Gambit

Overview

Gambit is a superscalar processor with a 52-bit native operating mode. Native mode makes use of a 32-entry register file. Native mode instructions vary in length up to 52-bits. The processor manages branching using a compare instruction and compare result registers.

Motivation

Gambit’s goal is to be a cost reduced processing core. Consuming 20% fewer resources than a 64-bit core would.

Programming Model

General Registers

Gambit contains a 32-entry by 52-bits wide general-purpose integer register file. One register designation is reserved for the stack pointer. There are four stack pointers, one for each operating level.

Reg			Usage
R0	z	This register is always zero	
R1	acc	Accumulator	First parameter / return value / loop count
R2	x	‘x’ index register	Second parameter
R3	y	‘y’ index register	Third parameter
R4			
R5			
R6			
R7 to 29			
R30	fp	frame pointer	
R31	sp	stack pointer	user stack pointer
R31	ssp		supervisor stack pointer
R31	hsp		hypervisor stack pointer
R31	msp		machine stack pointer

Compare Result Registers

Compare result registers are two-bit registers that can hold a value from -1 to 1. Compare results are set as the target to several instructions including compare, bit test, and set operations.

-1 means the first operand was less than the second one. 0 means both operands are equal, +1 means the first operand was greater than the second.

Reg	Usage Convention
C0	Integer Compares / set
C1	Float compares / set
C2 to C7	

Link Registers

There are four link registers provided for subroutine calls. There is an additional link register to support exception processing routines.

Reg	Usage
L0	scrap (jumps)
L1	subroutine calls
L2	
L3	

XL	exception link

Control and Status Registers

Control Register Zero (CSR #000)

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

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

This register supports bit set / clear CSR instructions.

DCE

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

BPE

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

WBM bit

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

SPLE

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

HARTID (0x001)

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

TICK (0x002)

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

PTA (0x003)

This register contains the base address of the highest-level page directory for memory management and the paging table depth. The base address must be page aligned (16kB).

51 14	13 11	10 8	7 0
Paging Directory Base Address$_{15..14}$	~	TD	~

TD	
0	1 level lookup
1	2 level lookup
2	3 level lookup
3	4 level lookup
4 to 7	reserved

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 13 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 XL register.

BAD_INSTR (CSR 0x00B)

This register contains a copy of the exceptioned instruction.

SEMA (CSR 0x00C) Semaphores

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

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

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
51:47			reserved
46:44	**RM**	rm	rounding mode
43	**E5**	inexe	- inexact exception enable
42	**E4**	dbzxe	- divide by zero exception enable
41	**E3**	underxe	- underflow exception enable
40	**E2**	overxe	- overflow exception enable
39	**E1**	invopxe	- invalid operation exception enable
38	**NS**	ns	- non standard floating point indicator
Result Status			
32		fractie	- the last instruction (arithmetic or conversion) rounded intermediate result (or caused a disabled overflow exception)
31	**RA**	rawayz	rounded away from zero (fraction incremented)
30	**SC**	C	denormalized, negative zero, or quiet NaN
29	**SL**	neg <	the result is negative (and not zero)
28	**SG**	pos >	the result is positive (and not zero)
27	**SE**	zero =	the result is zero (negative or positive)
26	**SI**	inf ?	the result is infinite or quiet NaN
Exception Occurrence			
21 to 25			reserved
20	**X6**	swt	{reserved} - set this bit using software to trigger an invalid operation
19	**X5**	inerx	- inexact result exception occurred (sticky)
18	**X4**	dbzx	- divide by zero exception occurred
17	**X3**	underx	- underflow exception occurred
16	**X2**	overx	- overflow exception occurred
15	**X1**	giopx	- global invalid operation exception – set if any invalid operation exception has occurred
14	**GX**	gx	- global exception indicator – set if any enabled exception has happened
13	**SX**	sumx	- summary exception – set if any exception could occur if it was enabled - can only be cleared by software

Exception Type Resolution			
8 to 12			reserved
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

KEYS – (CSR 0x020 to 0x023)

These registers contain a collection of keys associated with the process for the memory system. Each key is twenty bits in size. Each register contains two keys for a total of eight keys. All four registers are searched in parallel for keys matching the one associated with the memory page.

51 46	45 26	25 20	19 0
$\sim_6$	key2	$\sim_6$	key1

DOI_STACK (0x040)

This register contains the stacks for the data operating level, code operating level and interrupt mask. All three stacks are packed into a single register for convenience and performance if the stacks are required to be saved or restored as part of context. When an exception or interrupt occurs, a) the interrupt stack is shifted to the left by three and the low order bits are set to all ones causing all interrupts to be masked b) the code operating level stack is shifted to the left and the low order bits are set to zero causing a switch to the machine operating level for code 3) the data operating level stack is shifted to the left and the low order bits are set to zero causing the machine operating level to be used for data access.

When an RTI instruction is executed these registers are shifted to the right, restoring the previous settings. a) The last interrupt stack entry is set to seven masking all interrupts on stack underflow. The low order three bits represent the current interrupt mask level. b) The last code operating level stack entry is set to zero causing a switch to machine mode on stack underflow. c) The last data operating level stack entry is set to zero causing the machine operating level to be used for data access.

Only the low order 45 bits of the register are implemented, remaining bits read as zero.

Bits 0 to 2 represent the current interrupt mask setting.

Bits 15 to 17 represent the current code operating level setting.

Bits 30 to 32 represent the current data operating level setting.

PL_STACK (0x042,0x043)

This pair of CSR's contains the privilege level stack. When an exception or interrupt occurs, this register is shifted to the left by 13, when an RTI instruction is executed this register is shifted to the right by 13. On RTI the last stack entry will be set to zero which will select privilege level zero on stack underflow. The low order thirteen bits of the register represent the current privilege level.

STATUS (0x044)

51 37	36 29	28	27 26	25 24	23 14	13 4	3	2	1	0
~	ASID	MPRV	XS	FS	~10	~10	~	$\sim_3$		

Bit		Meaning
0 to 2	$\sim_3$	reserved
4 to 13	~10	reserved
14 to 23	~10	reserved
24 to 25	FS	floating point state
26 to 27	XS	additional core extension state
28	MPRV	memory data level swap
29 to 36	ASID	address space identifier
37 to 51		reserved

BM_CTR (0xFC1)

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

IRQ_CTR (0xFC3)

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

BR_CTR (0xFC4)

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

TIME (0xFE0, 0xFE1)

The TIME pair of registers 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. CSR 0xFE0 bits 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 bits represent the fraction of one second. CSR 0xFE1 bits represent seconds passed. For example, if the tm_clk_i frequency is 100MHz the bits should count from 0 to 99,999,999 then cycle back to 0 again. When the bits cycle back to 0 again, the bits of the CSR 0xFE1 register are incremented.

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.

Note the time register is not set until the seconds register is set. The fraction of seconds value won't transfer to the time register until the seconds are set.

Each of these registers is only 40-bits in size.

TIME (0xFE1)

This register contains a reference of the number of seconds offset from the real wall clock time. It is a carry over from CSR 0xFE0 which contains the fraction of seconds value. The bits of the register represent the number of seconds passed since an arbitrary point in the past.

