : Rb
1D	max	maximum	memory[Ra] = memory[Ra] >Rb ? memory[Ra] : Rb
1E	minu	minimum unsigned	memory[Ra] = memory[Ra] < Rb ? memory[Ra] : Rb
1F	maxu	maximum unsigned	memory[Ra] = memory[Ra] > Rb ? memory[Ra] : Rb
20	swapi	swap	memory[Ra] = imm
24	addi	addition	memory[Ra] = memory[Ra] + imm
28	andi	bitwise and	memory[Ra] = memory[Ra] & imm
29	ori	bitwise or	memory[Ra] = memory[Ra] | imm
2A	xori	bitwise exclusive or	memory[Ra] = memory[Ra] ^ imm
2C	shli	shift left	memory[Ra] = memory[Ra] << imm
2D	shri	shift right	memory[Ra] = memory[Ra] >> imm
3C	mini	minimum	memory[Ra] = memory[Ra] < imm ? memory[Ra] : imm

3D	maxi	maximum	memory[Ra] = memory[Ra] > imm ? memory[Ra] : imm
3E	minui	minimum	memory[Ra] = memory[Ra] < imm ? memory[Ra] : imm
3F	maxui	maximum	memory[Ra] = memory[Ra] > imm ? memory[Ra] : imm

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.

Instruction Format:

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

$Immed_{16}$	Rt_5	Ra_5	$08h_6$

Rt = Ra & Rb

08_6	$\sim_2$	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

Instruction Format:

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

$Immed_{16}$	Rt_5	2_3	Q_2	$3Bh_6$

Q_2	Bits
0	0 to 15
1	16 to 31
2	32 to 47
3	48 to 63

Clock Cycles: 0.5

Execution Units: All ALUs

Exceptions: none

ASL – Arithmetic Shift Left

Description:

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

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

Instruction Format:

$Func_6$	2_4	E	Rt_5	Rb_5	Ra_5	$02h_6$

$Func_6$	Ah_4	Imm_6	Rt_5	Ra_5	$02h_6$

$Func_6$	Op Size	If E set
0Fh	word	word
1Fh	byte	byte parallel
2Fh	char	char parallel
3Fh	half	half parallel

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions:

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

ASR – Arithmetic Shift Right

Description:

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

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

Instruction Format:

$Func_6$	3_4	E	Rt_5	Rb_5	Ra_5	$02h_6$

$Func_6$	Bh_4	Imm_6	Rt_5	Ra_5	$02h_6$

$Func_6$	Op Size	If E set
0Fh	word	word
1Fh	byte	byte parallel
2Fh	char	char parallel
3Fh	half	half parallel

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

BBC –Branch if Bit Clear

Description:

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

Instruction Format:

31 22	21	19	11	10 6	5 1	0
$Displacement_{10}$	P_2	1_3	$Bitno_6$	Ra_5	$13h_5$	D_1

Operation:

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

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

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

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

Execution Units: FCU Only

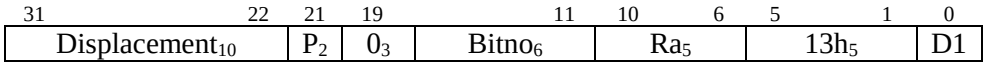
Exceptions: none

BBS –Branch if Bit Set

Description:

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

Instruction Format:



Operation:

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

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

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

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

Execution Units: FCU Only

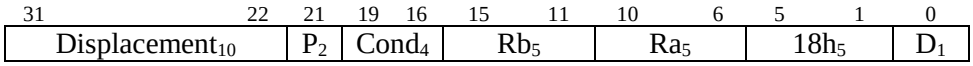
Exceptions: none

Bcc – Conditional Branch

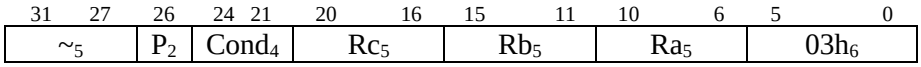
Description:

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

Instruction Format:



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



$Cond_4$	Mne.	
0	BEQ	Ra = Rb signed
1	BNE	Ra <> Rb
2	BLT	Ra < Rb
3	BGE	Ra >= Rb
4	BLTU	Ra < Rb (unsigned)
5	BGEU	Ra >= Rb (unsigned)
6		reserved
7	BOR	Ra || Rb (either Ra or Rb is true)
8	FBEQ	Ra = Rb (floating point)
9	FBNE	Ra != Rb (floating point)
10	FBLT	Ra < Rb (floating point)
11	FBGE	Ra >= Rb (floating point)
12		reserved
13		reserved
14		reserved
15	FBUN	register Ra contains unordered floating point constant

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

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

Clock Cycles:

Typically 1 with correct branch outcome and target prediction.

BCDADD - Register-Register

Description:

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

Instruction Format:

00_6	0_5	Rt_5	Rb_5	Ra_5	$02h_6$

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_5	Rt_5	Rb_5	Ra_5	$02h_6$

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_5	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 only

Operation:

Rt = Ra - Rb

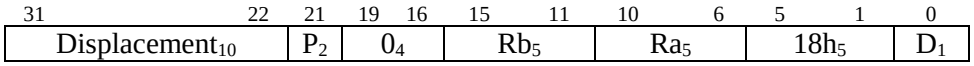
Exceptions: none

BEQ –Branch if Equal

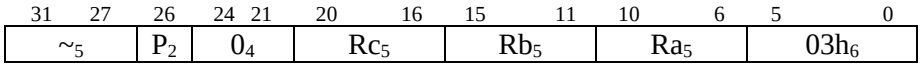
Description:

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

Instruction Format:



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



Operation:

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

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

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

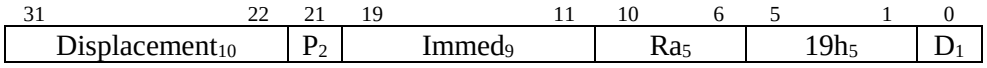
Clock Cycles: Typically 1 with correct branch outcome and target prediction.

