The Free Instruction Set Architecture (FISA)

Torbjörn Granlund

2021-03-03

Chapter 1

Foreword

[This manual is work-in-progress. A newer version might be available for download here: `http://gmplib.org/~tege/fisa.pdf`]

FISA is a carefully designed instruction set architecture with superior performance integer computations in mind. The F in FISA stands for that it is Free, that it supports Four operands per instruction, and that it is suited for Fast computers. There are other Free ISAs out there, but none of them is as Fine as FISA.

Several aspects differentiate FISA from legacy ISAs:

- FISA is very carefully designed
- FISA has a balanced non-redundant instruction set
- FISA has clean instruction encoding
- FISA allows very simple implementations
- FISA allows very fast implementations
- FISA provides 64 general purpose registers and 63 floating-point registers
- FISA is a pure 64-bit architecture
- FISA has architectural support for call point register partitioning
- FISA allows for partial implementations, with mid-ware emulation layer
- FISA has up to 3 input operands and 1 result operand for both integer and floating-point operations.
- FISA is completely free

1.1 Notation

In the definition of instructions, the arithmetic is done accurately, with no implied truncation. The division operation "/" truncates towards zero. The assignment operator, $\leftarrow$, reduces the value $\bmod 2^{64}$. Comparisons yield 0 for false outcome, and -1, i.e., 64 ones, for true outcome.

Chapter 2

Instruction encoding overview

The instruction set assumes the following register usage:

Register	**Usage**	Relevant instructions
`gr(0)`	Return address pointer	`brl`
`gr(1)`	Global pointer	`gotoff`
`gr(2)`	Stack pointer	`spoff, ldm, stm, smask`
`gr(3)`	Frame pointer	`fpoff`

2.1 Arithmetic Instructions

op	rd	rc	rb	ra

`mulladd`	`rd,ra,rb,rc`	$gr(rd) \leftarrow gr(ra) \times gr(rb) + gr(rc)$
`mullsub`	`rd,ra,rb,rc`	$gr(rd) \leftarrow gr(ra) \times gr(rb) - gr(rc)$
`mulhadd`	`rd,ra,rb,rc`	$gr(rd) \leftarrow (gr(ra) \times gr(rb) + gr(rc)) \,/\, 2^{64}$ [unsigned]
`mulhsub`	`rd,ra,rb,rc`	$gr(rd) \leftarrow (gr(ra) \times gr(rb) - gr(rc)) \,/\, 2^{64}$ [unsigned]
`dsll`	`rd,rc,ra,rb`	$gr(rd) \leftarrow (gr(rc),,gr(ra)) \times 2^{gr(rb)} \,/\, 2^{64}$
`dsrl`	`rd,rc,ra,rb`	$gr(rd) \leftarrow (gr(rc),,gr(ra)) \,/\, 2^{gr(rb)}$
`addc`	`rd,ra,rb,rc`	$gr(rd) \leftarrow gr(rb) + gr(rc) + (gr(ra) \wedge 1)$
`subc`	`rd,ra,rb,rc`	$gr(rd) \leftarrow gr(rb) - gr(rc) - (gr(ra) \wedge 1)$
`cmpac`	`rd,ra,rb,rc`	$gr(rd) \leftarrow (gr(rb) + gr(rc) + (gr(ra) \wedge 1)) \geq 2^{64}$
`cmpsc`	`rd,ra,rb,rc`	$gr(rd) \leftarrow (gr(rb) - gr(rc) - (gr(ra) \wedge 1)) < 0$
`fmadd`	`rd,ra,rb,rc`	$fr(rd) \leftarrow fr(ra) \times fr(rb) + fr(rc)$
`fmnadd`	`rd,ra,rb,rc`	$fr(rd) \leftarrow -(fr(ra) \times fr(rb)) + fr(rc)$
`fmsub`	`rd,ra,rb,rc`	$fr(rd) \leftarrow fr(ra) \times fr(rb) + (-fr(rc))$
`fmnsub`	`rd,ra,rb,rc`	$fr(rd) \leftarrow -(fr(ra) \times fr(rb)) + (-fr(rc))$
`seleq`	`rd,ra,rb,rc,i1`	$gr(rd) \leftarrow$ if $gr(ra) = 0$ then $gr(rb)$ else $gr(rc)$
`sellt`	`rd,ra,rb,rc,i1`	$gr(rd) \leftarrow$ if $gr(ra) < 0$ then $gr(rb)$ else $gr(rc)$
`selle`	`rd,ra,rb,rc,i1`	$gr(rd) \leftarrow$ if $gr(ra) \leq 0$ then $gr(rb)$ else $gr(rc)$
`selev`	`rd,ra,rb,rc,i1`	$gr(rd) \leftarrow$ if $gr(ra) \bmod 2 = 0$ then $gr(rb)$ else $gr(rc)$
`mux`	`rd,ra,rb,rc`	$gr(rd) \leftarrow (gr(ra) \wedge gr(rb)) \vee (\neg gr(ra) \wedge gr(rc))$

