FISA64

Programming Model

General Purpose Register Array

There is an array of 32, 64 bit general purpose integer registers.

	Usage
r0	always zero
r1	return value
r2	
r3	temporary register
r4	temporary register
r5	temporary register
r6	temporary register
r7	temporary register
r8	temporary register
r9	temporary register
r10	temporary register
r11	register var
r12	register var
r13	register var
r14	register var
r15	register var
r16	register var
r17	register var
r18	register var
r19	
r20	
r21	
r22	
r23	
r24	task register (TR)[1]
r25	thread pointer
r26	global pointer
r27	frame pointer (BP)
r28	catch link address (XLR)
r29	
r30	stack pointer (SP)
r31	return address (LR)

1 Not updateable in user mode.

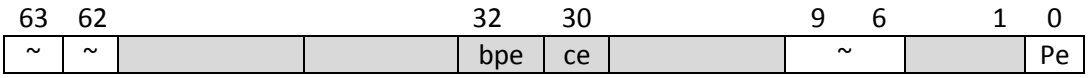
Special Purpose Registers

Most special purpose registers are accessible only in kernel mode. A privilege violation will result if attempting to access a special purpose register in user mode that is not available to that mode.

There are no results forwarding on the update of a special purpose register. If the value of the register is required immediately in the following few instructions, then some provision must be made to allow the special purpose register to update. This can be done by following a move to the spr with a couple of NOP instructions. Alternately a branch to a delay subroutine could be performed.

Control Register Zero (SPR 00 or CR0)

This register contains a bit to enable protected mode.



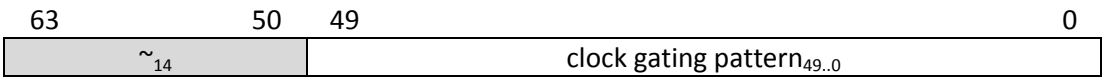
PE: Protected Mode enable: 1 = enabled, 0 = disabled.
CE: cache enable: 1=enabled, 0 = disabled
bpe: branch predictor enable: 1=enabled, 0=disabled

Tick Count Register (SPR 04 or TICK)

This register contains a count of the number of clock cycles that have passed since the last time the processor was reset. Tick may be used for high-resolution timing or performance measurement.

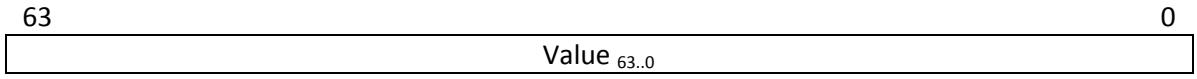
Clock Register (SPR 06)

The clock register controls clock gating to the processor to allow lower power consumption. Gating is controlled with a bit pattern which is fed to a clock enable gate. The pattern is 50 bits long, allowed clock control (or power control) in 2% increments. For example loading the register with h2AAAAAAAAAAA will cause every other clock to be gated off, reducing the effective operating frequency of the core in half. Loading the register with a zero will stop the clock completely. However, a non-maskable interrupt or reset will reload the clock register with all ones, causing the processor to operate at maximum frequency.



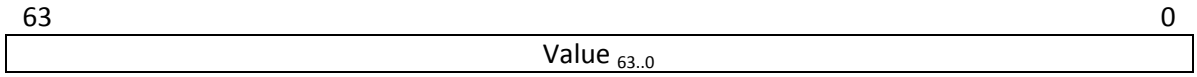
DBPC (SPR07)

This register stores the return address for a debug interrupt processing routine. This register is automatically loaded when a debug interrupt occurs. The program counter is loaded from this register automatically as part of the RTD instruction processing.



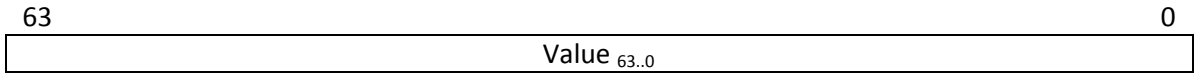
IPC (SPR08)

This register stores the return address for a hardware interrupt (NMI / IRQ) processing routine. This register is automatically loaded when a hardware interrupt occurs. The program counter is loaded from this register automatically as part of the RTI instruction processing.



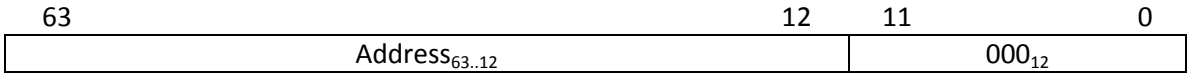
EPC (SPR09)

This register stores the return address for a software exception processing routine (OVERFLOW / privilege violation). This register is automatically loaded when a software exception occurs. The program counter is loaded from this register automatically as part of the RTE instruction processing.

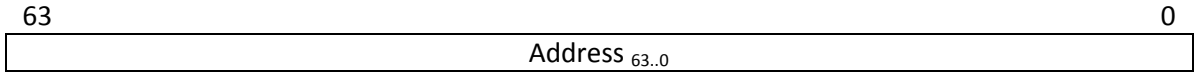


Interrupt Vector Table Base Address (SPR 10 or VBR)

This register contains the physical base address of the interrupt vector table in memory. The Table is 4kB aligned.

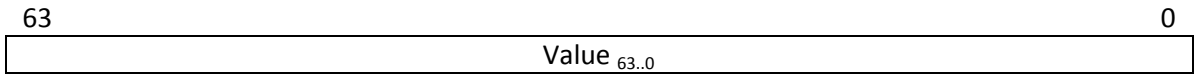


Interrupt vector table entries are 64 bits in size.



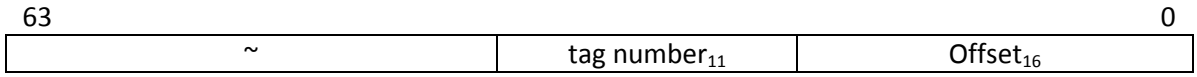
MULH (SPR14)

This register contains the high order bits of the multiplier product. It is available to both kernel and user modes.



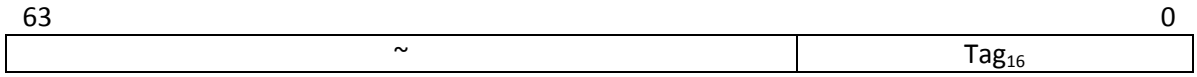
EA (SPR40)

This register holds the effective address associated with a memory tag. The tag number is contained in bits 16 to 26. The tag associated with this address will be accessible in the TAGS special purpose register. Note that this register and following tag access should be executed with interrupts disabled to prevent the effective address from changing before the tag is updated or read. Also no memory operation should occur between setting this register and updating or reading the tag. This register also reflects the latest effective address calculated by the processor and will be automatically updated when a memory operation occurs.



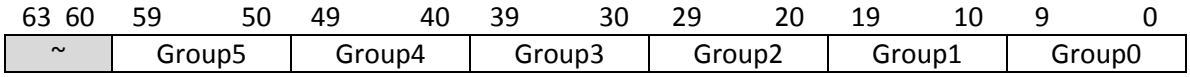
TAGS (SPR41)

This register makes the tag value accessible for update or read-back. It is used in association with the EA special purpose register. Writing this register will update the tag identified in the EA register.