INSTRET (0xFE2)

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

INFO (0xFF0 to 0xFFF)

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

Memory Addressing

The cpu is word oriented with byte addressable memory. The smallest addressable unit of data is a 13-bit byte. Up to 2^{52} Bytes of data are supported and 2^{52} bytes of code. Code and data share the same address space.

Memory data may be accessed using load and store instructions. The load and store instructions support two basic addressing modes: register indirect with displacement and scaled indexed addressing.

Program code is byte aligned.

Operating Levels

The core has six distinct 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. When operating at level zero addresses are not subjected to translation and the virtual address and physical address are the same.

Operating Level		Privilege Level Available
0	machine	0 to 8191
1	hypervisor level	0 to 8191
2	supervisor level1	16 to 8191
3	supervisor level2	128 to 8191
4	supervisor level3	1024 to 8191
5	user	7168 to 8191

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 an 8192-level privilege level system. Available privilege levels correspond to the operating level as shown in the table above.

Exceptions

All processor exceptions vector using the same vector stored in memory location $FFFFFFFFE0000. The exceptions are differentiated by the value in the exception cause register.

Cause Code	Which Exception	
0	no exception	
25h	Unimplemented instruction encountered	
28h	divide by zero	
150h to 157h	Interrupt occurred	
160h	Non-maskable interrupt occurred	
161h	Sequence number reset	
170h	Processor was reset	
140h to 14Fh	the BRK instruction was executed	

Sequence Numbers

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

Sequence numbers are probably conceptually the simplest means to understand and are simple to implement. The drawback is that they may require more hardware than other means. The order of instructions in the queue is associated with a number. The core assigns a sequence number to each instruction as it enters the instruction queue. One purpose of the sequence number is to allow the core to determine which instructions should be invalidated because of a branch miss. All the instructions in the queue with a sequence number coming after the branch instruction's sequence number will be invalidated. In the case of Gambit, the sequence number assigned is simply the tick count times two, plus the queue slot number. One challenge of such a simple approach is that sequence numbers might overflow. The size of the sequence number is set small enough that there's no impact to core performance, and large enough that overflows are infrequent. The chosen size is 26 bits, allowing up to 64 million clock cycles between each sequence number overflow. If the sequence number were to overflow in an uncontrolled fashion the order of instructions in the queue would be upset. So, the sequence number overflow happens in a controlled fashion. Just before the number would overflow an interrupt is generated. During the interrupt routine the sequence number is reset to zero at a point of execution of instructions where sequence numbering won't impact execution of instructions. This is done by executing enough NOP instructions to fill the processor queue then issuing a sequence number reset command. It's important that no branches are executed during the sequence number reset. Note that the sequence number interrupt may be avoided by resetting the sequence number before it overflows. Many systems have a periodic interrupt which occurs much more frequently than a sequence number would need to be reset. Resetting the sequence number could be done during this routine. It does require a number of instructions to execute without branching; the number of such instructions depends on the processor queue size.

The sequence number mechanism allows the core to speculate across any number of branches that might be in the instruction queue. However, sequence numbers require more bits in the instruction queue than branch tags and even more hardware.

For debugging purposes, it can be handy to see the sequence number assigned to the instruction.

Memory Management

TLB – The Translation Lookaside Buffer

Overview

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

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

The TLB is manipulated with the TLB instruction.

Size / Organization

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

Updating the TLB

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

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

Issues with Existing Paged Units

Not enough information stored in the page table for proper OS management. This leads to the OS duplicating page table entries in order to store more information.

Page Table Entry

The following layout shows the page table entry structure as stored in memory.

51 39	38 26	25	24	23 20	19 16	15 12	11 8	7 4	2 1 0
Physical Page Number$_{38}$						DUA$_3$	~$_8$	T$_2$	P
Virtual Page Number$_{38}$									
Share Count$_{13}$	Privilege Level$_{13}$	U	S	CRWX	CRWX	CRWX	CRWX	CRWX	CRWX
Reference Counter$_{26}$		~$_6$			Protection Key$_{20}$				

Word	Bit				
0	0		P	Page present	1 = page present in memory
	1 to 2		T	Entry type	1 = page directory entry, 2 = page table entry
	3 to 10			reserved	
	11		A	1 = accessed	
	12		U	undefined usage	available for use by OS
	13		D	1 = dirty	set if the page is written to
	14 to 51		PPN	physical page number	
1	0 to 13			reserved	
	14 to 51		VPN	virtual page number	
2	0	User	X_u	1 = executable	
	1		W_u	1 = page writeable	
	2		R_u	1 = readable	
	3		C_u	1 = cachable	Ignored for executable pages which are always cached
	4 to 7	Other	…	supervisor	
	8 to 11		…	supervisor	
	12 to 15		…	supervisor	
	16 to 19		…	hypervisor	
	20	Machine	X_m		
	21		W_m		
	22		R_m		
	23		C_m		
	24		S	1 = shortcut translation	Translation shortcut bit eg (16MiB pages)
	25		U	undefined usage	available for use by OS
	26 to 38		PL	Privilege Level	
	39 to 51		SC	Share Count	number of times page is shared
3	0 to 19		PK	Protection Key	process must have a matching key in its collection for access
	20 to 25			reserved	
	26 to 51		RC	Reference Counter	

Page Directory Entry

51 … 14	13 … 11	10 … 3	2 1	0
Page Number$_{38}$	DUA$_3$	~$_8$	T$_2$	P
Virtual Page Number$_{38}$				

Instruction Set Summary

The instruction set includes basic arithmetic, logic, and shift instructions including add, sub, and, or, eor, asl, rol, lsr, and ror. There are several additional instructions which are alternate mnemonics for other instructions. These include bit, and cmp.

The cycle counts are assuming no wait states are required for either instructions or data and both instructions and data can be found in the cache.

ALU Operations

ADD – Addition

Description:

Add two operand values and place the result in the target register. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ADDIS – Add Immediate Shifted

Description:

Add an immediate value to bits 22 to 51 of register Ra_5 and place the result in target register Rt_5. Use this instruction when performing an addition with a full 52-bit constant or when building a value in a register.

Formats Supported: RIS

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

AND – Bitwise ‘And’

Description:

Bitwise ‘And’ two operand values and place the result in the target register. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ANDIS – And Immediate Shifted

Description:

Bitwise ‘and’ an immediate value to bits 22 to 51 of register Ra_5 and place the result in target register Rt_5. Use this instruction when performing an addition with a full 52-bit constant or when building a value in a register. The constant used is one extended on the right hand side.

Formats Supported: RIS

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ASL – Arithmetic Shift Left

Description:

Left shift one operand value by a second operand value and place the result in the target register. Zeros are shifted into the least significant bits. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ASR – Arithmetic Shift Right

Description:

Right shift one operand value by a second operand value while preserving the sign bit and place the result in the target register. The sign bit is preserved as the shift takes place. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

BIT – Test Bits

Description:

Bitwise 'And' two operand values and place a result in a compare result register. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value. If the result of the 'and' operation is zero, a zero is stored in the compare result register. If the result of the 'and' operation is non-zero then either +1 or -1 is stored in the result register depending on the sign of the result.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

CMP – Comparison

Description:

Compare two operand values and store the relationship in the target compare result register. The operand values are treated as signed integers. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

If the first operand is less than the second, a minus one is stored in the target register. If the first operand equals the second a zero is stored to the target register, otherwise a positive one is stored.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

CMPU – Unsigned Comparison

Description:

Compare two operand values and store the relationship in the target compare result register. The operand values are treated as unsigned integers. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

If the first operand is less than the second, a minus one is stored in the target register. If the first operand equals the second a zero is stored to the target register, otherwise a positive one is stored.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

CSR – Control and Status Register Operation

Description:

The CSR instruction may be used to read, write or read and write a control or status register. The CSR to access is identified by a twelve-bit constant in the instruction. The current value of the CSR will be read into the register specified by the Rt field of the instruction. If Rt is zero the read data will be discarded. The CSR will be updated with the contents of the register specified by the Ra field of the instruction. If Ra is specified as register zero, then no update takes place. The CSR read / update operation is an atomic operation.