BEQI –Branch if Equal Immediate

Description:

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

Instruction Format:



Operation:

```
if (Ra = Immediate)
        pc = pc + displacement
```

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

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

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

BFCHG – Bitfield Change

Description:

A bitfield is inverted in the target register.

Instruction Format:

2_4	Me_6	Mb_6	Rt_5	Ra_5	$22h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

BFCLR – Bitfield Clear

Description:

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

Instruction Format:

3_4	Me_6	Mb_6	Rt_5	0_5	$22h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

BFEXT – Bitfield Extract

Description:

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

Instruction Format:

5_4	Me_6	Mb_6	Rt_5	Ra_5	$22h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

BFEXTU – Bitfield Extract

Description:

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

Instruction Format:

6_4	Me_6	Mb_6	Rt_5	Ra_5	$22h_6$

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	Me_6	Mb_6	Rt_5	Ra_5	$22h_6$

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	Me_6	Mb_6	Rt_5	Imm_5	$22h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

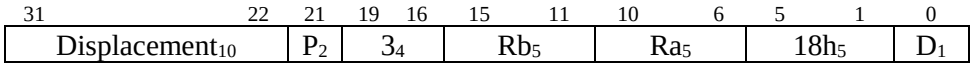
Exceptions: none

BGE –Branch if Greater or Equal

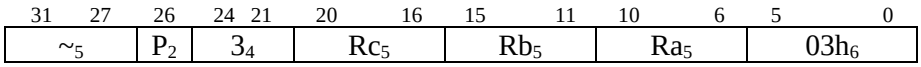
Description:

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

Instruction Format:



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



Operation:

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

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

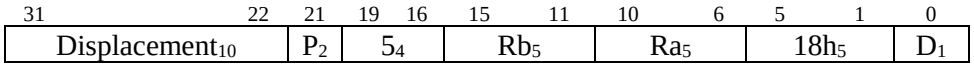
If a branch prediction is supplied, then the branch instruction doesn't occupy room in the history tables.

BGEU –Branch if Greater or Equal Unsigned

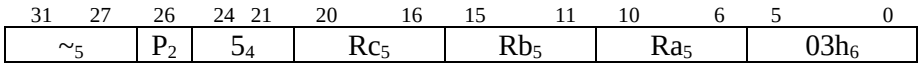
Description:

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

Instruction Format:



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



Operation:

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

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

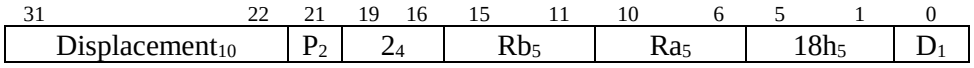
If a branch prediction is supplied, then the branch instruction doesn't occupy room in the history tables.

BLT –Branch if Less Than

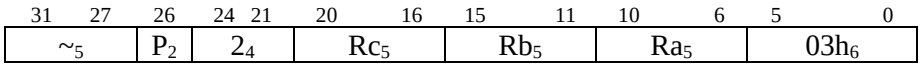
Description:

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

Instruction Format:



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



Operation:

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

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

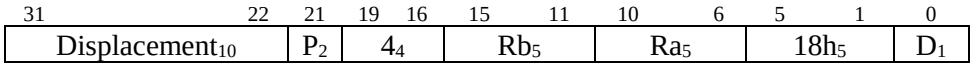
If a branch prediction is supplied, then the branch instruction doesn't occupy room in the history tables.

BLTU –Branch if Less Than Unsigned

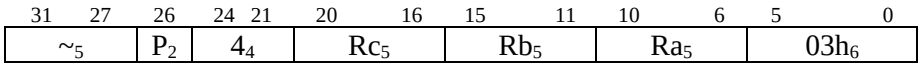
Description:

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

Instruction Format:



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



Operation:

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

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

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

BMM – Bit Matrix Multiply

BMM Rt, Ra, Rb

Description:

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

Instruction Format:

03_6	$\sim_3$	$\sim_2$	Rt_5	Rb_5	Ra_5	$02h_6$

Operation:

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

Clock Cycles: 1

Execution Units: ALU #0 only

Exceptions: none

Notes:

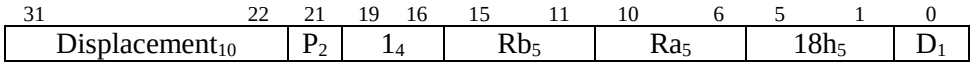
The bits are numbered with bit 63 of a register representing I,j = 0,0 and bit 0 of the register representing I,j = 7,7.