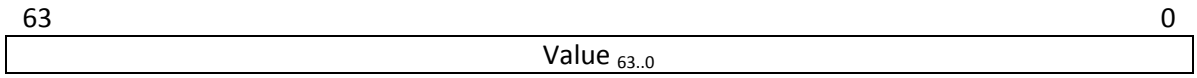
LOTGRP (SPR 42)

This register contains a list of memory groups that the process belongs to. The owning group associated with a memory tag is compared to this list during a memory access. If the group is in the list then the memory access is allowed, otherwise a memory fault exception occurs. This comparison takes place only in user mode; in kernel mode the kernel owns all of memory so the memory access is always allowed.



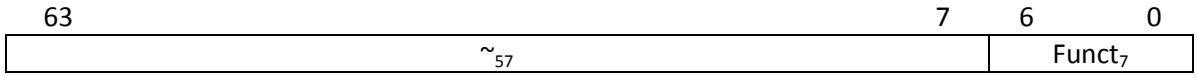
Compare and Swap (SPR44 or CAS)

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.



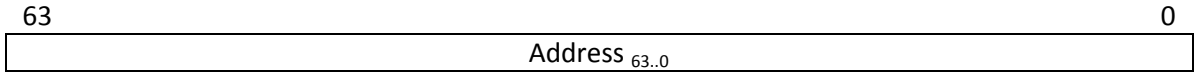
MYST (SPR45)

This register is to supports the MYST instruction. During execution of the MYST instruction the function code of the operation to be performed is loaded from this register. The MYST register is available to both user and kernel modes.



Debug Address Register (SPR50 to SPR53 or DBAD0 to DBAD3)

These registers contain addresses of instruction or data breakpoints.



Debug Control Register (SPR54)

These registers contains bits controlling the circumstances under which a debug interrupt will occur.

<table>
<tr><td>bits</td><td></td><td></td><td></td></tr>
<tr><td>3 to 0</td><td>Enables a specific debug address register to do address matching. If the corresponding bit in this register is set and the address (instruction or data) matches the address in the debug address register then a debug interrupt will be taken.</td><td></td><td></td></tr>
<tr><td>17, 16</td><td>This pair of bits determine what should match the debug address register zero in order for a debug interrupt to occur.
<table>
<tr><td>17:16</td><td></td><td></td></tr>
<tr><td>00</td><td>match the instruction address</td><td></td></tr>
<tr><td>01</td><td>match a data store address</td><td></td></tr>
<tr><td>10</td><td>reserved</td><td></td></tr>
<tr><td>11</td><td>match a data load or store address</td><td></td></tr>
</table></td><td></td><td></td></tr>
<tr><td>19, 18</td><td>This pair of bits determine how many of the address bits need to match in order to be considered a match to the debug address register. These bits are ignored when matching instruction addresses, which are always half-word aligned.
<table>
<tr><td>19:18</td><td></td><td>Size</td></tr>
<tr><td>00</td><td>all bits must match</td><td>byte</td></tr>
<tr><td>01</td><td>all but the least significant bit should match</td><td>char</td></tr>
<tr><td>10</td><td>all but the two LSB’s should match</td><td>half</td></tr>
<tr><td>11</td><td>all but the three LSB’s should match</td><td>word</td></tr>
</table></td><td></td><td></td></tr>
<tr><td>23 to 20</td><td>Same as 16 to 19 except for debug address register one.</td><td></td><td></td></tr>
<tr><td>27 to 24</td><td>Same as 16 to 19 except for debug address register two.</td><td></td><td></td></tr>
<tr><td>31 to 28</td><td>Same as 16 to 19 except for debug address register three.</td><td></td><td></td></tr>
<tr><td>62</td><td>This bit is a history bit for single stepping mode. The debug interrupt records bit 63 into bit 62 when a debug interrupt occurs. Then turns off SSM by writing a zero to bit 63. On return from debug routine (RTD) this bit is restored into bit 63 re-enabling SSM.</td><td></td><td></td></tr>
<tr><td>63</td><td>This bit enables SSM (single stepping mode)</td><td></td><td></td></tr>
</table>

Debug Status Register (SPR55)

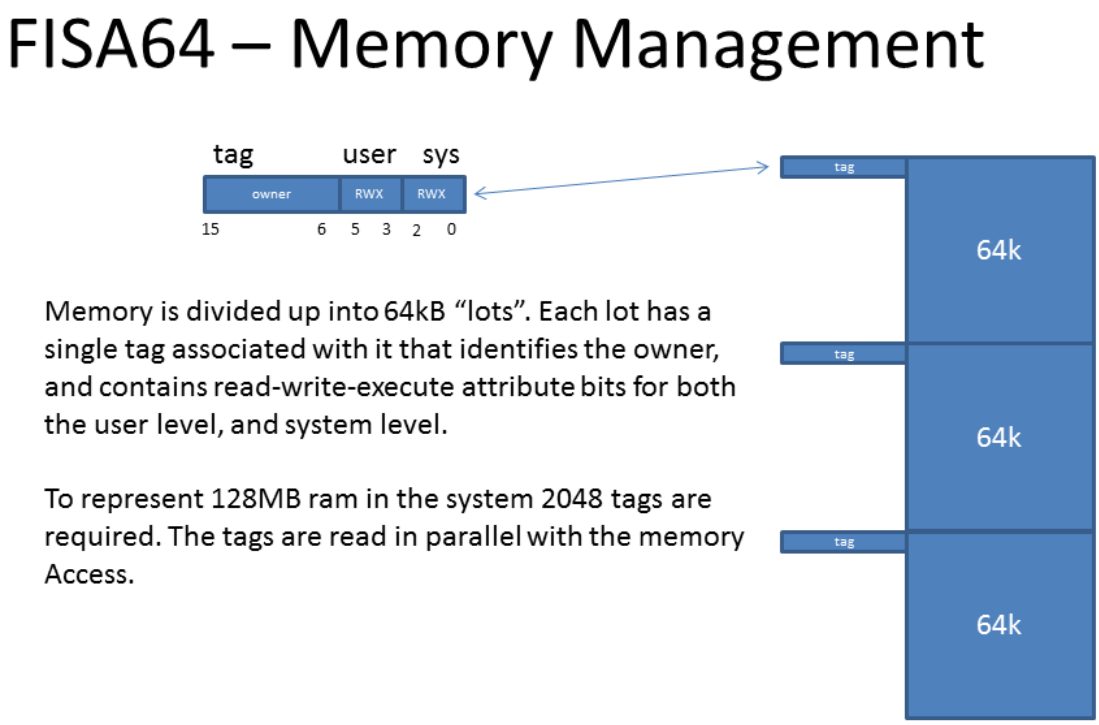
This register contains bits indicating which addresses matched. These bits are set when an address match occurs, and must be reset by software.

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

Memory Protection System

A key feature required to increase system reliability and robustness is memory protection. Memory should be protected against inadvertent access by the process that doesn't own a particular piece of memory. The system used here provides memory protection, but not address virtualization.

Memory is organized into lots which are 64kB in size. Memory is protected using a system of tags associated with each lot of memory. The tag associated with a memory lot contains the lot owner's group, and read / write / execute indicators.



The lot owner field in the memory tag represents a group of processes which may access the memory lot. Each process in the system may be associated with up to six memory groups. Which memory groups the process is a part of is stored in the LOTGRP special purpose register.

Interrupts

FISA64 uses a vectored interrupt system with support for 512 interrupt vectors.