Formats Supported: CSR

Op_3		Operation
0	CSRRD	Only read the CSR, no update takes place, Rs1 should be R0.
1	CSRRW	Both read and write the CSR
2	CSRRS	Read CSR then set CSR bits
3	CSRRC	Read CSR then clear CSR bits
4		reserved
5	CSRRWI	Read and Write CSR with immediate
6	CSRRSI	Read and set using immediate
7	CSRRCI	Read and clear using immediate

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

The OL_2 field is reserved to specify the operating level. Note that registers cannot be accessed by a lower operating level.

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

Examples:

CSRRD r1,r0,#CAUSECD ; read the cause code register into r1

DIV – Division

Description:

Divide two operand values and place the result in the target register. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or a small immediate value. Both operands are treated as signed values.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 55

Exceptions: none

EOR – Bitwise Exclusive 'Or'

Description:

Bitwise exclusive 'Or' two operand values and place the result in the target register, updating status flags. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

LSR – Logical Shift Right

Description:

Right shift one operand value by a second operand value and place the result in the target register. Zeros are shifted into the most significant bits. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

MOV – Move Register to Register

Description:

This instruction moves from one general-purpose register to another general-purpose register. It is an alternate mnemonic for the OR instruction where Rb is assumed to be R0.

Formats Supported: RR

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

MVF – Move From Register

Description:

This instruction moves from a specific use register to a general-purpose register.

Formats Supported: RR

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

Ra_5	Register	
0 to 3	L0 to L3	link registers
4	XL	exception link register
5,6		reserved
7	CA	all compare registers combined
8 to 15	C0 to C7	compare result registers
16 to 23		reserved (vector mask)
24	SSP	supervisor stack pointer
25	HSP	hypervisor stack pointer
26	MSP	machine stack pointer
27 to 31		reserved

MVT – Move To Register

Description:

This instruction moves a general-purpose register into one of the specific use registers.

Formats Supported: RR

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

Rt_5	Register	
0 to 3	L0 to L3	link registers
4	XL	exception link register
5,6		reserved
7	CA	all compare registers combined
8 to 15	C0 to C7	compare result registers
16 to 23		reserved (vector mask)
24	SSP	supervisor stack pointer
25	HSP	hypervisor stack pointer
26	MSP	machine stack pointer
27 to 31		reserved

MUL – Multiplication

Description:

Multiply two operand values and place the result in the target register. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value. Both operands are treated as signed values.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

OR – Bitwise ‘Or’

Description:

Bitwise ‘Or’ two operand values and place the result in the target register, updating status flags. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ORIS – Or Immediate Shifted

Description:

Bitwise ‘or’ an immediate value to bits 22 to 51 of register Ra_5 and place the result in target register Rt_5. Use this instruction when performing an addition with a full 52-bit constant or when building a value in a register. The constant used is zero extended on the right-hand side.

Formats Supported: RIS

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

PERM – Permute Bytes

Description:

This instruction allows any combination of bytes in a source register to be copied to a target register. There are four two-bit fields which specify which source bytes to copy to the target. The source fields may be either a constant specified in the instruction, or the low order eight bits of register Rb. Field S0 indicates the source byte for target byte position 0. S1 indicates the source byte for target byte position 1. S2 and S3 work similarly for the last two target bytes. There are many interesting possibilities with this instruction. A single source byte could be copied to all target byte positions for instance. Or the order of bytes in a word could be reversed.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ROL – Rotate Left

Description:

Rotate left one operand value by a second operand value and place the result in the target register, updating status flags. The most significant bits are placed in the least significant bits. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

ROR – Rotate Right

Description:

Rotate right one operand value by a second operand value and place the result in the target register, updating status flags. The least significant bits are placed in the most significant bits. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value.

Formats Supported: RR, RI8

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

SUB – Subtraction

Description:

Subtract two operand values and place the result in the target register, updating status flags. The first operand must be in a register specified by the Ra_5 field of the instruction. The second operand may be either a register specified by the Rb_5 field of the instruction, or an immediate value. The subtraction instruction has an alternate mnemonic CMP which updates the flags differently when the target register is R0.

Formats Supported: RR, RI8, RI22, RI35

Execution Units: ALU

Clock Cycles: 0.5

Exceptions: none

Memory Operations

LD – Load Data (52 bits)

Description:

Data is loaded from the memory address which is either the sum of Ra and an immediate value or the sum of Ra and Rb. Both register indirect with displacement and indexed addressing are supported.

Formats Supported: RR, RI8, RI22, RI35

Operation:

Rt = Memory_{52}[d+Ra]
or
Rt = Memory_{52}[Ra+Rb]

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

LDB – Load Data Byte (13 bits)

Description:

Data is loaded from the memory address which is the sum of Ra and an immediate value. Only register indirect with displacement addressing is supported.

Formats Supported: RI36

Flags Affected: n z

Operation:

$Rt = Memory_{13}[d+Ra]$

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

LDR – Load and Reserve (52 bits)

Description:

Data is loaded from the memory address which is either the sum of Ra and an immediate value or the sum of Ra and Rb. Additionally a reservation is placed on the load address. Both register indirect with displacement and indexed addressing are supported.

Formats Supported: RR, RI8

Operation:

Rt = Memory_{52}[d+Ra]
or
Rt = Memory_{52}[Ra+Rb]

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

ST – Store Data (52 bits)

Description:

Data is stored to the memory address which is either the sum of Ra and an immediate value or the sum of Ra and Rb. Both register indirect with displacement and indexed addressing are supported.

Formats Supported: RR, RI8, RI22, RI35

Flags Affected: none

Operation:

$Memory_{52}[d+Ra] = Rs$
or
$Memory_{52}[Ra+Rb] = Rs$

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

STB – Store Data Byte (13 bits)

Description:

Data is stored to the memory address which is the sum of Ra and an immediate value. Only register indirect with displacement addressing is supported.

Formats Supported: RI35

Flags Affected: none

Operation:

$Memory_{13}$[d+Ra] = Rs
or
$Memory_{13}$[Ra+Rb] = Rs

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

STC – Store Data and Clear Reservation

Description:

Data is conditionally stored to the memory address if a reservation exists on the address which is either the sum of Ra and an immediate value or the sum of Ra and Rb. Both register indirect with displacement and indexed addressing are supported. The status of the store (completed or not) is stored in processor semaphore bit #0.

Formats Supported: RR, RI8

Flags Affected: none

Operation:

$Memory_{52}$[d+Ra] = Rs
or
$Memory_{52}$[Ra+Rb] = Rs

Execution Units: Mem

Clock Cycles: 4 if data is in the cache.