BNE –Branch if Not Equal

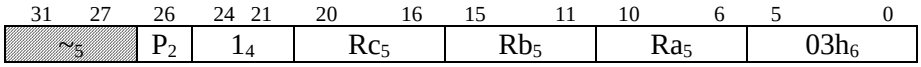
Description:

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

Instruction Format:



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



Operation:

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

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

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

BRK – Hardware / Software Breakpoint

Description:

Invoke the break handler routine. The break handler routine handles all the hardware and software exceptions in the core. A cause code is loaded into the CAUSE CSR register. The break handler should read the CAUSE code to determine what to do. The break handler is located by TVEC[0]. This address should contain a jump to the break handler. Note the reset address is $FFFC0100. An exception will automatically switch the processor to the machine level operating mode. The break handler routine may redirect the exception to a lower level using the REX instruction.

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

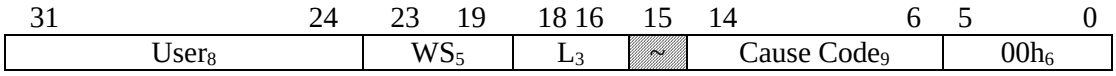
The core maintains an internal eight level interrupt stack for each of the following:

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:



WS = word skip 1 = software interrupt – return address is next instruction

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

L_3 = 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 $User_8$ field may be used to pass constant data to the break handler.

CACHE – Cache Command

CACHEX –

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

Description:

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

Instruction Formats:

$Displacement_{16}$				Cmd_5	Ra_5	$1Eh_6$	CACHE Cmd,d16(Rn)
$1Eh_6$	$\sim_3$	Sc_2	Cmd_5	Rb_5	Ra_5	$02h_6$	CACHE Cmd,d(Ra+Rb*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 + Rb * 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 Subroutine

Description:

Call a subroutine. This instruction is a longer address form than the JAL instruction and has the link register as an implied target for the return address. This is the preferred method to call a subroutine. If a larger address range is required then the address must be loaded into a register and the JAL instruction used.

Instruction Format:

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

$Address_{[27..2]}$	$19h_6$

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

Execution Units: FCU

Clock Cycles: 1

Exceptions: none

Notes:

There is no need for the instruction queue to flush as the address is entirely determined during the fetch stage.

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:

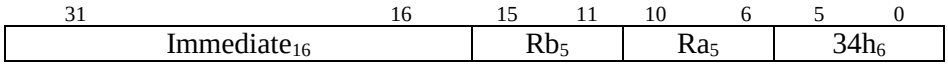
CAS Rt, displacement[Ra]

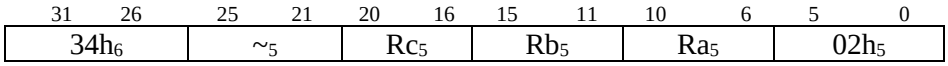
CHK – Check Register Against Bounds

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:





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	$\sim_7$	0_3	$\sim_5$	0_5	$02h_6$

Clock Cycles: 0.5

CMOVEQ – Conditional Move Equal

Description:

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

Instruction Format:

$28h_6$	Rt_5	Rc_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 0.5

CMOVNE – Conditional Move Not Equal

Description:

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

Instruction Format:

$29h_6$	Rt_5	Rc_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 0.5

CMP – Signed Comparison

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$06h_6$

$06h_6$	Cnd_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Parallel Operand (SIMD) compare

$19h_6$	Cnd_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 0.5

Sz_2	
0	Byte
1	Char
2	Half
3	Word

Cnd_3	
0	CMP
2	SEQ
3	SNE
4	SLT
5	SGE
6	SLE
7	SGT

CMPU – Unsigned Comparison

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$07h_6$

$07h_6$	Cnd_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Parallel Operand (SIMD) compare

$1Ah_6$	Cnd_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 0.5

Sz_2	
0	Byte
1	Char
2	Half
3	Word

Cnd_3	
0	CMPU
4	SLTU
5	SGEU
6	SLEU
7	SGTU

CNTLO – Count Leading Ones

Description:

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

Instruction Format:

01_6	$\sim_3$	Sz_2	1_5	Rt_5	Ra_5	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

Sz_2	
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	$\sim_3$	Sz_2	0_5	Rt_5	Ra_5	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

Sz_2	
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	$\sim_3$	Sz_2	2_5	Rt_5	Ra_5	$02h_6$

Clock Cycles: 1

Execution Units: ALU #0 Only

Exceptions: none

Sz_2	
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_3	$Regno_{11}$	Rt_5	Ra_5	$0Eh_6$

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_3 field is reserved to specify the operating level. Note that registers cannot be accessed by a lower operating level.

$Regno_{12}$		Access	Description
001	HARTID	R	hardware thread identifier (core number)
002	TICK	R	tick count, counts every cycle from reset
030-037	TVEC	RW	trap vector handler address
040	EPC	RW	exceptioned pc, pc value at point of exception
044	STATUSL	RWSC	status register, contains interrupt mask, operating level
045	STATUSH	RW	status register bits 64 to 127
080-0BF	CODE	RW	code buffers
7F0	INFO	R	Manufacturer name
7F1	“	R	“
7F2	“	R	cpu class
7F3	“	R	“
7F4	“	R	cpu name
7F5	“	R	“
7F6	“	R	model number
7F7	“	R	serial number
7F8	“	R	cache sizes instruction (bits 32 to 63), data (bits 0 to 31)

Clock Cycles: 0.5

DBNZ –Decrement, Branch if Not Zero

Description:

If the specified register is non-zero then an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. The register is also decremented by one.