Interrupt Vector Table Usage

The following table outlines which vector is used for a given purpose. These vectors are specific to FISA64. Under the HW column an 'x' indicates that the interrupt is internally generated by the processor; the vector is hard-wired to that use. An 'e' indicates an externally generated interrupt, the usage may vary depending on the system.

Vecno		HW	Description	
0				
1				
2			FMTK Scheduler	
3			debug interrupt	
4			OS API call	
449	KRST	e	Keyboard reset interrupt	
450	MSI	e	Millisecond Interrupt	
451	TICK	e	FMTK Tick Interrupt	
463	KBD	e	Keyboard interrupt	
487	BND	x	Bounds check exception	
488	DBZ	x	divide by zero	
489	OFL	x	overflow	
493	FLT	x	floating point exception	
494	TAP	x	debug tap interrupt	
495	SSM	x	single-step interrupt	
496	BPT	x	breakpoint	
497	EXF	x	Executable fault	
498	DWF	x	Data write fault	
499	DRF	x	data read fault	
501	PRIV	x	privilege level violation	
508	DBE	x	data bus error	
509	IBE	x	instruction bus error	
510	NMI	x	Non-maskable interrupt	

Instruction Set Description

A description of the instruction set follows.

ADD - addition

ADD Rt, Ra, #i15

ADD Rt, Ra, Rb

ADDU Rt, Ra, #i15

ADDU Rt, Ra, Rb

Instruction Formats:

04_7h	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	ADD Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	04_7	ADD Rt,Ra,#imm
14_7h	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	ADDU Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	14_7	ADDU Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra + $immediate_{15}$

Register-Register Form

Rt = Ra + Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

Currently the ADD and ADDU instruction both operate the same way. The distinction between the ADD and ADDU instructions is that the ADD instruction may cause an overflow exception, while the ADDU instruction never will.

AND – bitwise logical 'and'

AND Rt, Ra, #i15
AND Rt, Ra, Rb

Instruction Formats:

$0C_7$	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	AND Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	$0C_7$	AND Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra & $immediate_{15}$

Register-Register Form

Rt = Ra & Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

ASL – Arithmetic Shift Left

ASL Rt, Ra, #i6
ASL Rt, Ra, Rb

Instruction Formats:

30_7		Rb_5	Rt_5	Ra_5	02_7	ASL
38_7		Imm_6	Rt_5	Ra_5	02_7	ASL #

Operation:

Register Immediate Form

Rt = Ra << $immediate_6$

Register-Register Form

Rt = Ra << Rb

Notes:

The least significant bits are loaded with zeros.

ASR – Arithmetic Shift Right

ASR Rt, Ra, #i6
ASR Rt, Ra, Rb

Instruction Formats:

34_7	$\sim_3$	Rb_5	Rt_5	Ra_5	2_7	ASR Rt, Ra, Rb
$3C_7$	$\sim_2$	Imm_6	Rt_5	Ra_5	2_7	ASR Rt, Ra, #i6

Operation:

Register Immediate Form

Rt = Ra >> $immediate_6$

Register-Register Form

Rt = Ra >> Rb

Notes:

Performs an arithmetic shift right, preserving the sign bit of the value.

BFCHG – Bitfield Change

2_3	me_6	mb_6	Rt_5	Ra_5	$03h_7$

Description:

Inverts the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

BFCLR – Bitfield Clear

1_3	me_6	mb_6	Rt_5	Ra_5	$03h_7$

Description:

Sets the bits to zero of the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

BFEXT – Bitfield Extract

5_3	me_6	mb_6	Rt_5	Ra_5	$03h_7$

Description:

Extracts a bitfield from register Ra located between the mask begin (mb) and mask end (me) bits and places the sign extended result into the target register. This instruction may be used to sign extend a value beginning at any bit.

BFEXTU – Bitfield Extract Unsigned

6_3	me_6	mb_6	Rt_5	Ra_5	$03h_7$

Description:

Extracts a bitfield from register Ra located between the mask begin (mb) and mask end (me) bits and places the zero extended result into the target register. This instruction may be used to zero extend a value beginning at any bit.

BFINS – Bitfield Insert

3_3	me_6	mb_6	Rt_5	Ra_5	$03h_7$
4_3	me_6	mb_6	Rt_5	Imm_5	$03h_7$

Description:

Inserts a bitfield into the target register located between the mask begin (mb) and mask end (me) bits from the low order bits of Ra or an immediate value.

BFSET – Bitfield Set

0_3	me_6	mb_6	Rt_5	Ra_5	$03h_7$

Description:

Sets the bits to one of the bitfield in Ra located between the mask begin (mb) and mask end (me) bits and stores the result in the target register.

Bcc – Branches

Bcc Ra,target_address

Instruction Formats:

Operation:

If (condition) PC= PC +{displacement,2'b00}

Notes:

Branches are relative to the current program counter. A branch is taken to the target address if the condition is true. Branches may branch forwards or backwards up to 64kB in range. The unused bits in the instruction should be set to zero.

The branch condition tests a register against the value zero.

Op_3	Mne.	
0	BEQ	branch if equal to zero
1	BNE	branch if not equal
2	BGT	branch if greater than
3	BGE	branch if greater or equal
4	BLT	branch if less than
5	BLE	branch if less or equal
6		reserved
7		reserved

BRA – Branch Unconditionally

BRA target_address

Instruction Formats:

$Disp_{25}$	$3A_7h$	Bcc address

Operation:

PC= PC +{displacement,2'b00}

Notes:

Branches relative to the current program counter. A branch is taken to the target address if the condition is true. Branches may branch forwards or backwards up to 64MB in range.

BRK – Breakpoint

BRK address

Instruction Formats:

H_2	$\sim_4$	$Vector_9$	$\sim_{10}$	38_7	BRK

Operation:

```
if (h = 0)
    epc = pc
else if (h = 1)
     dbpc = pc
else if (h = 2)
     ipc = pc
PC = vbr + vector * 16
```

Notes:

Perform an interrupt, exception or debug handler. The handler type is indicated by the 'H' field of the instruction. The BRK instruction is used by hardware interrupts to call a hardware interrupt processing routine. The appropriate return instruction (RTE, RTD, or RTI) should be used to return from the BRK handler. The BRK instruction causes the processor to switch to kernel mode.

BSR – Branch to Subroutine

BSR target

Instruction Formats:

$Disp_{25}$	39_7h	BSR address

Operation:

Relative Address Form

PC = PC+sign extend({Displacement,2'b00})

Branch to a subroutine using program counter relative addressing. The displacement field of the instruction is shifted left twice before being used. The subroutine must be within +/-64MB of the current program counter.

CAS – Compare and Swap

CAS R1,R2,d[R4]

Instruction Format:

$Disp_{15}$	Rst_5	Ra_5	$6C_7h$	CAS

Operation:

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

Description:

If the contents of the addressed memory cell is equal to the contents of CAS special purpose 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 CAS. 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 LWAR and SWCR may also be used to perform a similar function where the memory system does not support bus locks, but support address reservations instead.

Assembler:

CAS Rt,Rt,displacement[Ra]

CHK – Check and Exception

CHK Rt, Ra, Bn

Instruction Formats:

$1A_7$	$\sim_3$	Rb_5	Rc_5	Ra_5	02_7	CHK Ra,Rc,Rb
Imm_{15}			Rc_5	Ra_5	$0B_7$	CHK Ra,Rc,#n

Operation:

if not (Ra >= Rc and Ra < Rb or immediate)
  take bounds execption

Notes:

This instruction may be used to validate a pointer or array index.