Exceptions: none

Flow Control (Branch Unit) Operations

There are only signed versions of branches since the branch is primarily due to the output of the compare instruction which outputs values of -1,0 or +1.

BEQ – Branch if Equal

Description:

This instruction branches to the target address if the compare results register is zero. The target address is the address of the next instruction plus a 12-bit displacement specified in the instruction. This branch may be either statically or dynamically predicted.

Formats Supported: Bcc

P_2	Meaning
0	no prediction (dynamic prediction)
1	reserved
2	predict not taken (static prediction)
3	predict taken (static prediction)

Operation:

If (cr==0)
    PC = NextPC + Displacement_{12}

Execution Units: Branch

Clock Cycles: 1

Exceptions: none

BGE – Branch if Greater Than or Equal

Description:

This instruction branches to the target address if the compare results register is greater than or equal to zero (0 or +1). The target address is the address of the next instruction plus a 12-bit displacement specified in the instruction.

Formats Supported: Bcc

Operation:

If (cr >= 0)
    PC = NextPC + Displacement_{12}

Execution Units: Branch

Clock Cycles: 1

Exceptions: none

BGT – Branch if Greater Than

Description:

This instruction branches to the target address if the compare results register is greater than zero (+1). The target address is the address of the next instruction plus a 12-bit displacement specified in the instruction.

Formats Supported: Bcc

Operation:

If (cr > 0)
    PC = NextPC + $Displacement_{12}$

Execution Units: Branch

Clock Cycles: 1

Exceptions: none

BLE – Branch if Less Than or Equal

Description:

This instruction branches to the target address if the compare results register is less than or equal to zero. The target address is the address of the next instruction plus a 12-bit displacement specified in the instruction.

Formats Supported: Bcc

Operation:

If (cr <= 0)
    PC = NextPC + Displacement_{12}

Execution Units: Branch

Clock Cycles: 1

Exceptions: none

BLT – Branch if Less Than

Description:

This instruction branches to the target address if the compare results register is less than zero (-1 or -2). The target address is the address of the next instruction plus a 12-bit displacement specified in the instruction.

Formats Supported: Bcc

Operation:

If (cr < 0)
    $PC = NextPC + Displacement_{12}$

Execution Units: Branch

Clock Cycles: 1

Exceptions: none

BNE – Branch if Not Equal

Description:

This instruction branches to the target address if the compare results register is not zero. The target address is the address of the next instruction plus a 12-bit displacement specified in the instruction.

Formats Supported: Bcc

Operation:

If (cr <> 0)
    PC = NextPC + $Displacement_{12}$

Execution Units: Branch

Clock Cycles: 1

Exceptions: none

BRA – Branch Always

Description:

This instruction branches unconditionally to the target address. The target address is the address of the next instruction plus a 12-bit displacement specified in the instruction. The prediction bits should be set to indicate a static prediction of taken so that the branch does not consume history table resources.

Formats Supported: Bcc

Operation:

PC = NextPC + $Displacement_{12}$

Execution Units: Branch

Clock Cycles: 1

Exceptions: none

BRK – Break

Description:

This instruction initiates the processor break routine. The cause code register is set to four. The program counter is reset to $FFFFFFFFE0000 and instructions begin executing.

Formats Supported: BRK

Operation:

CAUSE = 40h | $Const_4$
OLS = OLS << 2
DLS = DLS << 2
XL = PC + 1
PC = $FFFFFFFFE0000

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

JMP – Jump

Description:

This instruction is an alternate mnemonic for the JAL instruction where the link register is assumed to be L0. JMP transfers execution of instructions to the address specified by the instruction. The target address may be either a 43-bit absolute address or an address contained in a register. For absolute address mode only the low order 43 bits of the program counter are affected. The upper nine bits of the program counter remain the same.

Formats Supported: ABS43, R

Flags Affected: none

Operation:

PC = $Address_{43}$
or
PC = Ra

Execution Units: Branch

Clock Cycles: 1

Exceptions: none

JAL – Jump and Link

Description:

Store the return address in the specified link register then jump to the address specified as an absolute constant or the contents of register Ra_4. The target address may be either a 43-bit absolute address or an address contained in a register. For absolute address mode only the low order 43 bits of the program counter are affected. The upper nine bits of the program counter remain the same. Note that only registers R0 to R15 may be specified as containing a jump target. This is due to limitations in the instruction encoding.

Formats Supported: ABS43, R

Flags Affected: none

Operation:

Lk = NextPC
PC = $Address_{43}$
or
PC = Ra

Execution Units: Mem

Clock Cycles: 0.5

Exceptions: none

Notes:

The next PC is either the current PC plus four when absolute addressing is used, or the current PC plus one if register indirect addressing is used.

MRK – Marker

Description:

This instruction is treated by the processor as a NOP operation. It is used to mark positions in a software emulator or simulator.

Formats Supported: MRK

Flags Affected: none

Operation: none

Execution Units: Branch

Clock Cycles: 0.5

Exceptions: none

NMI – Non-Maskable Interrupt

Description:

This instruction initiates the processor exception handling routine. The cause code register is set to 60h. The program counter is reset to $FFFFFFFFFE0000 and instructions begin executing.

Formats Supported: BRK

Operation:

CAUSE = 60h
OLS = OLS << 2
DLS = DLS << 2
IMS = (IMS << 3) | 7
XL = PC + 1
PC = $FFFFFFFFFE0000

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

NOP – No Operation

Description:

This instruction is treated by the processor as a NOP operation.

Formats Supported: MRK

Flags Affected: none

Operation: none

Execution Units: Branch

Clock Cycles: 0.5

Exceptions: none

PFI – Poll for Interrupt

Description:

The PFI instruction tests for the presence of an interrupt and performs the interrupt routine if an interrupt is present. If no interrupt is present a NOP operation is performed, and the program continues with the next instruction. PFI does not check for a non-maskable (NMI) interrupt or a reset (RST). Processing for the interrupt routine begins at the universal exception handler address of $FFFFFFFFE0000.

PFI may scan three interrupt signalling lines, which lines to scan are specified by a bit-mask in the instruction.

Formats Supported: PFI

Flags Affected: none

Operation:

```
If (IRQ & scan mask)
        Cause Code  = 50h | IRQ Level
        OLS = OLS << 3
        DLS = DLS << 3
        IMS = (IMS << 3) | 7
        PLS = PLS << 13
        XLR = PC + 1;
        PC = $FFFFFFFFE0000
Else
        NOP
```

Execution Units: Fetch stage

Clock Cycles:

Exceptions: none

Notes:

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. 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 8191. The instruction will not allow setting the privilege level numerically less than the operating level.

The exception handler address is $FFFFFFFFE0000

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

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

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

Instruction Format: REX

Tgt_3		
0	not used	0 to 8191
1	redirect to hypervisor level	0 to 8191
2	redirect to supervisor level1	16 to 8191
3	redirect to supervisor level2	128 to 8191
4	redirect to supervisor level3	1024 to 8191
5		7168 to 8191

Clock Cycles: 4

Execution Units: Branch

Example:

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

Notes:

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

RST – Reset Processor

Description:

This instruction initiates the processor reset routine. The cause code register is set to 70h. The program counter is reset to $FFFFFFFFE0000 and instructions begin executing.