The `sel` instructions' i1 argument select between dependency modes; if it is 1, the implementation must read both gr(rb) and gr(rc) before determining which register is to be written to the destination register. The result register is made a proper dependency on both gr(rb) and gr(rc). That means that they never leak timing information as a result of the condition.

op	rd	rc	rb	op

`add`	`rd,rb,rc`	gr(rd) $\leftarrow$ gr(rb) + gr(rc)
`sub`	`rd,rb,rc`	gr(rd) $\leftarrow$ gr(rb) $-$ gr(rc)
`sll`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rc) $<<$ gr(rb)
`srl`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rc) $>>$ gr(rb) [unsigned]
`sra`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rc) $>>$ gr(rb) [signed]
`cmpeq`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $=$ gr(rc)
`cmpne`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\neq$ gr(rc)
`cmplts`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $<$ gr(rc) [signed]
`cmpltu`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $<$ gr(rc) [unsigned]
`cmples`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\leq$ gr(rc) [signed]
`cmpleu`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\leq$ gr(rc) [unsigned]
`ncmpeq`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $=$ $-$gr(rc)
`ncmpne`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\neq$ $-$gr(rc)
`ncmplts`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $<$ $-$gr(rc) [signed]
`ncmpltu`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $<$ $-$gr(rc) [unsigned]
`ncmples`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\leq$ $-$gr(rc) [signed]
`ncmpleu`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\leq$ $-$gr(rc) [unsigned]
`and`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\wedge$ gr(rc)
`andn`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\wedge$ $\neg$gr(rc)
`or`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\vee$ gr(rc)
`orn`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\vee$ $\neg$gr(rc)
`xor`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) XOR gr(rc)
`xorn`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) XOR $\neg$gr(rc)
`nand`	`rd,rc,rb`	gr(rd) $\leftarrow$ $\neg$(gr(rb) $\wedge$ gr(rc))
`nor`	`rd,rc,rb`	gr(rd) $\leftarrow$ $\neg$(gr(rb) $\vee$ gr(rc))
`mull`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) $\times$ gr(rc)
`mulh`	`rd,rc,rb`	gr(rd) $\leftarrow$ (gr(rb) $\times$ gr(rc)) / 2^{64} [unsigned]
`divs`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) / gr(rc) [signed]
`divu`	`rd,rc,rb`	gr(rd) $\leftarrow$ gr(rb) / gr(rc) [unsigned]

op	rd	rc	simm (12)

`add`	`rd,simm,rc`	gr(rd) ← simm + gr(rc)
`sub`	`rd,simm,rc`	gr(rd) ← simm − gr(rc)
`sll`	`rd,rc,simm`	gr(rd) ← gr(rc) << simm
`srl`	`rd,rc,simm`	gr(rd) ← gr(rc) >> simm [unsigned]
`sra`	`rd,rc,simm`	gr(rd) ← gr(rc) >> simm [signed]
`cmpeq`	`rd,rc,simm`	gr(rd) ← simm = gr(rc)
`cmpne`	`rd,rc,simm`	gr(rd) ← simm ≠ gr(rc)
`cmplts`	`rd,rc,simm`	gr(rd) ← simm < gr(rc) [signed]
`cmpltu`	`rd,rc,simm`	gr(rd) ← simm < gr(rc) [unsigned]
`ncmpeq`	`rd,rc,simm`	gr(rd) ← simm = −gr(rc)
`ncmpne`	`rd,rc,simm`	gr(rd) ← simm ≠ −gr(rc)
`ncmplts`	`rd,rc,simm`	gr(rd) ← simm < −gr(rc) [signed]
`ncmpltu`	`rd,rc,simm`	gr(rd) ← simm < −gr(rc) [unsigned]
`and`	`rd,rc,simm`	gr(rd) ← simm ∧ gr(rc)
`andn`	`rd,rc,simm`	gr(rd) ← simm ∧ ¬gr(rc)
`or`	`rd,rc,simm`	gr(rd) ← simm ∨ gr(rc)
`orn`	`rd,rc,simm`	gr(rd) ← simm ∨ ¬gr(rc)
`xor`	`rd,rc,simm`	gr(rd) ← simm XOR gr(rc)
`xorn`	`rd,rc,simm`	gr(rd) ← simm XOR ¬gr(rc)
`nand`	`rd,rc,simm`	gr(rd) ← ¬(simm ∧ gr(rc))
`nor`	`rd,rc,simm`	gr(rd) ← ¬(simm ∨ gr(rc))
`mull`	`rd,rc,simm`	gr(rd) ← simm × gr(rc)
`mulh`	`rd,rc,simm`	gr(rd) ← (simm × gr(rc)) / 2^{64}