A register is checked against the upper and lower bounds contained in a register identified by the instruction. If the register is not between the upper and lower bounds then a bounds check exception is taken.

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

CMP - Comparison

CMP Rt, Ra, #i16
CMP Rt, Ra, Rb

Instruction Formats:

06_7	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	CMP Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	06_7	CMP Rt,Ra,#imm

Operation:

Register Immediate Form

```
if (Ra < immediate)
   Rt = -1
else if (Ra = immediate)
   Rt = 0
else
   Rt = 1
```

Register-Register Form

```
if (Ra < Rb)
   Rt = -1
else if (Ra = Rb)
   Rt = 0
else
   Rt = 1
```

Notes:

CMP performs a signed comparison of operands and sets the target register to -1, 0, or +1 if the first operand is less than, equal to, or greater than the second respectively.

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

CMPU – Unsigned Comparison

CMPU Rt, Ra, #i16
CMPU Rt, Ra, Rb

Instruction Formats:

16_7	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	CMP Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	16_7	CMP Rt,Ra,#imm

Operation:

Register Immediate Form

```
if (Ra < immediate)
   Rt = -1
else if (Ra = immediate)
   Rt = 0
else
   Rt = 1
```

Register-Register Form

```
if (Ra < Rb)
   Rt = -1
else if (Ra = Rb)
   Rt = 0
else
   Rt = 1
```

Notes:

CMP performs a signed comparison of operands and sets the target register to -1, 0, or +1 if the first operand is less than, equal to, or greater than the second respectively.

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

COM – bitwise ones complement

COM Rt, Ra

Instruction Formats:

-1_{15}	Rt_5	Ra_5	$0E_7$	COM Rt, Ra

Operation:

Register-Register Form

Rt = ~Ra

Notes:

All the bits in Ra are inverted and placed into the target register Rt. This is an alternate mnemonic for the EOR instruction.

CPUID – Processor Identification

CPUID Rt, Ra, #n

Instruction Formats:

Operation:

Register-Register Form

Rt = Processor Info Table[Ra|#n]

Notes:

The CPUID instruction returns information about the processor. The contents of register Ra and a four bit immediate value are OR'd together to form an index into the information table. One or the other of register Ra or the immediate value should be zero.

Index	bits	Information Returned
0	15 to 0	The processor core number. This field is determined from an external input. It would be hard wired to the number of the core in a multi-core system.
	23 to 16	Processor chip number. On a motherboard with multiple chips this identifies the chip the core is located in. It is typically hardwired to zero.
	31 to 24	Board number. The number of the processor board in a system with more than one board.
	39 to 32	Box number, which box on a rack contains the processor.
2	63 to 0	Manufacturer name first eight chars
3	63 to 0	Manufacturer name
4	63 to 0	CPU class
5	63 to 0	CPU class
6	63 to 0	CPU Name
7	63 to 0	CPU Name
8	63 to 0	Model Number
9	63 to 0	Serial Number
10	63 to 0	Features bitmap

DIV - Division
DIVU - Division

DIV Rt, Ra, #i15
DIV Rt, Ra, Rb
DIVU Rt, Ra, #i15
DIVU Rt, Ra, Rb

Instruction Formats:

08_7h	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	DIV Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	08_7	DIV Rt,Ra,#imm
18_7h	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	DIVU Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	18_7	DIVU Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra / $immediate_{15}$

Register-Register Form

Rt = Ra / Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

The signed registered form of the instruction may generate a divide by zero error if the divisor is zero. All other forms of the instruction including signed division by a constant never generate any exceptions.

EOR – bitwise logical exclusive 'or'

EOR Rt, Ra, #i15
EOR Rt, Ra, Rb

Instruction Formats:

$0E_7$	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	EOR Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	$0E_7$	EOR Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra ^ $immediate_{15}$

Register-Register Form

Rt = Ra ^ Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

IMM – Immediate Prefix

IMM #i25
IMM #i25

Instruction Formats:

Operation:

IMM1: constant buffer = sign extend ($immediate_{32}$)

IMM2: constant buffer[63:32] = $immediate_{32}$

Notes:

The IMM prefix appends 25 bits onto the 15 bit constant field of the following instruction then sign extends the resulting 40 bit constant out to 64 bits. Two immediate prefix instructions may be used in succession in order to append up to 49 bits onto the constant field of the following instruction. Thus a full 64 bit constant may be used by most instructions.

When debugging in single-step mode the immediate prefix is not treated as an independent instruction, rather it is an extension of the following instruction, so both the prefix and following instruction get executed in a single step.

The immediate prefix may not be used to extend the range of a branch instruction. If there is an immediate prefix applied to an instruction that doesn't use a constant, then the prefix will be ignored.

INC – Increment memory word

INC d(Rn),#n

Instruction Formats:

$Displacement_{15}$	Imm_5	Ra_5	64_7h	INC d15(Rn),#n

Operation:

Register Indirect with Displacement Form

memory[displacement + Ra] = memory[displacement + Ra] + n

Notes:

Increments the memory word by a signed five bit immediate constant. The displacement constant may be extended up to 64 bits with immediate prefix instructions.

JMP – Jump

JMP (abs,Rn)
JMP d(Rn)

Instruction Formats:

$Immed_{15}$	0_5	Ra_5	$3C_7$	JMP
$Immed_{15}$	0_5	Ra_5	$3E_7$	JMPI

Operation:

Memory Indexed Indirect Form

PC = memory[address + Rn]

Register Indirect with Displacement Form

PC = displacement + Rn

Notes:

The address constant may be extended up to 64 bits with immediate prefix instructions.

This instruction is an alternate mnemonic for the JAL / JALI instruction, where the target register is specified as zero.

JSR – Jump to Subroutine

JSR (abs,Rn)
JSR d(Rn)

Instruction Formats:

$Immed_{15}$	$1F_5$	Ra_5	$3C_7$	JSR
$Immed_{15}$	$1F_5$	Ra_5	$3E_7$	JSRI

Operation:

Memory Indexed Indirect Form

LR = PC
PC = memory[address + Rn]

Register Indirect with Displacement Form

LR = PC
PC = displacement + Rn

Notes:

The address constant may be extended up to 64 bits with immediate prefix instructions.

This instruction is an alternate mnemonic for the JAL / JALI instruction, where the target register is specified as thrity-one (the link register).

LB – Load Byte with Sign Extend

LBX – Load Byte with Sign Extend

LB Rt, d(Rn)
LB Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rt_5	Ra_5	40_7h	LB Rt,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rt_5	Ra_5	48_7h	LB Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

LBU – Load Byte with Zero Extend

LBUX – Load Byte with Zero Extend

LBU Rt, d(Rn)
LBU Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rt_5	Ra_5	41_7h	LBU Rt,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rt_5	Ra_5	49_7h	LBU Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

Rt = zero extend(memory[displacement + Ra])

Register-Register Form

Rt = zero extend(memory[offset + Ra + Rb * scale])

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

LC – Load Character with Sign Extend

LCX – Load Character with Sign Extend

LC Rt, d(Rn)
LC Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rt_5	Ra_5	42_7h	LC Rt,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4A_7h$	LC Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

This instruction loads a sixteen bit value from memory and sign extends it to sixty-four bits.

