

FISA64

Programming Model

General Purpose Register Array

FISA64 has an array of 32, 64 bit general purpose integer registers, coupled with an array of 32, 64 bit floating point 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 (TP)
r26	global pointer (GP)
r27	frame pointer (BP)
r28	catch link address (XLR)
r29	
r30	stack pointer (SP)
r31	return address (LR)

fp0	
fp1	return value
fp2	
fp3	temporary register
fp4	temporary register
fp5	temporary register
fp6	temporary register
fp7	temporary register
fp8	temporary register
fp9	temporary register
fp10	temporary register
fp11	
fp12	
fp13	
fp14	
fp15	
fp16	
fp17	
fp18	
fp19	
fp20	
fp21	
fp22	
fp23	
fp24	
fp25	
fp26	
fp27	
fp28	
fp29	
fp30	
fp31	

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.

63	62			32	30		9	6	1	0
~	~			bpe	ce		~			Pe

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 h2AAAAAAAAAAAA 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.

63		50	49		0
~ ₁₄				clock gating pattern _{49..0}	

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.

63	0
Value _{63..0}	

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.

63	0
Value _{63..0}	

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.

63	0
Value _{63..0}	

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.

63	12	11	0
Address _{63..12}			000 ₁₂

Interrupt vector table entries are 64 bits in size.

63	0
Address _{63..0}	

MULH (SPR14)

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

63	0
Value _{63..0}	

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.

63			0
	~	tag number ₁₁	Offset ₁₆

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.

63		0
	~	Tag ₁₆

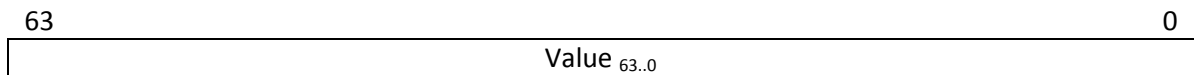
LOTGRP (SPR 42)

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.

63	60	59	50	49	40	39	30	29	20	19	10	9	0
~		Group5		Group4		Group3		Group2		Group1		Group0	

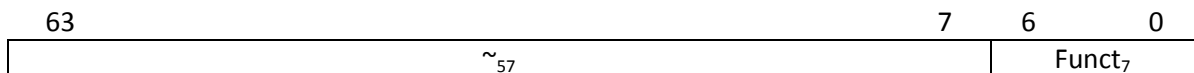
Compare and Swap (SPR44 or CAS)

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.



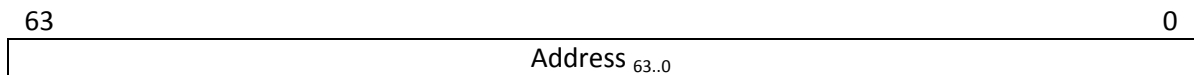
MYST (SPR45)

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.

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

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

Floating Point Status and Control Register –SPR20

The floating point status and control register may be read using the MFSPR instruction.

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

Floating Point Number Format

The floating point number format used by FISA64 is the IEEE standard double precision format:

1	1	10	52 + 1 hidden bit
S_m	S_e	EEEEEEEEEE	.MMMMMMMM.....MMMMMMMMMM

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_e EEEEEEEEEE	
1111111111	Maximum exponent
....	
0111111111	exponent of zero
....	
0000000000	Minimum exponent