Instruction Format:

31 22	21 19	11	10 6	5 1	0
$Displacement_{10}$	P_2	7_3	0_6	Ra_5	$13h_5$	D1

Operation:

```
if (Ra<>0)
        pc = pc + displacement
Ra = Ra - 1
```

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

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

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

Execution Units: FCU Only

Exceptions: none

DIV – Signed Division

Description:

Compute the quotient. The first operand must be in a register. The second operand may be in either a register or an immediate value specified in the instruction. The operands are treated as signed values and the result is a signed result. Note that for the registered form of the instruction both the quotient and remainder may be calculated at the same time.

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$3Eh_6$

Return quotient

$3E_6$	0_2	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

Return remainder

$3E_6$	1_2	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

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.

DIVSU – Signed-Unsigned Division

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$3Dh_6$

Return quotient

$3Dh_6$	0_2	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

Return remainder

$3Dh_6$	1_2	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

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

ALU Support: ALU #0 Only

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

DIVU – Unsigned Division

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$3Ch_6$

Return quotient

$3Ch_6$	0_2	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

Return remainder

$3Ch_6$	1_2	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

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

ALU Support: ALU #0 Only

Exceptions: none

EXEC – Execute Code Buffer

Description:

> Execute code from code buffer. The 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:

$\sim_{10}$	N_6	$\sim_5$	$\sim_5$	$1Fh_6$

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

JAL – Jump-And-Link

Description:

Instruction Format:

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

$Immed_{16}$	Rt_5	Ra_5	$18h_6$

Execution Units: FCU

Clock Cycles:

JMP – Jump to Address

Description:

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

Instruction Format:

$Address_{[27..2]}$	$28h_6$

Execution Units: FCU

Clock Cycles: 1

Exceptions: none

Notes:

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

LB – Load Byte

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$13h_6$

$13h_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 4 minimum depending on memory access time

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

LBO – Load Byte Only

Description:

This instruction loads a byte (8 bit) value from memory. Only the lower eight bits of the target register are updated, the upper bits of the register are not affected. This instruction may be used to perform unaligned memory loads when combined with a shift instruction.

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$2Ah_6$

$2Ah_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

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 Format:

$Immed_{16}$	Rt_5	Ra_5	$23h_6$

$23h_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

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:

$Immed_{16}$	Rt_5	Ra_5	$20h_6$

$20h_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 4 minimum depending on memory access time

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

LCO – Load Char Only (16 bits)

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$2Bh_6$

$2Bh_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

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.

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$21h_6$

$21h_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 4 minimum depending on memory access time

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

LDI – Load Immediate

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	0_5	$09h_6$

Clock Cycles: 0.5

LEAX – Load Effective Address

Description:

This instruction loads an address value into a register.

Instruction Format:

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

$Immed_{16}$	Rt_5	Ra_5	$04h_6$

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.

$18h_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 0.5

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:

$Immed_{16}$	Rt_5	Ra_5	$10h_6$

$10h_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 4 minimum depending on memory access time

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

LHO – Load Half-Word Only (32 bits)

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$35h_6$

$35h_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

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:

$Immed_{16}$	Rt_5	Ra_5	$11h_6$

$11h_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 4 minimum depending on memory access time

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

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.

Instruction Format:

0_4	$Immed_{12}$	Rt_5	Ra_5	$3Bh_6$

$3Bh_6$	0_3	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

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.

Instruction Format:

1_4	$Immed_{12}$	Rt_5	Ra_5	$3Bh_6$

$3Bh_6$	1_3	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

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:

2_4	$Immed_{12}$	Rt_5	Ra_5	$3Bh_6$

$3Bh_6$	2_3	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

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:

6_4	$Immed_{12}$	Rt_5	Ra_5	$3Bh_6$

$3Bh_6$	6_3	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

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:

$Immed_{16}$	Rt_5	Ra_5	$12h_6$

$12h_6$	$\sim_3$	Sc_2	Rt_5	Rb_5	Ra_5	$02h_6$

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:

$Immed_{16}$	Rt_5	Ra_5	$1Dh_6$

$1Dh_6$	A	R	Sc_3	Rt_5	Rb_5	Ra_5	$02h_6$

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:

$2Eh_6$	Rt_5	Rc_5	Rb_5	Ra_5	$02h_6$

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.

Instruction Format:

$2Dh_6$	$\sim_2$	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

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:

$2Ch_6$	$\sim_2$	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

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:

$Immed_{16}$	Rt_5	Ra_5	$2Eh_6$

Return remainder

$3E_6$	1_2	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

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.

MODSU – Signed-Unsigned Modulus

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$2Dh_6$

Return remainder

$3Dh_6$	1_2	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$2Ch_6$

Return remainder

$3Ch_6$	1_2	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

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.

Instruction Format:

$22h_6$	D_3	$\sim_1$	Rgs_6	Rt_5	Ra_5	$02h_6$

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$3Ah_6$

Multiply, return low order product

$3Ah_6$	0_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Multiply, return high order product

$3Ah_6$	1_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 19

MULSU – Signed-Unsigned Multiply

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$39h_6$

Multiply, return low order product

$39h_6$	0_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Multiply, return high order product

$39h_6$	1_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 19

MULU – Unsigned Multiply

Description:

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

Instruction Format:

$Immed_{16}$	Rt_5	Ra_5	$38h_6$

Multiply, return low order product

$38h_6$	0_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Multiply, return high order product

$38h_6$	1_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 19

Exceptions: none

MUX – Multiplex

Description:

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

Instruction Format:

$1Bh_6$	Rt_5	Rc_5	Rb_5	Ra_5	$02h_6$

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:

$0C_6$	$\sim_2$	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 0.5

Exceptions: none

NEG - Negate

Description:

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

Instruction Format:

05_6	$\sim_2$	Sz_3	Rt_5	Rb_5	0_5	$02h_6$

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.