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

LCU – Load Character with Zero Extend

LCUX – Load Character with Zero Extend

LCU Rt, d(Rn)
LCU Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rt_5	Ra_5	43_7h	LCU Rt,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4B_7h$	LCU Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

Rt = zero extend(memory[displacement + Ra])

Register-Register Form

Rt = zero extend(memory[offset + Ra + Rb * scale])

Notes:

A sixteen bit value is loaded from memory, zero extended and placed in the target register.

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

LEA – Load Effective Address

LEA Rt,d(Ra)
LEA Rt, d(Ra + Rb * scale)

Instruction Formats:

$Disp_{15}$			Rt_5	Ra_5	47_7	LEA
$Offs_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4F_7$	LEAX

Operation:

Indexed Form

Rt = address of ($memory_{32}$[offset + Ra + Rb * scale])

Notes:

This instruction loads the target register with the address of the memory determined by the indexing operation.

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

LH – Load Half-word with Sign Extend

LHX – Load Half-word with Sign Extend

LH Rt, d(Rn)
LH Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rt_5	Ra_5	44_7h	LH Rt,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4C_7h$	LH Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

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

Register-Register Form

Rt = sign extend(memory[offset + Ra + Rb * scale])

Notes:

This instruction loads a thirty-two bit value from memory and sign extends it to sixty-four bits.

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

LHU – Load Half-word with Zero Extend

LHUX – Load Half-word with Zero Extend

LHU Rt, d(Rn)
LHU Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rt_5	Ra_5	45_7h	LHU Rt,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4D_7h$	LHU Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

Rt = zero extend(memory[displacement + Ra])

Register-Register Form

Rt = zero extend(memory[offset + Ra + Rb * scale])

Notes:

A thirty-two bit value is loaded from memory, zero extended and placed in the target register.

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

LW – Load Word

LWX – Load Word

LW Rt, d(Rn)
LW Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rt_5	Ra_5	46_7h	LW Rt,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4E_7h$	LW Rt,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

Rt = memory[displacement + Ra]

Register-Register Form

Rt = memory[offset + Ra + Rb * scale]

Notes:

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

LWAR – Load Word and Reserve

LWAR Rt, d(Rn)

Instruction Formats:

$Displacement_{15}$	Rt_5	Ra_5	$5C_7h$	LWAR Rt,d15(Rn)

Operation:

Register Indirect with Displacement Form

Rt = memory[displacement + Ra]

Notes:

This instruction performs the same operation as a load word (LW) instruction except that it sets the sr_o output signal during the load. The sr_o output signal can be used to set a memory reservation. LWAR is useful for implementing semaphores.

There is no indexed form of this instruction.

The displacement constant may be extended up to 64 bits with immediate prefix instructions. The offset constant for indexed mode may not be extended.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

MFSPR – Move from Special Register

MFSPR Rt,Spr
MFSPR Rt,Ra

Instruction Formats:

$1F_7h$	Spt_8	Rt_5	0_5	02_7h	MFSPR Rt,Spr
$1F_7h$	0_8	Rt_5	Ra_5	02_7h	MFSPR Rt,Ra

Operation:

Sprt = Ra

Notes:

The general purpose register is loaded from the special purpose register. There are two forms of this instruction. The first form specifies the special purpose register using a constant field in the instruction, the second form specifies the special purpose register using another general purpose register.

MTSPR – Move to Special Register

MTSPR Sprt, Ra
MTSPR Rc,Ra

Instruction Formats:

$1E_7h$	$Sprt_8$	0_5	Ra_5	02_7h	MTSPR Sprt,Ra
$1E_7h$	0_8	Rc_5	Ra_5	02_7h	MTSPR Rc,Ra

Operation:

Sprt = Ra

Notes:

The general purpose register is moved to the special purpose register. There are two forms of this instruction. The first form specifies the special purpose register using a constant field in the instruction, the second form specifies the special purpose register using another general purpose register.

MUL - Multiplication
MULU - Multiplication

MUL Rt, Ra, #i15
MUL Rt, Ra, Rb
MULU Rt, Ra, #i15
MULU Rt, Ra, Rb

Instruction Formats:

07_7h	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	MUL Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	07_7	MUL Rt,Ra,#imm
17_7h	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	MULU Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	17_7	MULU Rt,Ra,#imm

Operation:

Register Immediate Form

Rt = Ra * $immediate_{15}$

Register-Register Form

Rt = Ra * Rb

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

ROL – Rotate Left

ROL Rt, Ra, #i6
ROL Rt, Ra, Rb

Instruction Formats:

32_7		Rb_5	Rt_5	Ra_5	02_7	ROL
$3A_7$		Imm_6	Rt_5	Ra_5	02_7	ROL #

Operation:

Register Immediate Form
Rt = Ra << $immediate_6$

Register-Register Form
Rt = Ra << Rb

Notes:

Most significant bits are rotated into the least significant bits.

RTD – Return from Debug

This instruction returns the processor from debug mode into the mode prior. The program counter is loaded with the value in the DBPC register.

RTI – Return from Interrupt

This instruction returns the processor from kernel mode into the mode prior. The program counter is loaded with the value in the IPC register.

RTL – Return From Leaf Subroutine

RTL #i15

Instruction Formats:

$Immed_{15}$	$1F_5$	$1E_5$	37_7	RTL

Operation:

PC = LR
SP = SP + Immediate

Notes:

This instruction is used to return from a leaf subroutine (A leaf subroutine does not call another routine). The link register is loaded into the program counter, then the stack pointer updated. The stack pointer may be adjusted in order to remove parameters from the stack. This instruction differs from the RTS instruction in that it doesn't pop the link register from the stack. As a result the immediate constant specified to adjust the stack pointer is eight less than would be used for the RTS instruction.

RTS – Return From Subroutine

RTS #i15

Instruction Formats:

$Immed_{15}$	$1F_5$	$1E_5$	$3B_7$	RTS

Operation:

LR = $memory_{63..2}$[SP]
PC = LR
SP = SP + Immediate

Notes:

This instruction is used to return from a subroutine. The link register is popped from the stack and loaded into the program counter, then the stack pointer updated. The stack pointer may be adjusted in order to remove parameters from the stack. In assembler code if an immediate value is specified it must include eight bytes for popping the link register. By default the immediate value is set to eight.

SB – Store Byte

SBX – Store Byte

SC Rt, d(Rn)
SC Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rs_5	Ra_5	60_8h	SB Rs,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rs_5	Ra_5	68_8h	SB Rs,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

memory[displacement + Ra] = Rs

Register-Register Form

memory[offset + Ra + Rb * scale] = Rs

Notes:

Store an eight bit value to memory.

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

Scc – Set Conditionally

SEQ Rt, Ra, #i15
SNE Rt, Ra, Rb

Instruction Formats:

Op_7	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	Scc Rt,Ra,Rb
$Immediate_{15}$			Rt_5	Ra_5	Op_7	Scc Rt,Ra,#imm

Operation:

Register Immediate Form

If (condition(Ra,Immediate))
  Rt = 1
else
  Rt = 0

Register-Register Form

If (condition(Ra,Rb))
  Rt = 1
else
  Rt = 0

Notes:

The immediate constant may be extended up to 64 bits with immediate prefix instructions.

This instruction sets the target register to a '1' or '0' based on whether or not the condition is true.