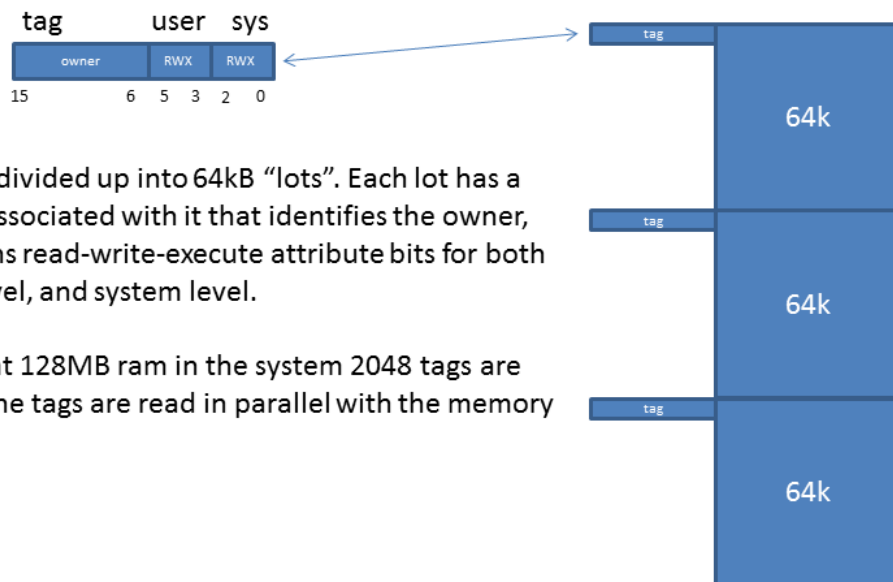
The exponent ranges from -1024 to +1023

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.

FISA64 – Memory Management



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 ₇ h	~ 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 ₇ h	~ 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

$Rt = Ra + \text{immediate}_{15}$

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	~ 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 & immediate₁₅

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

$Rt = Ra \ll \text{immediate}_6$

Register-Register Form

$Rt = Ra \ll Rb$

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 >> immediate₆

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:

Disp ₁₅	~ ₂	Op ₃	Ra ₅	3D ₇ h	Bcc address
--------------------	----------------	-----------------	-----------------	-------------------	-------------

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	BUN	branch if unordered (register[3..0] = 2)
7		reserved

The branch unordered (BUN) instruction checks only the least significant nybble to see if it contains the value 2. The instruction is used in association with a floating point comparison (FCMP).

BRA – Branch Unconditionally

BRA target_address

Instruction Formats:

Disp ₂₅	3A ₇ h	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	~ ₄	Vector ₉	~ 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 ₇ h	BSR address
--------------------	-------------------	-------------

Operation:

Relative Address Form

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

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 ₇ h	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	~ 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 exception

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	~ 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	~ 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	OE 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 ₇ h	~ 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 ₇ h	~ 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

$Rt = Ra / \text{immediate}_{15}$

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	~ 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

$Rt = Ra \wedge \text{immediate}_{15}$

Register-Register Form

$Rt = Ra \wedge Rb$

Notes:

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

FCMP – Floating Point Comparison

FCMP Rt, FPa, FPb

Instruction Formats:

62 ₇	~ ₃	Rb ₅	Rt ₅	Ra ₅	02 ₇	FCMP Rt,FPa,FPb
-----------------	----------------	-----------------	-----------------	-----------------	-----------------	-----------------

Operation:

Register-Register Form

if (unordered)

 Rt = 2

else if (Ra < Rb)

 Rt = -1

else if (Ra = Rb)

 Rt = 0

else

 Rt = 1

Notes:

FCMP performs a comparison between two floating point registers 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. If the comparison is unordered the target register is set to the value 2.

IMM – Immediate Prefix

IMM #i28

IMM #i28

Instruction Formats:



Operation:

IMM1: constant buffer = sign extend (immediate₃₂)

IMM2: constant buffer[63:32] = immediate₃₂

Notes:

The IMM prefix appends 28 bits onto the 15 bit constant field of the following instruction then sign extends the resulting 43 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 ₇ h	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 ₇ h	LB Rt,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	Rt ₅	Ra ₅	48 ₇ h	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 ₇ h	LBU Rt,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	Rt ₅	Ra ₅	49 ₇ h	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 ₇ h	LC Rt,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	Rt ₅	Ra ₅	4A ₇ h	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 ₇ h	LCU Rt,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	Rt ₅	Ra ₅	4B ₇ h
					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 (memory₃₂[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 ₇ h	LH Rt,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	Rt ₅	Ra ₅	4C ₇ h	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 ₇ h	LHU Rt,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	Rt ₅	Ra ₅	4D ₇ h	LHU Rt,d(Ra+Rb*scale)