Formats Supported: RST

Operation:

Execution Units: Branch

Clock Cycles:

Exceptions: none

Notes:

RTI – Return from Interrupt Subroutine

Description:

Restore the previous interrupt and operating level and transfer program execution back to the address in the exception link register. One of the first sixteen semaphore registers may also be cleared. Semaphore register zero is always cleared by this instruction.

Formats Supported: RTI

Flags Affected: none

Operation:

OLS = OLS >> 3
DLS = DLS >> 3
IMS = IMS >> 3
PLS = PLS >> 13
Semaphore[0] = 0
Semaphore[$Sema_4$] = 0
PC = XL

Execution Units: Mem

Clock Cycles:

Exceptions: none

Notes:

RTS – Return from Subroutine

Description:

Transfer program execution to an address stored in a link register. The link register will have been previously set by a subroutine call operation.

Formats Supported: RTS

Flags Affected: none

Operation:

PC = Lk

Execution Units: Mem

Clock Cycles: 0.5

Exceptions: none

Notes:

SNR – Sequence Number Reset

Description:

This instruction initiates the processor exception handling routine. The cause code register is set to 61h for the sequence number reset interrupt which is non-maskable. The program counter is reset to $FFFFFFFFE0000 and instructions begin executing. A sequence number reset exception is generated internally by the core based on the tick count which is about to roll-over in the sequence number bits. (A portion of the tick count is used for sequence numbering).

Formats Supported: BRK

Operation:

CAUSE = 61h
OLS = OLS << 3
DLS = DLS << 3
IMS = (IMS << 3) | 7
PLS = PLS << 13
XL = PC + 1
PC = $FFFFFFFFE0000

Execution Units: Branch

Clock Cycles:

Exceptions: none

Sample Code:

CSRRD r1,r0,#CAUSE	; read the cause
CMP c0,r1,#61h	; is it a sequence reset?
BNE c0,.notSNR	; no go do other exception code
NOP	; use 63 nops to flush the processor queue
… (61 more NOP's)	;
NOP	;
CSRRSI r0,#1,#SNRREG	; pulse the sequence number reset
RTI	; and it's finished so return.
.notSNR:	

Notes:

STP – Stop

Description:

This instruction is used to stop the processor by stopping the clock internally. The stopped state may be exited by a reset or nmi.

Formats Supported: STP

Flags Affected: none

Operation: none

Execution Units: Branch

Clock Cycles: 0.5

Exceptions: none

WAI – Wait for Interrupt

Description:

The WAI instruction waits for an interrupt to occur by holding the program counter steady. This instruction is similar to the PFI instruction except that it stops and waits for an interrupt whereas PFI doesn't wait. WAI does not check for a non-maskable (NMI) interrupt or a reset (RST).

Formats Supported: WAI

Flags Affected: none

Operation:

```
If (IRQ)
        Cause Code  = 50h | IRQ Level
        OLS = OLS << 3
        DLS = DLS << 3
        IMS = (IMS << 3) | 7
        PLS = PLS << 13
        XLR = PC + 1;
        PC = $FFFFFFFFE0000
Else
        PC = PC
```

Execution Units: Fetch stage

Clock Cycles:

Exceptions: none

Notes:

Floating Point Instructions

Overview

The floating-point unit provides basic floating-point operations including addition, subtraction, multiplication, division, square root, and float to integer and integer to float conversions. The core contains two identical floating-point units. Only 52-bit precision floating-point operations are supported. The core features results caching, if the same operation is performed on the same values as is present in the cache then the result is returned in a single clock cycle.

The rounding mode is normally specified directly in the instruction. However, if the instruction indicates to use dynamic rounding mode then the rounding mode in the floating-point control and status register is used.

Representation

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

52-bit Precision Format:

51	50	49 40	39 0
S_M	S_E	Exponent	Mantissa

S_M – sign of mantissa
S_E – sign of exponent

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

S_eEEEEEEEEEE	
11111111111	Maximum exponent
….	
01111111111	exponent of zero
….	
00000000000	Minimum exponent

The exponent ranges from -1023 to +1024

FABS – Floating Absolute Value

Description:

Take the absolute value of a floating-point number in register Fa and places the result into target register Ft. The sign bit (bit 51) of the register is set to zero. No rounding of the number occurs.

Instruction Format: FLT1

Clock Cycles: 0.5

Execution Units: Floating Point

FADD – Floating point addition

Description:

Add two floating point numbers in registers Fa and Fb and place the result into target register Ft.

Instruction Format: FLT2

Clock Cycles: 6

Execution Units: Floating Point

FCLASS – Classify Value

Description:

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

Bit	Meaning
0	1 = negative infinity
1	1 = negative number
2	1 = negative subnormal number
3	1 = negative zero
4	1 = positive zero
5	1 = positive subnormal number
6	1 = positive number
7	1 = positive infinity
8	1 = signalling nan
9	1 = quiet nan

FCMP - Float Compare

Description:

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

Instruction Format: FLT2

Clock Cycles: 0.5

Execution Units: Floating Point

Operation:

```
if Fa < Fb
        Cr = -1
else if Fa = Fb
        Cr = 0
else
        Cr = 1
```

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: FLT1

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: FXX

Clock Cycles: 2

Execution Units: Floating Point

FDIV – Floating point divide

Description:

Divide two floating point numbers in registers Fa and Fb and place the result into target register Ft.

Instruction Format: FLT2

Clock Cycles: 22 (est).

Execution Units: Floating Point

FEX – Floating Enable Exceptions

Description:

This instruction enables floating point exceptions. The Exceptions enabled are identified as the bits set in the union of register 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: FXX

Clock Cycles: 2

Execution Units: Floating Point

FINITE – Number is Finite

Description:

Test the value in Fa to see if it's a finite number and return true (1) or false (0) in compare result register Ct.

Instruction Format: FLT1

Clock Cycles: 0.5

Execution Units: Floating Point

Example:

```
finite    $cr1,$f7
```

FMUL – Floating point multiplication

Description:

Multiply two floating point numbers in registers Fa and Fb and place the result into target register Ft.

Instruction Format: FLT2

Clock Cycles: 7

Execution Units: Floating Point

FNABS – Floating Negative Absolute Value

Description:

Take the negative absolute value of the floating-point number in register Fa and place the result into target register Ft. The sign bit (bit 51) of the register is set to one. No rounding of the number occurs.

Instruction Format: FLT1

Clock Cycles: 0.5

Execution Units: Floating Point

FNEG – Floating Negative Value

Description:

Negate the value of the floating-point number in register Fa and place the result into target register Ft. The sign bit (bit 51) of the register is inverted. No rounding of the number occurs.

Instruction Format: FLT1

Clock Cycles: 0.5

Execution Units: Floating Point

FRES – Reciprocal Estimate

Description:

This function uses a 1024 entry 16-bit precision lookup table to create a piece-wise approximation of the reciprocal and linear interpolation to approximate the reciprocal of the value in Fa. The value is returned in Ft as a 52-bit floating-point value. The value returned is accurate to about eight bits.

Instruction Format: FLT1

Clock Cycles: 5

Execution Units: Floating Point

FRSQRTE – Float Reciprocal Square Root Estimate

Description:

Estimate the reciprocal of the square root of the number in register Fa and place the result into target register Ft.

Instruction Format: FLT1

Clock Cycles: 5

Execution Units: Floating Point

Notes:

The estimate is only accurate to about 3%. The estimate is performed in single precision (32-bit) floating point, then converted to a 52-bit format. That means that input values must in the range of a 32-bit floating point number. Values outside of this range will return infinity or zero as a result.