Op_7	Mnemonic	Conditional Test	
20	SEQ	Set true if operands are equal	
21	SNE	not equal	
28	SGT	first operand is greater than second (signed)	
29	SLE	less than or equal (signed)	
2A	SGE	greater than or equal (signed)	
2B	SLT	less than (signed)	
2C	SHI	higher (unsigned)	
2D	SLS	lower or same (unsigned)	
2E	SHS	higher or same (unsigned)	
2F	SLO	lower (unsigned)	

SC – Store Character

SCX – Store Character

SC Rt, d(Rn)
SC Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rs_5	Ra_5	61_8h	SC Rs,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rs_5	Ra_5	69_8h	SC Rs,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

memory[displacement + Ra] = Rs

Register-Register Form

memory[offset + Ra + Rb * scale] = Rs

Notes:

Store a sixteen bit value to memory. The memory access does not need to be aligned, but unaligned accesses will take longer to complete.

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

SH – Store Half-word

SHX– Store Half-word

SH Rt, d(Rn)
SH Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rs_5	Ra_5	62_8h	SH Rs,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rs_5	Ra_5	$6A_8h$	SH Rs,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form
memory[displacement + Ra] = Rs

Register-Register Form
memory[offset + Ra + Rb * scale] = Rs

Notes:

Store a thirty-two bit half-word to memory. The memory access does not need to be aligned, but unaligned accesses will take longer to complete.

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

STP – Stop Processor

STP

This instruction stops the processor placing it in low power mode by stopping the processor clock. The clock rate register is loaded with zero. The processor may begin processing again once a non-maskable interrupt occurs or a reset occurs. The processor may be slowed down without stopping the clock by adjusting the value in the clock rate register.

SW – Store Word

SWX – Store Word

SW Rt, d(Rn)
SW Rt, d(Ra + Rb * scale)

Instruction Formats:

$Displacement_{15}$			Rs_5	Ra_5	63_8h	SW Rs,d15(Rn)
$Offset_8$	Sc_2	Rb_5	Rs_5	Ra_5	$6B_8h$	SW Rs,d(Ra+Rb*sc)

Operation:

Register Indirect with Displacement Form

memory[displacement + Ra] = Rs

Register-Register Form

memory[offset + Ra + Rb * scale] = Rs

Notes:

Store a word to memory. The memory access does not need to be aligned, but unaligned accesses will take longer to complete.

The displacement constant may be extended up to 64 bits with immediate prefix instructions.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

SWCR – Store Word and Clear Reservation

SWCR Rt, d(Rn)

Instruction Formats:

$Displacement_{15}$	Rs_5	Ra_5	$6E_8h$	SWCR Rs,d15(Rn)

Operation:

Register Indirect with Displacement Form

if (address reserved)

memory[displacement + Ra] = Rs

cr0[36] = 1

else

cr0[36] = 0

Notes:

Conditionally store a word to memory if an address reservation is present. If successful bit 36 of cr0 will be set, otherwise bit 36 of cr0 will be cleared. This instruction sets the cr_o signal during execution. The memory system must be capable of aborting the store if there is no reservation present.

The memory access does not need to be aligned, but unaligned accesses will take longer to complete.

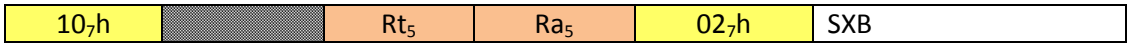
The displacement constant may be extended up to 64 bits with immediate prefix instructions.

Sc_2 Code	Multiply By
0	1
1	2
2	4
3	8

SXB – Sign Extend Byte

SXB Rt, Ra

Instruction Formats:



Operation:

Register Form

Rt = sign extend (Ra)

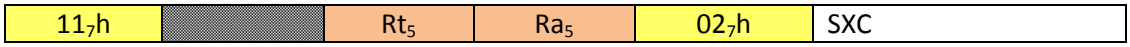
Notes:

The most significant bits (8 to 63) are loaded with the sign extension of bit 7.

SXC – Sign Extend Character

SXC Rt, Ra

Instruction Formats:



Operation:

Register Form

Rt = sign extend (Ra)

Notes:

The most significant bits (16 to 63) are loaded with the sign extension of bit 15.

WAI – Wait For Interrupt

WAI

Instruction Formats:

37_7		03_5	00_5	$\sim_5$	02_7	WAI

Operation:

if (no interrupt)
  PC = PC
else
  PC = PC + 4

Notes:

This instruction waits for an interrupt to occur before proceeding..

Sample Code

Register – Register Format Instructions

FISA64 includes a standard set of arithmetic and logical instructions including add / subtract / multiply/ divide / modulus / logical and / or / and exclusive or. Also present are shift instructions for both signed and unsigned operations.

The CMP instruction performs a signed comparison of two registers, or a register and immediate value and stores a -1, 0, or +1 in the target register if the first operand is less than, equal to or greater than the second operand respectively. The comparison result may be used by a following branch instruction. The CMPU instruction works the same way as CMP except that it performs an unsigned comparison. CMPU performs an unsigned comparison but produces a signed result.

Executing an RTI instruction enables interrupts. Interrupts may also be enabled and disabled with the CLI and SEI instructions. The RTI instruction also restored the processor mode (user or kernel) that was present before the interrupt. The processor does not support nested interrupts. However an interrupt may be processed during a software exception handler.

$Func_7$	$\sim_3$	Rb_5	Rt_5	Ra_5	02_7	{RR}
00_7		Rb_5	Rt_5	Ra_5	02_7	NAND
01_7		Rb_5	Rt_5	Ra_5	02_7	NOR
02_7		Rb_5	Rt_5	Ra_5	02_7	ENOR
04_7		Rb_5	Rt_5	Ra_5	02_7	ADD
05_7		Rb_5	Rt_5	Ra_5	02_7	SUB
06_7		Rb_5	Rt_5	Ra_5	02_7	CMP
07_7		Rb_5	Rt_5	Ra_5	02_7	MUL
08_7		Rb_5	Rt_5	Ra_5	02_7	DIV
09_7		Rb_5	Rt_5	Ra_5	02_7	MOD
$0A_7$		$\sim_5$	Rt_5	Ra_5	02_7	NOT
$0C_7$		Rb_5	Rt_5	Ra_5	02_7	AND
$0D_7$		Rb_5	Rt_5	Ra_5	02_7	OR
$0E_7$		Rb_5	Rt_5	Ra_5	02_7	EOR
10_7		~	Rt_6	Ra_6	02_7	SXB
11_7		~	Rt_6	Ra_6	02_7	SXC
12_7		~	Rt_6	Ra_6	02_7	SXH
14_7		Rb_5	Rt_5	Ra_5	02_7	ADDU
15_7		Rb_5	Rt_5	Ra_5	02_7	SUBU
16_7		Rb_5	Rt_5	Ra_5	02_7	CMPU
17_7		Rb_5	Rt_5	Ra_5	02_7	MULU
18_7		Rb_5	Rt_5	Ra_5	02_7	DIVU
19_7		Rb_5	Rt_5	Ra_5	02_7	MODU
$1A_7$		Rb_6	Rc_5	Ra_5	02_7	CHK
$1E_7$	Spr_8		Rc_5	Ra_5	02_7	MTSPR