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 ₇ h	LW Rt,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	Rt ₅	Ra ₅	4E ₇ h	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 ₇ h	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 ₇ h	Spt ₈	Rt ₅	0 ₅	02 ₇ h	MFSPR Rt,Spr
1F ₇ h	0 ₈	Rt ₅	Ra ₅	02 ₇ h	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 ₇ h	Sprt ₈	0 ₅	Ra ₅	02 ₇ h	MTSPR Sprt,Ra
1E ₇ h	0 ₈	Rc ₅	Ra ₅	02 ₇ h	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 _h	~ 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 _h	~ 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

$Rt = Ra \ll \text{immediate}_6$

Register-Register Form

$Rt = Ra \ll Rb$

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 ₈ h	SB RS,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	RS ₅	Ra ₅	68 ₈ h	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	~ ₃	Rb ₅	Rt ₅	Ra ₅	OZ 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 ₈ h	SC Rs,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	Rs ₅	Ra ₅	69 ₈ h	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 ₈ h	SH Rs,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	RS ₅	RA ₅	6A ₈ h
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 ₈ h	SW Rs,d15(Rn)
Offset ₈	Sc ₂	Rb ₅	RS ₅	RA ₅	6B ₈ h
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 ₈ h	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:

10 ₇ h		Rt ₅	Ra ₅	02 ₇ h	SXB
-------------------	--	-----------------	-----------------	-------------------	-----

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:

11 ₇ h		Rt ₅	Ra ₅	02 ₇ h	SXC
-------------------	--	-----------------	-----------------	-------------------	-----

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	~ 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	~ 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		~ 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
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	~ ₅	02 ₇	CLI
37 ₇		01 ₅	00 ₅	~ ₅	02 ₇	SEI
37 ₇		02 ₅	00 ₅	~ ₅	02 ₇	STP
37 ₇		03 ₅	00 ₅	~ ₅	02 ₇	WAI
37 ₇		1D ₅	1E ₅	~ ₅	02 ₇	RTD
37 ₇		1E ₅	1E ₅	~ ₅	02 ₇	RTE
37 ₇		1F ₅	1E ₅	~ 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	~ ₅	02 ₇	FENCE

Floating Point

~ ₅	Sz ₂	rm ₃	FPb ₅	FPT ₅	FPA ₅	28 ₇	FADD
~ ₅	Sz ₂	rm ₃	FPb ₅	FPT ₅	FPA ₅	29 ₇	FSUB
~ ₅	Sz ₂	rm ₃	FPb ₅	Rt₅	FPA ₅	2A ₇	FCMP
~ ₅	Sz ₂	rm ₃	FPb ₅	FPT ₅	FPA ₅	2B ₇	FMUL
~ ₅	Sz ₂	rm ₃	FPb ₅	FPT ₅	FPA ₅	2C ₇	FDIV
60 ₇	~ ₃	Sz ₂	rm ₃	FPT ₅	FPA ₅	02 ₇	FIX2FLT
61 ₇	~ ₃	Sz ₂	rm ₃	FPT ₅	FPA ₅	02 ₇	FLT2FIX
62 ₇	~ ₃	Sz ₂	~ ₃	FPT ₅	FPA ₅	02 ₇	FMOV
63 ₇	~ ₃	Sz ₂	~ ₃	FPT ₅	FPA ₅	02 ₇	FNEG
64 ₇	~ ₃	Sz ₂	~ ₃	FPT ₅	FPA ₅	02 ₇	FABS
65 ₇	~ ₃	~ ₂	~ ₃	Rt₅	FPA ₅	02 ₇	MFFP
66 ₇	~ ₃	Sz ₂	~ ₃	Rt₅	FPA ₅	02 ₇	MV2FIX
1C ₇	~ ₃	~ ₂	~ ₃	FPT ₅	Ra₅	02 ₇	MTFP
1D ₇	~ ₃	Sz ₂	rm ₃	FPT ₅	Ra₅	02 ₇	MV2FLT