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

FSEQ - Float Set if Equal

Description:

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

Instruction Format: FLT2

Clock Cycles: 0.5

Execution Units: Floating Point

Operation:

```
if Fa == Fb
        Cr= 1
else
        Cr= 0
```

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. This operation may also store the sign in a compare result register as -1, 0 or +1.

Instruction Format: FLT1

Clock Cycles: 0.5

Execution Units: Floating Point

FSLE - Float Set if Less Than or Equal

Description:

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

Instruction Format: FLT2

Clock Cycles: 0.5

Execution Units: Floating Point

Operation:

```
if Fa <= Fb
        Cr = 1
else
        Cr = 0
```

FSLT - Float Set if Less Than

Description:

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

Instruction Format: FLT2

Clock Cycles: 0.5

Execution Units: Floating Point

Operation:

```
if Fa < Fb
        Cr = 1
else
        Cr = 0
```

FSNE - Float Set if Not Equal

Description:

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

Instruction Format: FLT2

Clock Cycles: 0.5

Execution Units: Floating Point

Operation:

```
if Fa <> Fb
        Cr= 1
else
        Cr= 0
```

FSQRT – Floating point square root

Description:

Take the square root of the floating-point number in register Fa and place the result into target register Ft. The sign bit (bit 51) of the register is set to zero. This instruction can generate NaNs.

Instruction Format: FLT1

Clock Cycles: 50 (est).

Execution Units: Floating Point

FSUB – Floating point subtraction

Description:

Subtract two floating-point numbers in registers Fa and Fb and place the result into target register Ft.

Instruction Format: FLT2

Clock Cycles: 6

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: FSYNC

Clock Cycles: varies depending on queue contents

FTOI – Floating Convert to Integer

Description:

Convert the floating-point value in Fa into an integer and place the result into a target register. The target register may be either another floating-point register or an integer register. If the result overflows the value placed in the target is a maximum integer value. Note that the result in the target register is no longer of a floating-point representation.

Instruction Format: FLT1

Clock Cycles: 3

Execution Units: Floating Point

FTRUNC – Truncate Value

Description:

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

Instruction Format: FLT1

Clock Cycles: 0.5

Execution Units: Floating Point

ISNAN – Is Not a Number

Description:

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

Instruction Format: FLT1

Clock Cycles: 0.5

Execution Units: Floating Point

Example:

```
isnan    $cr1,$f7
```

ITOF – Convert Integer to Float

Description:

Convert the integer value in Ra into a floating-point value and place the result into target register Ft. Ra is from either the floating-point register set or the integer register set, Ft is in the floating-point register set. Some precision of the integer converted may be lost if the integer is larger than 52 bits. 52-bit precision floating point values only have a precision of 41 bits.

Instruction Format: FLT1

Clock Cycles: 3

Execution Units: Floating Point

Oddball

An assortment of instructions that are not executed on one of the regular functional units.

CACHE – Cache Command

CACHE Cmd, [Rn]

Description:

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

Instruction Formats: CACHE

Commands:

IC_2	Mne.	Operation
0	NOP	no operation
1	invline	invalidate line associated with given address
2	invall	invalidate the entire cache (address is ignored)

DC_3	Mne.	Operation
0	NOP	no operation
1	enable	enable cache (instruction cache is always enabled)
2	disable	not valid for the instruction cache
3	invline	invalidate line associated with given address
4	invall	invalidate the entire cache (address is ignored)

Notes:

Instruction Formats

Arithmetic / Logical

ADD							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	Rt_5	04h	ADD Rt,Ra,Rb	2
		1	Imm_8		Ra_5	Rt_5	04h	ADD Rt,Ra,#imm_8	2
	Imm_{22}				Ra_5	Rt_5	14h	ADD Rt,Ra,#imm_{22}	3
Imm_{35}					Ra_5	Rt_5	24h	ADD Rt,Ra,#imm_{35}	4

SUB							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	Rt_5	05h	Rt,Ra,Rb	2
		1	Imm_8		Ra_5	Rt_5	05h	Rt,Ra,#imm_8	2
	Imm_{22}				Ra_5	Rt_5	15h	Rt,Ra,#imm_{22}	3
Imm_{35}					Ra_5	Rt_5	25h	Rt,Ra,#imm_{35}	4

CMP								Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11	7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	1	Ct_3	06h	Rt,Ra,Rb	2
		1	Imm_8		Ra_5	1	Ct_3	06h	Rt,Ra,#imm_8	2
	Imm_{22}				Ra_5	1	Ct_3	16h	Rt,Ra,#imm_{22}	3
Imm_{35}					Ra_5	1	Ct_3	26h	Rt,Ra,#imm_{35}	4

CMPU								Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11	7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	1	Ct_3	07h	Rt,Ra,Rb	2
		1	Imm_8		Ra_5	1	Ct_3	07h	Rt,Ra,#imm_8	2
	Imm_{22}				Ra_5	1	Ct_3	17h	Rt,Ra,#imm_{22}	3
Imm_{35}					Ra_5	1	Ct_3	27h	Rt,Ra,#imm_{35}	4

MUL							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	Rt_5	0Eh	Rt,Ra,Rb	2
		1	Imm_8		Ra_5	Rt_5	0Eh	Rt,Ra,#imm_8	2
	Imm_{22}				Ra_5	Rt_5	1Eh	Rt,Ra,#imm_{22}	3
Imm_{35}					Ra_5	Rt_5	2Eh	Rt,Ra,#imm_{35}	4

DIV							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	0	Rb_5	Ra_5	Rt_5	03h	Rt,Ra,Rb	2
		1	Imm_8		Ra_5	Rt_5	03h	Rt,Ra,#imm_8	2

MOD							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	4	Rb_5	Ra_5	Rt_5	03h	Rt,Ra,Rb	2

AND							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	~$_3$	Rb$_5$	Ra$_5$	Rt$_5$	08h	Rt,Ra,Rb	2
		1	Imm$_8$		Ra$_5$	Rt$_5$	08h	Rt,Ra,#imm$_6$	2
	Imm$_{22}$				Ra$_5$	Rt$_5$	18h	Rt,Ra,#imm$_{14}$	3
Imm$_{35}$					Ra$_5$	Rt$_5$	28h	Rt,Ra,#imm$_{30}$	4

BIT								Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11	7	6 0		
		0	~$_3$	Rb$_5$	Ra$_5$	~	Ct$_3$	55h	Rt,Ra,Rb	2
		1	Imm$_8$		Ra$_5$	~	Ct$_3$	55h	Rt,Ra,#imm$_6$	2
	Imm$_{22}$				Ra$_5$	~	Ct$_3$	65h	Rt,Ra,#imm$_{14}$	3
Imm$_{35}$					Ra$_5$	~	Ct$_3$	75h	Rt,Ra,#imm$_{30}$	4

OR							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Op$_3$	Rb$_5$	Ra$_5$	Rt$_5$	09h	Rt,Ra,Rb	2
		1	Imm$_8$		Ra$_5$	Rt$_5$	09h	Rt,Ra,#imm$_6$	2
	Imm$_{22}$				Ra$_5$	Rt$_5$	19h	Rt,Ra,#imm$_{14}$	3
Imm$_{35}$					Ra$_5$	Rt$_5$	29h	Rt,Ra,#imm$_{30}$	4