Instruction Format:

$Immediate_{26}$	$1Ch_6$

NOR – Bitwise Nor

Description:

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

Instruction Format:

$0D_6$	$\sim_3$	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 0.5

Exceptions: none

OR – Bitwise Or

Description:

Perform a bitwise or operation between operands.

Instruction Format:

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

Rt = Ra | immed

$Immed_{16}$	Rt_5	Ra_5	$09h_6$

Rt = Ra | Rb

09_6	$\sim_2$	Sz_3	Rt_5	Rb_5	Ra_5	$02h_6$

Instruction Format:

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

$Immed_{16}$	Rt_5	0_3	Q_2	$1Ah_6$

Q_2	Bits
0	0 to 15
1	16 to 31
2	32 to 47
3	48 to 63

Clock Cycles: 0.5

Execution Units: All ALUs

Exceptions: none

RET – Return from Subroutine

Description:

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

Instruction Format:

$Immed_{16}$	$1Dh_5$	$1Fh_5$	$29h_6$

PC = RA
SP = SP + Immediate

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.

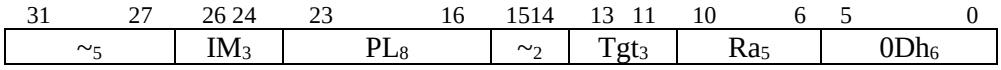
The location of the target exception handler is found in the trap vector register for that operating level (tvec[xx]).

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:



Tgt_3	
0	not used
1	redirect to hypervisor level
2	redirect to supervisor level
3	redirect to supervisor level
4	redirect to supervisor level
5	redirect to supervisor level
6	redirect to supervisor level
7	not used

Clock Cycles: 3

Example:

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

Notes:

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

ROL – Rotate Left

Description:

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

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

Instruction Format:

$Func_6$	4_4	E	Rt_5	Rb_5	Ra_5	$02h_6$

$Func_6$	Ch_4	Imm_6	Rt_5	Ra_5	$02h_6$

$Func_6$	Op Size	If E set
0Fh	word	word
1Fh	byte	byte parallel
2Fh	char	char parallel
3Fh	half	half parallel

Clock Cycles: 1

ALU Support: ALU #0 Only

Exceptions: none

ROR – Rotate Right

Description:

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

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

Instruction Format:

$Func_6$	5_4	E	Rt_5	Rb_5	Ra_5	$02h_6$

$Func_6$	Dh_4	Imm_6	Rt_5	Ra_5	$02h_6$

$Func_6$	Op Size	If E set
0Fh	word	word
1Fh	byte	byte parallel
2Fh	char	char parallel
3Fh	half	half parallel

Clock Cycles: 1

ALU Support: ALU #0 Only

Exceptions: none

RTI – Return from Interrupt

Description:

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

Instruction Format:

$32h_6$	$\sim_4$	$Sema_6$	$\sim_5$	Ra_5	$02h_6$

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

Clock Cycles: 8 minimum

Execution Units: Flow Control Unit

RTE – Return from Exception

Description:

This is an alternate mnemonic for the RTI instruction.

Instruction Format:

$32h_6$	$\sim_4$	$Sema_6$	$\sim_5$	Ra_5	$02h_6$

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

Clock Cycles:

SB – Store Byte

Description:

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

Instruction Format:

$Immed_{16}$	Rb_5	Ra_5	$15h_6$

$15h_6$	$\sim_3$	Sc_2	Rc_5	Rb_5	Ra_5	$02h_6$

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:

$Immed_{16}$	Rb_5	Ra_5	$24h_6$

$24h_6$	$\sim_3$	Sc_2	Rc_5	Rb_5	Ra_5	$02h_6$

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

Scc – Set

Description:

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

Instruction Format:

$Cond_4$	$Immed_{12}$	Rt_5	Ra_5	$1Bh_6$

Signed

06_6	Cnd_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Unsigned

07_6	Cnd_3	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 0.5

$Cond_3$ / $Cond_4$	
0	CMP / CMPU
2	SEQ
3	SNE
4 /12	SLT / SLTU
5 /13	SGE / SGEU
6 /14	SLE / SLEU
7 /15	SGT / SGTU

Sz_2	
0	Byte
1	Char
2	Half
3	Word

SEI – Set Interrupt Mask

SEI #3

Description:

The interrupt level mask is set to the value specified by the instruction. The value used is the bitwise or of the contents of register Ra and an immediate (M_3) supplied in the instruction. The assembler assumes a mask value of seven, masking all interrupts, if no mask value is specified. Usually either M_3 or Ra should be zero.

Instruction Format:

30_6	$\sim_4$	$\sim_3$	M_3	$\sim_5$	Ra_5	$02h_6$

Operation:

im = M_3 | Ra

SGN – Get Sign

Description:

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

Instruction Format:

06_6	$\sim_3$	Sz_2	Rt_5	0_5	Ra_5	$02h_6$

Clock Cycles: 0.5

Sz_2	
0	Byte
1	Char
2	Half
3	Word

SH – Store Half-Word (32 bits)

Description:

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

Instruction Format:

$Immed_{16}$	Rb_5	Ra_5	$14h_6$

$14h_6$	$\sim_3$	Sc_2	Rc_5	Rb_5	Ra_5	$02h_6$

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:

$Func_6$	0_4	E	Rt_5	Rb_5	Ra_5	$02h_6$

E indicates to update all lanes of target register.