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	~ 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 #

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}		~ ₂	0 ₃	Ra ₅	3D ₇	BEQ
Disp _{16..2}		~ ₂	1 ₃	Ra ₅	3D ₇	BNE
Disp _{16..2}		~ ₂	2 ₃	Ra ₅	3D ₇	BGT
Disp _{16..2}		~ ₂	3 ₃	Ra ₅	3D ₇	BGE / BPL
Disp _{16..2}		~ ₂	4 ₃	Ra ₅	3D ₇	BLT / BMI
Disp _{16..2}		~ ₂	5 ₃	Ra ₅	3D ₇	BLE
H	~ ₄	Vector ₉	~ ₅	1E ₅	38 ₇	BRK
Offset _{26..2}					39 ₇	BSR
Offset _{26..2}					3A ₇	BRA
Immed ₁₅			1F ₅	1E ₅	27 ₇	RTL
		Imm ₉			37 ₇	RTL
Immed ₁₅			1F ₅	1E ₅	3B ₇	RTS
		Imm ₉			30 ₇	RTS
Immed ₁₅			Rt ₅	Ra ₅	3C ₇	JAL
Immed ₁₅			Rt ₅	Ra ₅	3E ₇	JALI
~ ₂₅					3F ₇	NOP
0 ₉					36 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
Offs ₈	Sc ₂	Rb ₅	Imm ₅	Ra ₅	6F ₇	INCX

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#	LD#					
2x	MOV		ADDQ		MYST			RTL	FADD	FSUB	FCMP	FMUL	FDIV			
3x	RTS	PUSH/POP	BEQ	BNE	LDIQ	SYS	{PCtrl}	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		PUSHF	POPF		POP	LFSX	LFDX	LFQX		LWAR			
6x	SB	SC	SH	SW	INC	PEA	PUSH m	PUSH r	SBX	SCX	SHX	SWX	CAS	PEAX	SWCR	INCX
7x	SFS	SFD	SFQ		SFSX	SFDX	SFQX		IMM							

02 Group Func

[illegible]

		6 ₄	Op ₃	
Immediate ₉		6 ₄	0 ₃	RTS
Immediate ₉		6 ₄	7 ₃	RTL
0 ₄	Ra ₅	6 ₄	1 ₃	PUSH
1 ₄	FPa ₅	6 ₄	1 ₃	PUSHF
2 ₄	Rt ₅	6 ₄	1 ₃	POP
3 ₄	FPT ₅	6 ₄	1 ₃	POPF
Disp ₄	Ra ₅	6 ₄	2 ₃	BEQ
Disp ₄	Ra ₅	6 ₄	3 ₃	BNE
Imm ₄	Rt ₅	6 ₄	4 ₃	LDI
Vector ₉		6 ₄	5 ₃	SYS
0 ₉		6 ₄	6 ₃	NOP
1 ₉		6 ₄	6 ₃	STP
2 ₉		6 ₄	6 ₃	WAI
3 ₉		6 ₄	6 ₃	INT #3
4 ₉		6 ₄	6 ₃	CLI
5 ₉		6 ₄	6 ₃	SEI
6 ₉		6 ₄	6 ₃	RTI
7 ₉		6 ₄	6 ₃	RTE
8 ₉		6 ₄	6 ₃	RTD
Rt ₄	Ra ₅	4 ₄	0 ₂ R	MOV
Imm ₄	Rt ₅	4 ₄	1 ₃	ADDQ