Op$_3$	Function
0	normal ‘Or’ operation
1	zero extend byte result
2	zero extend wyde result
3	reserved
4	reserved
5	sign extend byte (13 bits)
6	sign extend wyde (26 bits)
7	reserved

EOR							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	Rt_5	0Ah	Rt,Ra,Rb	2
		1	Imm_8		Ra_5	Rt_5	0Ah	Rt,Ra,#imm_6	2
	Imm_{22}				Ra_5	Rt_5	1Ah	Rt,Ra,#imm_{14}	3
Imm_{35}					Ra_5	Rt_5	2Ah	Rt,Ra,#imm_{30}	4

Shifted Immediate

ADDIS						Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0		
4_5	Imm_{30}			Ra_5	Rt_5	2Ch	Rt,Ra,#imm_{35}	4

ANDIS						Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0		
8_5	Imm_{30}			Ra_5	Rt_5	2Ch	Rt,Ra,#imm_{35}	4

ORIS						Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0		
9_5	Imm_{30}			Ra_5	Rt_5	2Ch	Rt,Ra,#imm_{35}	4

EORIS						Opcode		Bytes
51 39	38 26	25 22	21 17	16 12	11 7	6 0		
10_5	Imm_{30}			Ra_5	Rt_5	2Ch	Rt,Ra,#imm_{35}	4

Shift Operations

ASL							Opcode		Bytes
51 39	38 26	25 22		21 17	16 12	11 7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	Rt_5	0Ch	Rt,Ra,Rb	2
		1	$\sim_2$	Imm_6	Ra_5	Rt_5	0Ch	Rt,Ra,#imm$_6$	2

ROL							Opcode		Bytes
51 39	38 26	25 22		21 17	16 12	11 7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	Rt_5	1Ch	Rt,Ra,Rb	2
		1	$\sim_2$	Imm_6	Ra_5	Rt_5	1Ch	Rt,Ra,#imm$_6$	2

LSR							Opcode		Bytes
51 39	38 26	25 22		21 17	16 12	11 7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	Rt_5	0Dh	Rt,Ra,Rb	2
		1	$\sim_2$	Imm_6	Ra_5	Rt_5	0Dh	Rt,Ra,#imm$_6$	2

ROR							Opcode		Bytes
51 39	38 26	25 22		21 17	16 12	11 7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	Rt_5	1Dh	Rt,Ra,Rb	2
		1	$\sim_2$	Imm_6	Ra_5	Rt_5	1Dh	Rt,Ra,#imm$_6$	2

ASR							Opcode		Bytes
51 39	38 26	25 22		21 17	16 12	11 7	6 0		
		0	$\sim_3$	Rb_5	Ra_5	Rt_5	2Dh	Rt,Ra,Rb	2
		1	$\sim_2$	Imm_6	Ra_5	Rt_5	2Dh	Rt,Ra,#imm$_6$	2

Load and Store Instructions

S_3	Scale by
0	1
1	2
2	4
3	8
4 to 7	reserved

LD							Opcode		Bytes
51 39	38 26	25		17	16 12	11 7	6 0		
		0	S_3	Rb_5	Ra_5	Rt_5	50h	Rt.[Ra+Rb]	2
		1	$Disp_8$		Ra_5	Rt_5	50h	Rt,d8[Ra]	2
	$Disp_{22}$				Ra_5	Rt_5	60h	Rt,d22[Ra]	3
$Addr_{35}$					Ra_5	Rt_5	70h	Rt,d35[Ra]	4

LDB							Opcode		Bytes
51 39	38 26	25		17	16 12	11 7	6 0		
		0	S_3	Rb_5	Ra_5	Rt_5	51h	Rt.[Ra+Rb]	2
		1	$Disp_8$		Ra_5	Rt_5	51h	Rt,d8[Ra]	2
	$Disp_{22}$				Ra_5	Rt_5	61h	Rt,d22[Ra]	3
$Addr_{35}$					Ra_5	Rt_5	71h	Rt,d35[Ra]	4

ST							Opcode		Bytes
51 39	38 26	25		17	16 12	11 7	6 0		
		0	S_3	Rb_5	Ra_5	Rs_5	58h	Rs.[Ra+Rb]	2
		1	$Disp_8$		Ra_5	Rs_5	58h	Rs,d8[Ra]	2
	$Disp_{22}$				Ra_5	Rs_5	68h	Rs,d22[Ra]	3
$Addr_{35}$					Ra_5	Rs_5	78h	Rs,d35[Ra]	4

STB								Opcode		Bytes
51 39	38 26	25		17	16 12	11 7		6 0		
		0	S_3	Rb_5	Ra_5	Rs_5		59h	Rs.[Ra+Rb]	2
		1	$Disp_8$		Ra_5	Rs_5		59h	Rs,d8[Ra]	2
	$Disp_{22}$				Ra_5	Rs_5		69h	Rs,d22[Ra]	3
$Addr_{35}$					Ra_5	Rs_5		79h	Rs,d35[Ra]	4

Flow Control

JAL	Flags:					Bytes
	Address$_{43}$		L_2	42h	JAL abs43	4
		Ra_4	L_2	48h	JAL [Ra]	1

{RTGRP}

RTS		$\sim_2$	L_2	0	44h	RTS	1
RTI		Sema$_4$		1	44h	RTI	1
RTD		$\sim_4$		2	44h	RTD	1

{WAIGRP}

PFI		Sigmsk$_4$	0	02h	PFI	1
IRQ		$\sim_4$	1	02h	IRQ	1
WAI		Sigmsk$_4$	2	02h	WAI	1
IRQ		$\sim_4$	3	02h	IRQ	1

{STPGRP}

STP		$\sim_4$	0	43h	STP	1
NOP		$\sim_4$	1	43h	NOP	1
MRK		Const$_4$	2	43h	MRK	1
		$\sim_4$	3	43h		
MEMSB		0	3	43h		
MEMDB		1	3	43h		
SYNC		2	3	43h		
FSYNC		3	3	43h		

BEQ		$Disp_{12}$	Cr_3	P_2	0	40h	BEQ disp	2
BNE		$Disp_{12}$	Cr_3	P_2	1	40h	BNE disp	2
BGT		$Disp_{12}$	Cr_3	P_2	2	40h	BGT disp	2
BLT		$Disp_{12}$	Cr_3	P_2	3	40h	BLT disp	2
BGE		$Disp_{12}$	Cr_3	P_2	0	41h	BGE disp	2
BLE		$Disp_{12}$	Cr_3	P_2	1	41h	BLE disp	2
BRA		$Disp_{12}$	Cr_3	3_2	2	41h	BRA disp	2

{BRKGRP}

RST		$\sim_4$		3	00h	RST	1
NMI		$\sim_2$	0	2	00h	NMI	1
SNR		$\sim_2$	1	2	00h	SNR	1
IRQ		~	i_3	1	00h	IRQ	1
BRK		$Const_4$		0	00h	BRK	1

Shuffle

PERM										Opcode		Bytes
51 39	38 26	25 22				21 17	16 12	11 7		6 0		
		0	$\sim_3$			Rb_5	Ra_5	Rt_5		20h	Rt,Ra,Rb	2
		1	S3	S2	S1	S0	Ra_5	Rt_5		20h	Rt,Ra,#imm_8	2