$1F_7$	Spr_8	Rt_5	Ra_5	02_7	MFSPR
$2x_7$	Rb_5	Rt_5	Ra_5	02_7	Scc
20_7	Rb_5	Rt_5	Ra_5	02_7	SEQ
21_7	Rb_5	Rt_5	Ra_5	02_7	SNE
28_7	Rb_5	Rt_5	Ra_5	02_7	SGT
29_7	Rb_5	Rt_5	Ra_5	02_7	SLE
$2A_7$	Rb_5	Rt_5	Ra_5	02_7	SGE
$2B_7$	Rb_5	Rt_5	Ra_5	02_7	SLT
$2C_7$	Rb_5	Rt_5	Ra_5	02_7	SHI
$2D_7$	Rb_5	Rt_5	Ra_5	02_7	SLS
$2E_7$	Rb_5	Rt_5	Ra_5	02_7	SHS
$2F_7$	Rb_5	Rt_5	Ra_5	02_7	SLO
30_7	Rb_5	Rt_5	Ra_5	02_7	SLL
31_7	Rb_5	Rt_5	Ra_5	02_7	SRL
32_7	Rb_5	Rt_5	Ra_5	02_7	ROL
33_7	Rb_5	Rt_5	Ra_5	02_7	ROR
34_7	Rb_5	Rt_5	Ra_5	02_7	SRA
36_7	~ I_4	Rt_5	Ra_5	02_7	CPUID
37_7	00_5	00_5	$\sim_5$	02_7	CLI
37_7	01_5	00_5	$\sim_5$	02_7	SEI
37_7	02_5	00_5	$\sim_5$	02_7	STP
37_7	03_5	00_5	$\sim_5$	02_7	WAI
37_7	$1D_5$	$1E_5$	$\sim_5$	02_7	RTD
37_7	$1E_5$	$1E_5$	$\sim_5$	02_7	RTE
37_7	$1F_5$	$1E_5$	$\sim_5$	02_7	RTI
$Func_7$	Imm_6	Rt_5	Ra_5	02_7	Shifts #
38_7	Imm_6	Rt_5	Ra_5	02_7	SLL #
39_7	Imm_6	Rt_5	Ra_5	02_7	SRL #
$3A_7$	Imm_6	Rt_5	Ra_5	02_7	ROL #
$3B_7$	Imm_6	Rt_5	Ra_5	02_7	ROR #
$3C_7$	Imm_6	Rt_5	Ra_5	02_7	SRA #
40_7	$Pred_4$ $Succ_4$	0_5	$\sim_5$	02_7	FENCE

Register – Immediate Format Instructions

There are signed and unsigned versions of instructions. The mnemonics of the unsigned instructions are post-fixed with a 'U'.

ADD / SUB may generate an overflow exception when overflow occurs. ADDU / SUBU do not generate any exceptions.

$Immed_{15}$	Rt_5	Ra_5	04_7	ADD #
$Immed_{15}$	Rt_5	Ra_5	05_7	SUB #
$Immed_{15}$	Rt_5	Ra_5	06_7	CMP #
$Immed_{15}$	Rt_5	Ra_5	07_7	MUL #
$Immed_{15}$	Rt_5	Ra_5	08_7	DIV #
$Immed_{15}$	Rt_5	Ra_5	09_7	MOD #
$Immed_{15}$	Rt_5	$\sim_5$	$0A_7$	LDI #
$Immed_{15}$	Rc_5	Ra_5	$0B_7$	CHK #
$Immed_{15}$	Rt_5	Ra_5	$0C_7$	AND #
$Immed_{15}$	Rt_5	Ra_5	$0D_7$	OR #
$Immed_{15}$	Rt_5	Ra_5	$0E_7$	EOR #
$Immed_{15}$	Rt_5	Ra_5	14_7	ADDU #
$Immed_{15}$	Rt_5	Ra_5	15_7	SUBU #
$Immed_{15}$	Rt_5	Ra_5	16_7	CMPU #
$Immed_{15}$	Rt_5	Ra_5	17_7	MULU #
$Immed_{15}$	Rt_5	Ra_5	18_7	DIVU #
$Immed_{15}$	Rt_5	Ra_5	19_7	MODU #
$Immed_{15}$	Rt_5	Ra_5	$2x_7$	Scc #
$Immed_{15}$	Rt_5	Ra_5	20_7	SEQ #
$Immed_{15}$	Rt_5	Ra_5	21_7	SNE #
$Immed_{15}$	Rt_5	Ra_5	28_7	SGT #
$Immed_{15}$	Rt_5	Ra_5	29_7	SLE #
$Immed_{15}$	Rt_5	Ra_5	$2A_7$	SGE #
$Immed_{15}$	Rt_5	Ra_5	$2B_7$	SLT #
$Immed_{15}$	Rt_5	Ra_5	$2C_7$	SHI #
$Immed_{15}$	Rt_5	Ra_5	$2D_7$	SLS #
$Immed_{15}$	Rt_5	Ra_5	$2E_7$	SHS #
$Immed_{15}$	Rt_5	Ra_5	$2F_7$	SLO #

Bitfield Instructions

Bitfield						
Op_3	me_6	mb_6	Rt_5	Ra_5	03_7	BtFld
0_3	me_6	mb_6	Rt_5	Ra_5	03_7	BFSET
1_3	me_6	mb_6	Rt_5	Ra_5	03_7	BFCLR
2_3	me_6	mb_6	Rt_5	Ra_5	03_7	BFCHG
3_3	me_6	mb_6	Rt_5	Ra_5	03_7	BFINS
4_3	me_6	mb_6	Rt_5	Imm_5	03_7	BFINSI
5_3	me_6	mb_6	Rt_5	Ra_5	03_7	BFEXT
6_3	me_6	mb_6	Rt_5	Ra_5	03_7	BFEXTU
7_3	me_6	mb_6	Rt_5	Ra_5	03_7	

Flow Control Instructions

There are six relational branches which branch based on the result of a signed comparison of a register to zero. In order to branch based on an unsigned comparison, the CMPU instruction must be used prior to the branch. Since branches inherently compare a register to zero it is often possible to omit a preceding compare (CMP) operation. Branches branch relative to the program counter using a 17 bit signed displacement. This allows branching within +/- 64kB of the current program address.

The subroutine call instruction (BSR) stores the return address in the default link register – R31. The target address is specified as a 27 bit displacement from the current program counter.

In order to jump to a routine whose target address is computed in a register at run time, the JAL instruction is provided.

The BRA instruction works the same way as the BSR instruction, but doesn't store the return address.

The RTS instruction is used to return from a subroutine and de-allocate a stack frame at the same time.

The BRK instruction is used to transfer control to a kernel mode BRK handler. This is the means to communicate with the operating system. Hardware interrupts force an appropriate BRK instruction into the instruction stream.

The NOP instruction doesn't perform any operation.

Flow Control					
$Disp_{16..2}$	$\sim_2$	0_3	Ra_5	$3D_7$	BEQ
$Disp_{16..2}$	$\sim_2$	1_3	Ra_5	$3D_7$	BNE
$Disp_{16..2}$	$\sim_2$	2_3	Ra_5	$3D_7$	BGT
$Disp_{16..2}$	$\sim_2$	3_3	Ra_5	$3D_7$	BGE / BPL
$Disp_{16..2}$	$\sim_2$	4_3	Ra_5	$3D_7$	BLT / BMI
$Disp_{16..2}$	$\sim_2$	5_3	Ra_5	$3D_7$	BLE
H $\sim_4$ $Vector_9$	$\sim_5$		$1E_5$	38_7	BRK
$Offset_{26..2}$				39_7	BSR
$Offset_{26..2}$				$3A_7$	BRA
$Immed_{15}$	$1F_5$		$1E_5$	37_7	RTL
$Immed_{15}$	$1F_5$		$1E_5$	$3B_7$	RTS
$Immed_{15}$	Rt_5		Ra_5	$3C_7$	JAL
$Immed_{15}$	Rt_5		Ra_5	$3E_7$	JALI
$\sim_{25}$				$3F_7$	NOP

Memory Operate Instructions

FISA64 is a load / store / push / pop architecture.