$Func_6$	8_4	Imm_6	Rt_5	Ra_5	$02h_6$

$Func_6$	Op Size	If E set
0Fh	word	word
1Fh	byte	byte parallel
2Fh	char	char parallel
3Fh	half	half parallel

Clock Cycles: 1

ALU Support: ALU #0 Only

Exceptions: none

SHR – Shift Right

Description:

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

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

Instruction Format:

$Func_6$	1_4	E	Rt_5	Rb_5	Ra_5	$02h_6$

$Func_6$	9_4	Imm_6	Rt_5	Ra_5	$02h_6$

$Func_6$	Op Size	If E set
0Fh	word	word
1Fh	byte	byte parallel
2Fh	char	char parallel
3Fh	half	half parallel

Clock Cycles: 1

ALU Support: ALU #0 Only

Exceptions: none

SUB - Subtract

Description:

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

Instruction Format:

05_6	$\sim_2$	Ov	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

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

Overflow works properly only on 64 bit values.

Clock Cycles: 0.5

Exceptions:

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

SW – Store Word (64 bits)

Description:

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

Instruction Format:

$Immed_{16}$	Rb_5	Ra_5	$16h_6$

$16h_6$	$\sim_3$	Sc_2	Rc_5	Rb_5	Ra_5	$02h_6$

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:

$Immed_{16}$	Rb_5	Ra_5	$17h_6$

$17h_6$	A	R	Sc_3	Rc_5	Rb_5	Ra_5	$02h_6$

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_5	$12h_5$	$\sim_5$	$\sim_5$	$02h_6$

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.

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	$\sim_5$	Imm_5	Rb_5	Ra_5	$02h_6$

Operation:

```
if (no signal)
   delay instruction
else
   mark instruction done
```

Notes:

This instruction waits for a signal to occur before proceeding.

XNOR – Bitwise Exclusive Nor

Description:

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

Instruction Format:

$0E_6$	$\sim_3$	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Clock Cycles: 0.5

Exceptions: none

XOR – Bitwise Exclusive Or

Description:

Perform a bitwise exclusive or operation between operands.

Instruction Format:

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

$Immed_{16}$	Rt_5	Ra_5	$0Ah_6$

Rt = Ra ^ Rb ^ Rc

$0A_6$	$\sim_3$	Sz_2	Rt_5	Rb_5	Ra_5	$02h_6$

Instruction Format:

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

$Immed_{16}$	Rt_5	3_3	Q_2	$1Ah_6$

Q_2	Bits
0	0 to 15
1	16 to 31
2	32 to 47
3	48 to 63

Clock Cycles: 0.5

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 core automatically uses odd numbered register sets for the floating-point registers. For instance, if register set #16 is selected the corresponding floating-point registers are in register set #17. The floating-point registers may also be used as integer registers by selecting an odd numbered register set if floating-point is not required.

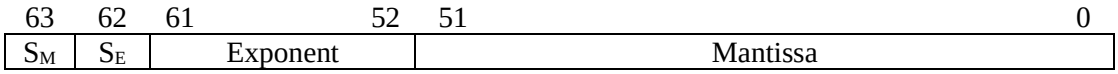
The precision field ($prec_2$) should be set to 1.

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:



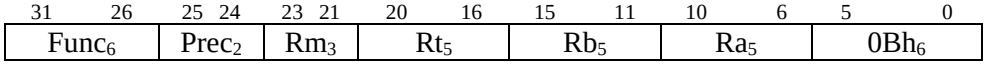
S_M – sign of mantissa
S_E – sign of exponent

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



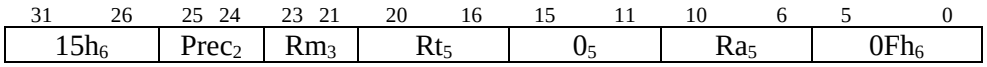
Not all instructions required the Rb_5 field. If not required Rb should be set to zero.

FABS – Floating Absolute Value

Description:

Take the absolute value of the floating-point number in 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:



Clock Cycles: 2

Execution Units: Floating Point

FADD – Floating point addition

Description:

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

Instruction Format:

31 26	25 24	23 21	20 16	15 11	10 6	5 0
04_6	$Prec_2$	Rm_3	Rt_5	Rb_5	Ra_5	$0Fh_6$

Clock Cycles: 10

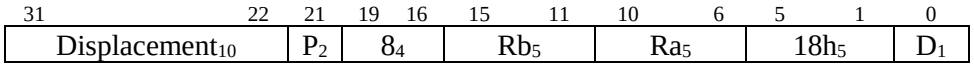
Execution Units: Floating Point

FBEQ –Branch if Equal

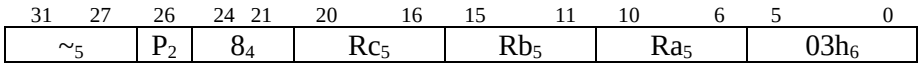
Description:

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

Instruction Format:



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



Operation:

```
if (Ra = Rb)
        pc = pc + displacement
```

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

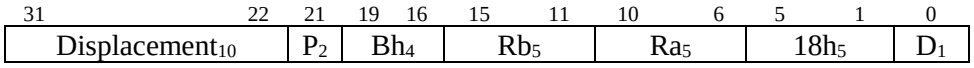
If a branch prediction is supplied, then the branch instruction doesn't occupy room in the history tables.