Control and Status Register Access

CSR	38 36	3533	32 29	28 17	16 12	11 7	6 0		
	OP_3	OL_3	$\sim_4$	$Regno_{12}$	Ra_5	Rt_5	01h	CSR	3

Register Move

MOV							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	$\sim_3$	0_5	Ra_5	Rt_5	09h	Rt,Ra,Rb	2

MT								Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11	7	6 0		
		0	$\sim_3$	$\sim_5$	Ra_5	0_3	L_2	4Ah	Lt,Ra	2

MT								Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11	7	6 0		
		0	$\sim_3$	$\sim_5$	Ra_5	1_2	C_3	4Ah	Ct,Ra	2

MF								Opcode		Bytes
51 39	38 26	25	22	21 17	16	12	11 7	6 0		
		0	$\sim_3$	$\sim_5$	0	L_2	Rt_5	5Ah	Rt,La	2

MT								Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11	7	6 0		
		0	$\sim_3$	$\sim_5$	Ra_5	1_3	3_2	4Ah	Rt,CA	2

Floating Point

FADD							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	Fb_5	Fa_5	Ft_5	4Fh	FADD Ft,Fa,Fb	2

FSUB							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	Fb_5	Fa_5	Ft_5	5Fh	FSUB Ft,Fa,Fb	2

FMUL							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	Fb_5	Fa_5	Ft_5	6Fh	FMUL Ft,Fa,Fb	2

FDIV							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	Fb_5	Fa_5	Ft_5	7Fh	FDIV Ft,Fa,Fb	2

FCMP							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	$\sim_3$	Fb_5	Fa_5	1_2 Ct_3	7Eh	FCMP Ct,Fa,Fb	2

{FLT1}							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	$Func_5$	Fa_5	Ft_5	6Eh	Ft,Fa	2

FMOV							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	0_5	Fa_5	Rt_5	6Eh	Rt,Fa	2
		1	Rm_3	0_5	Fa_5	Ft_5	6Eh	Ft,Fa	2

FMOV2							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	1_5	Ra_5	Ft_5	6Eh	Ft,Ra	2
		1	Rm_3	1_5	Fa_5	Ft_5	6Eh	Ft,Fa	2

FTOI							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	2_5	Fa_5	Rt_5	6Eh	Rt,Fa	2
		1	Rm_3	2_5	Fa_5	Ft_5	6Eh	Ft,Fa	2

ITOF							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	3_5	Ra_5	Ft_5	6Eh	Ft,Ra	2
		1	Rm_3	3_5	Fa_5	Ft_5	6Eh	Ft,Fa	2

FSIGN							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	$\sim_3$	6_5	Ra_5	Ft_5	6Eh	Ft,Fa	2
		1	$\sim_3$	6_5	Fa_5	1_2 Ct_3	6Eh	Ct,Fa	2

FMAN							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	7_5	Fa_5	Ft_5	6Eh	Ft,Ra	2

FSQRT							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		0	Rm_3	13_5	Fa_5	Ft_5	6Eh	Ft,Ra	2

ISNAN							Opcode		Bytes
51 39	38 26	25	22	21 17	16 12	11 7	6 0		
		1	$\sim_3$	14_5	Fa_5	1_2 Ct_3	6Eh	Ct,Fa	2

FINITE								Opcode		Bytes
51 39	38 26	25 22		21 17	16 12	11 7		6 0		
		1	$\sim_3$	15_5	Fa_5	1_2	Ct_3	6Eh	Ct,Fa	2

UNORD								Opcode		Bytes
51 39	38 26	25 22		21 17	16 12	11 7		6 0		
		1	$\sim_3$	31_5	Fa_5	1_2	Ct_3	6Eh	Ct,Fa	2

FCLASS							Opcode		Bytes
51 39	38 26	25 22		21 17	16 12	11 7	6 0		
		0	$\sim_3$	30_5	Fa_5	Rt_5	6Eh	Rt,Fa	2

FTX							Opcode		Bytes
51 39	38 26	25 22		21 17	16 12	11 7	6 0		
		0	$\sim_3$	16_5	Fa_5	$Const_5$	6Eh		2

FCX								Opcode		Bytes
51 39	38 26	25 22			21 17	16 12	11 7	6 0		
		0	~	C	17_5	Fa_5	$Const_5$	6Eh		2

FEX								Opcode		Bytes
51 39	38 26	25 22			21 17	16 12	11 7	6 0		
		0	~	C	18_5	Fa_5	$Const_5$	6Eh		2

FDX								Opcode		Bytes
51 39	38 26	25 22			21 17	16 12	11 7	6 0		
		0	~	C	19_5	Fa_5	$Const_5$	6Eh		2

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	FMOV	FMOV2	FTOI	ITOF	FNEG	FABS	FSIGN	FMAN	FNABS				FSTAT	FSQRT	ISNAN	FINITE
1x	FTX	FCX	FEX	FDX	FRM	TRUNC									FCLASS	UNORD

Oddball

CACHE								Opcode		Bytes
51 39	38 26	25			17	16 12	11 7	6 0		
		0	$\sim_3$	dc_3	ic_2	Ra_5	$\sim_5$	7Ah	<cmd>,[Ra]	2

REX							Opcode		Bytes
51 39	38 33	32 29	28 26	25 13	12	11 7	6 0		
	$\sim_6$	IM_4	Tgt_3	PL_{13}	~	Ra_5	6Ah	Ra,#PL,#Tgt,#IM	3

Opcode Maps

Root Level

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	BRK	CSR	{WAIGRP}	DIV	ADD	SUB	CMP	CMPU	AND	OR	EOR		ASL	LSR	MUL	
1x	{string}	{string}	{VP}		ADD	SUB	CMP	CMPU	AND	OR	EOR		ROL	ROR	MUL	
2x	PERM				ADD	SUB	CMP	CMPU	AND	OR	EOR		{ISOP}	ASR	MUL	
3x																
4x	{Branch}	{Branch}	JAL	STP	{RTGRP}				JAL [R]		MTx		SEQ	SLT	FSLT	FADD
5x	LD	LDB	LDF		LDR	BIT		STF	ST	STB	MFx		SNE	SLE	FSLE	FSUB
6x	LD	LDB	LDF		STC	BIT		STF	ST	STB	REX		FSEQ	SLTU	{FLT1}	FMUL
7x	LD	LDB	LDF			BIT		STF	ST	STB	CACHE		FSNE	SLEU	FCMP	FDIV

{FLT1} Opcode 6Eh

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	FMOV	FMOV2	FTOI	ITOF	FNEG	FABS	FSIGN	FMAN	FNABS				FSTAT	FSQRT	ISNAN	FINITE
1x	FTX	FCX	FEX	FDX	FRM	TRUNC		FRES						FRSQRTE	FCLASS	UNORD

Appendix

Register Tags

As part of the internal workings of the core, it tracks registers using a seven-bit register tag associated with each register in the programming model.

Tag	Register	
0 to 31	r0 to r31 general-purpose integer registers	
32 to 63	f0 to f31 general-purpose floating-point registers	
64 to 95	reserved for vector registers	
96 to 100	link registers	
101	machine status register	
102	reserved for vector length	
103	packed compare results	
104 to 111	compare results registers	
112 to 119	reserved for vector mask registers	
120	supervisor SP	
121	hypervisor SP	
122	machine SP	