There are two different instruction formats for memory operating instructions. These are register indirect with displacement format and scaled indexed addressing format.

Operand sizes of byte (8 bit), character (16 bit), half-word (32 bit) and word (64 bits) are supported. Sign and zero extension on load is available.

Loads and stores do not have to be aligned, however unaligned access will require additional clock cycles to complete.

Memory						64 bit
$Disp_{15}$			Rt_5	Ra_5	40_7	LB
$Disp_{15}$			Rt_5	Ra_5	41_7	LBU
$Disp_{15}$			Rt_5	Ra_5	42_7	LC
$Disp_{15}$			Rt_5	Ra_5	43_7	LCU
$Disp_{15}$			Rt_5	Ra_5	44_7	LH
$Disp_{15}$			Rt_5	Ra_5	45_7	LHU
$Disp_{15}$			Rt_5	Ra_5	46_7	LW
$Disp_{15}$			Rt_5	Ra_5	47_7	LEA
$Offs_8$	Sc_2	Rb_5	Rt_5	Ra_5	48_7	LBX
$Offs_8$	Sc_2	Rb_5	Rt_5	Ra_5	49_7	LBUX
$Offs_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4A_7$	LCX
$Offs_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4B_7$	LCUX
$Offs_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4C_7$	LHX
$Offs_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4D_7$	LHUX
$Offs_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4E_7$	LWX
$Offs_8$	Sc_2	Rb_5	Rt_5	Ra_5	$4F_7$	LEAX
8_{15}			Rt_5	$1E_5$	57_7	POP
$Disp_{15}$			Rt_5	Ra_5	$5C_7$	LWAR
$Disp_{15}$			Rs_5	Ra_5	60_7	SB
$Disp_{15}$			Rs_5	Ra_5	61_7	SC
$Disp_{15}$			Rs_5	Ra_5	62_7	SH
$Disp_{15}$			Rs_5	Ra_5	63_7	SW
$Disp_{15}$			Imm_5	Ra_5	64_7	INC
$Disp_{15}$			$1E_5$	Ra_5	65_7	PEA
$~_{15}$			$1E_5$	Ra_5	67_7	PUSH
$Disp_{15}$			$1E_5$	Ra_5	66_7	PUSH m
$Offs_8$	Sc_2	Rb_5	Rs_5	Ra_5	68_7	SBX
$Offs_8$	Sc_2	Rb_5	Rs_5	Ra_5	69_7	SCX
$Offs_8$	Sc_2	Rb_5	Rs_5	Ra_5	$6A_7$	SHX
$Offs_8$	Sc_2	Rb_5	Rs_5	Ra_5	$6B_7$	SWX
$Disp_{15}$			Rst_5	Ra_5	$6C_7$	CAS
$Disp_{15}$			Rs_5	Ra_5	$6E_7$	SWCR

Caveats

Branches

Branch instructions can't make proper use of an immediate prefix because they don't detect an immediate prefix at the If stage in order to keep the hardware simpler. (There is no requirement for conditional branching more than 15 bits). However a branch instruction just uses the same immediate value that is calculated for other instructions in the EX stage. This could lead to branches branching to two different locations if an immediate prefix is used for a branch.

For example if a prefix is used with a branch, BEQ *+$100010 for instance. Then the branch will branch to *+$10 if it is predicted taken, but to *+100010 if it's predicted not taken, then taken later in the EX stage.

If the branch is predicted taken, it'll branch using the 15 displacement field from the instruction. If the branch is predicted not taken, but is taken later in the EX stage, it'll branch using the full immediate value, which with prefixes could be up to 64 bits. The solution is that the assembler never outputs branches with prefixes. There is no hardware protection against using an immediate prefix with a branch.

In the IF stage ,rather than look at the previous instructions for an immediate prefix, the processor simply ignores the fact a prefix is present, and sign extends the branch displacement in the instruction without taking into account a prefix.

IF stage:

```
if (iopcode==`Bcc && predict_taken) begin

        pc <= pc + {{47{insn[31]}},insn[31:17],2'b00};   // Ignores potential immediate prefix

        dbranch_taken <= TRUE;

end
```

However, the EX stage uses a full immediate including any prefix, also to simplify hardware.

EX stage:

```
`Bcc:    if (takb & !xbranch_taken)
```

```
                update_pc(xpc + {imm,2'b00});   // This uses a “full” immediate value
        else if (!takb & xbranch_taken)
                update_pc(xpc + 64'd4);
```

Software Exceptions

For software type exceptions (divide by zero, overflow) the address stored in the EPC register is the address of the next instruction, not the current instruction address. The issue is that if a system call is being performed one wants to return the next instruction. Since system calls and other software exceptions share the same exception logic, for the usual usage the next instruction address is stored off. It is difficult to determine what the previous address might be as there could be a prefix instruction present.

Other Limitations

The task register can be read in user mode. This allows an application program to identify where in memory task control information is located. Ideally a user mode application should not be able to find out where operating system data is located. The task register is disabled from being updated by a user mode application so that the task isn’t inadvertently incorrectly switched.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x			{rr}	{bitfld}	ADD#	SUB#	CMP#	MUL#	DIV#	MOD#	LD#	CHK#	AND#	OR#	EOR#	
1x					ADDU#	SUBU#	CMPU #	MULU#	DIVU#	MODU#						
2x	SEQ#	SNE#			MYST				SGT#	SLE#	SGE#	SLT#	SHI#	SLS#	SHS#	SLO#
3x								RTL	BRK	BSR	BRA	RTS	JAL	Bcc	JALI	NOP
4x	LB	LBU	LC	LCU	LH	LHU	LW	LEA	LBX	LBUX	LCX	LCUX	LHX	LHUX	LWX	LEAX
5x	LFS	LFD	LFQ					POP	LFSX	LFDX	LFQX		LWAR			
6x	SB	SC	SH	SW	INC	PEA	PUSH m	PUSH r	SBX	SCX	SHX	SWX	CAS	PEAX	SWCR	
7x	SFS	SFD	SFQ						SFSX	SFDX	SFQX		IMM			

02 Group Func

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NAND	NOR	ENOR		ADD	SUB	CMP	MUL	DIV	MOD	NOT		AND	OR	EOR	
1x	SXB	SXC	SXH		ADDU	SUBU	CMPU	MULU	DIVU	MODU	CHK	CHKX			MTSPR	MFSPR
2x	SEQ	SNE							SGT	SLE	SGE	SLT	SHI	SLS	SHS	SLO
3x	SLL	SRL	ROL	ROR	ASR		CPUID	{ctrl}	SLLI	SRLI	ROLI	RORI	ASRI			
4x	FENCE															
5x																
6x	FADD	FSUB	FMUL	FDIV												
7x																