FBGE –Branch if Greater than or Equal

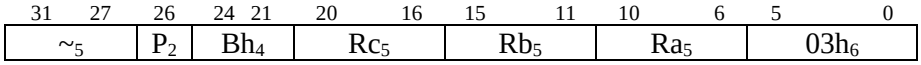
Description:

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

Instruction Format:



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



Operation:

```
if (Ra >= Rb)
        pc = pc + displacement
```

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

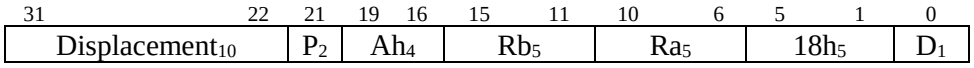
If a branch prediction is supplied, then the branch instruction doesn't occupy room in the history tables.

FBLT –Branch if Less Than

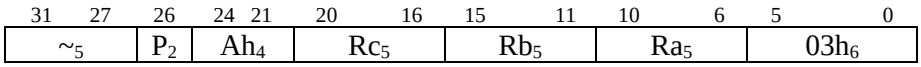
Description:

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

Instruction Format:



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



Operation:

```
if (Ra < Rb)
        pc = pc + displacement
```

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

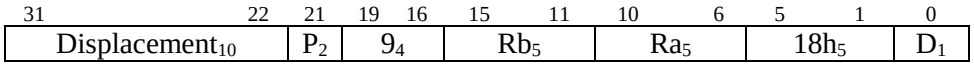
If a branch prediction is supplied, then the branch instruction doesn't occupy room in the history tables.

FBNE –Branch if Not Equal

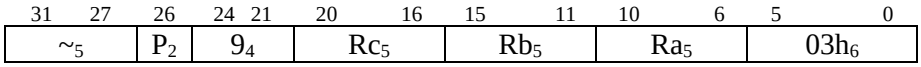
Description:

If two registers are unequal an eleven bit sign extended value is shifted left twice and added to the program counter. The branch is relative to the address of the instruction directly following the branch. Note that positive and negative zero are treated as equal.

Instruction Format:



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



Operation:

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

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

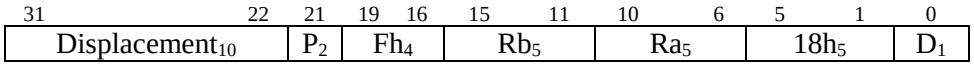
If a branch prediction is supplied, then the branch instruction doesn't occupy room in the history tables.

FBUN –Branch if Unordered

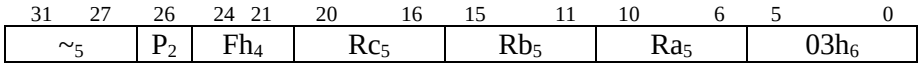
Description:

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

Instruction Format:



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



Operation:

```
if (Ra ? Rb)
        pc = pc + displacement
```

The P_2 field is reserved for branch prediction hints.

P_2	Prediction Type
0	no static prediction (use branch history)
1	reserved
2	always predict as not-taken
3	always predict as taken

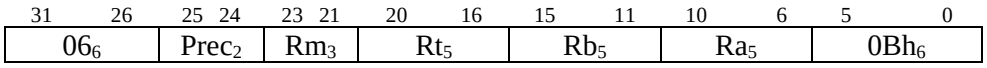
If a branch prediction is supplied, then the branch instruction doesn't occupy room in the history tables.

FCMP - Float Compare

Description:

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

Instruction Format:



Clock Cycles: 2

Execution Units: FPU

Operation:

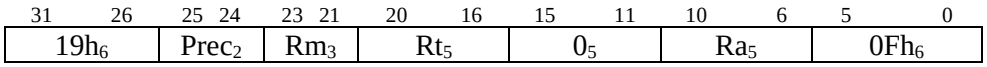
```
if Ra < Rb
        Rt[1]= true
else
        Rt[1] = false
if mag Ra < mag Rb
        Rt[2] = true
else
        Rt[2] = false
if Ra = Rb
        Rt[0] = true
else
        Rt[0] = false
if Ra <= Rb
        Rt[3] = true
else
        Rt[3] = false
if unordered
        Rt[4] = true
else
        Rt[4] = 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:



Clock Cycles: 3

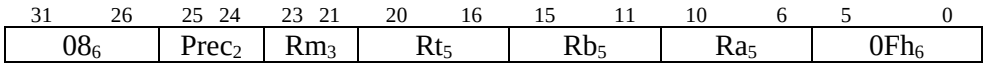
Execution Units: Floating Point

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:



Clock Cycles: 115

Execution Units: Floating Point

FCX – Clear Floating Point Exceptions

Description:

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

Instruction Format:

31 26	25 24	2322	21 16	15 11	10 6	5 0
$21h_6$	$Prec_2$	$\sim_2$	Imm_6	0_5	Ra_5	$0Fh_6$

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	2322	21 16	15 11	10 6	5 0
$23h_6$	$Prec_2$	$\sim_2$	Imm_6	0_5	Ra_5	$0Fh_6$

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	2322	21 16	15 11	10 6	5 0
$22h_6$	$Prec_2$	$\sim_2$	Imm_6	0_5	Ra_5	$0Fh_6$

Clock Cycles: 2

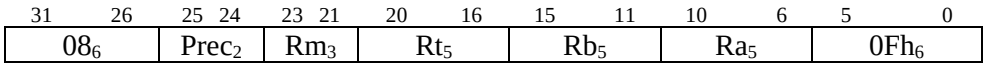
Execution Units: Floating Point

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:



Clock Cycles: 12

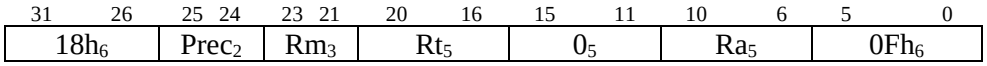
Execution Units: Floating Point

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:



Clock Cycles: 2

Execution Units: Floating Point

FNEG – Floating Negative Value

Description:

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

Instruction Format:

31 26	25 24	23 21	20 16	15 11	10 6	5 0
$14h_6$	$Prec_2$	Rm_3	Rt_5	0_5	Ra_5	$0Fh_6$

Clock Cycles: 2

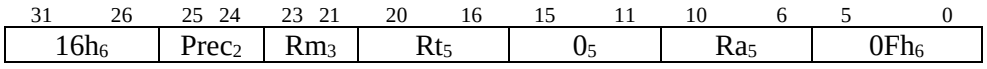
Execution Units: Floating Point

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:



Clock Cycles: 2

Execution Units: Floating Point

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.

Instruction Format:

31 26	25 24	23 21	20 16	15 11	10 6	5 0
$1Dh_6$	$Prec_2$	Rm_3	Rt_5	0_5	Ra_5	$0Fh_6$

Clock Cycles: 110

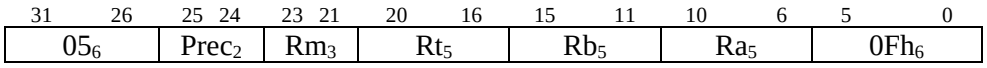
Execution Units: Floating Point

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:



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
$36h_6$	$\sim_2$	$\sim_3$	$\sim_5$	$\sim_5$	$\sim_5$	$0Fh_6$

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 24	23 21	20 16	15 11	10 6	5 0
$12h_6$	$Prec_2$	Rm_3	Rt_5	0_5	Ra_5	$0Fh_6$

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
$20h_6$	$Prec_2$	$\sim_2$	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 24	23 21	20 16	15 11	10 6	5 0
$15h_6$	$Prec_2$	Rm_3	Rt_5	0_5	Ra_5	$0Fh_6$

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.

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] = memory64[Ra + Immed + 8 * x]
        else
                NOP
```

Exceptions: DBE, DBG, LMT

LVWS – Load Vector With Stride

Synopsis

Load vector

Description:

Load a vector register from memory using indexed addressing.

Instruction Format:

$26h_6$	Vm_3	3_2	Vt_5	Rb_5	Ra_5	$02h_6$

Operation

for x = 0 to VL-1

Vt[x] = $memory_{64}$[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:

$36h_6$	$\sim_3$	3_2	Vt_5	Vb_5	Ra_5	$02h_6$

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])
                memory64[Ra + Immed + 8 * x] = Vb[x]
        else
                NOP
```

Exceptions: DBE, DBG, LMT

SVWS – Store Vector With Stride

Synopsis

Store vector

Description:

Store a vector register to memory using indexed addressing.

Instruction Format:

$27h_6$	Vm_3	3_2	Vc_5	Rb_5	Ra_5	$02h_6$

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	$02h_6$

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.

Instruction Format

04_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	

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	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]
```

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	$\sim_2$	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	$\sim_2$	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]
```

Opcode Tables

Major Opcode (inst. bits 0 to 5)

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

Major Funct (inst. bits 26 to 31)

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

Float Funct (inst. bits 26 to 31)

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

R1 (inst. bits 21 to 25)

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

Shift (inst. bits 22 to 25)

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

Vector Funct (inst. bits 26 to 31)

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

Appendix

Reducing the size of the core.

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

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

Architectural Register vs Physical Registers

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

Register Renaming

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

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

Instruction Cache Miss

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

Branches

Branches store the target address in iqentry_a0 the immediate constant field of the queue. The target address has to be stored somewhere in the instruction queue so that it may be used to update the branch target buffer later. It can't be stored in the result field, and it can't be stored in one of the other argument fields. Arg0 is the only place it can be stored safely.

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

Configuration Defines

Q2VECTORS

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

Parameters

SUP_TXE

- default 0
- enables support for the call target exception

SUP_VECTOR

- default 1
- enables support for vector instructions

-

Instructions Supported Only on ALU #0

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

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

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.

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.

WISHBONE Compatibility Datasheet

The FT64 core may be directly interfaced to a WISHBONE compatible bus.

WISHBONE Datasheet WISHBONE SoC Architecture Specification, Revision B.3		
Description:	Specifications:	
General Description:	Central processing unit (CPU core)	
Supported Cycles:	MASTER, READ / WRITE MASTER, READ-MODIFY-WRITE MASTER, BLOCK READ / WRITE, BURST READ (FIXED ADDRESS)	
Data port, size: Data port, granularity: Data port, maximum operand size: Data transfer ordering: Data transfer sequencing	64 bit 8 bit 64 bit Little Endian any (undefined)	
Clock frequency constraints:		
Supported signal list and cross reference to equivalent WISHBONE signals	Signal Name: ack_i adr_o(31:0) clk_i dat_i(63:0) dat_o(63:0) cyc_o stb_o wr_o sel_o(7:0) cti_o(2:0) bte_o(1:0)	WISHBONE Equiv. ACK_I ADR_O() CLK_I DAT_I() DAT_O() CYC_O STB_O WE_O SEL_O CTI_O BTE_O
Special Requirements:		