op	rd	uimm (18)

`mov`	`rd,imm`	gr(rd) ← uimm
`movn`	`rd,imm`	gr(rd) ← −uimm − 1
`mov8`	`rd,imm`	gr(rd) ← 8 × uimm
`movn8`	`rd,imm`	gr(rd) ← 8 × (−uimm − 1)
`gotoff`	`rd,imm`	gr(rd) ← gr(1) + 2^{12}× uimm
`spoff`	`rd,imm`	gr(rd) ← gr(2) + 2^{12}× uimm
`fpoff`	`rd,-imm`	gr(rd) ← gr(3) − 2^{12}× uimm

Note: The `mov8`, `movn8`, and `movn` instructions are not to be used directly. Assemblers should generate these when the operand range for `mov` is insufficient.

op	rd	rc	

`popcnt`	`rd,rc`	gr(rd) ← count_population(gr(rc))
`cnthz`	`rd,rc`	gr(rd) ← count_high_zeros(gr(rc))
`cntlz`	`rd,rc`	gr(rd) ← count_low_zeros(gr(rc))
`rcpr`	`rd,rc`	gr(rd) ← (2^{64} − 1)/gr(rc) − 1

2.2 Branch, Call, and Jump Instructions

op	simm (16)	cc	ra

`beq`	`ra,off`	if gr(ra) = 0 then pc ← pc + 4*simm
`blt`	`ra,off`	if gr(ra) < 0 then pc ← pc + 4*simm
`ble`	`ra,off`	if gr(ra) ≤ 0 then pc ← pc + 4*simm
`bev`	`ra,off`	if gr(ra) mod 2 = 0 then pc ← pc + 4*simm
`bne`	`ra,off`	if gr(ra) ≠ 0 then pc ← pc + 4*simm
`bge`	`ra,off`	if gr(ra) ≥ 0 then pc ← pc + 4*simm
`bgt`	`ra,off`	if gr(ra) > 0 then pc ← pc + 4*simm
`bod`	`ra,off`	if gr(ra) mod 2 = 1 then pc ← pc + 4*simm

The cc field chooses between the conditions EQ, LT, LE, EV. The main opcode chooses between plain branch and negated branch. Together, this allows for all 8 branch conditions.

The branch displacement range is -131072...131068.

op	simm (16)	simm (2)	rd

`ibnz`	`rd,off`	gr(rd) ← gr(rd) + C; if gr(rd) ≠ 0 then pc ← pc + 4*simm
`dbnz`	`rd,off`	gr(rd) ← gr(rd) − C; if gr(rd) ≠ 0 then pc ← pc + 4*simm

The value C is 1, 2, 4, or 8. It is calculated from the simm(2) field and the op field.

op	simm (24)

`br`	`off`	pc ← pc + 4*simm
`brl`	`off`	gr(0) ← pc + 8; pc ← pc + 4*simm

The branch displacement range is -33554432...33554428.

op		ra

`jmp`	`ra`	pc ← gr(ra)

op	rd		ra

`jmpl`	`rd, ra`	gr(rd) ← pc + 8; pc ← gr(ra)

2.3 Misc Instructions

op	rd	rc	

`copyfg`	`rd,rc`	gr(rd) ← fr(rc)
`copygf`	`rd,rc`	fr(rd) ← gr(rc)

2.4 Load and Store Instructions

op	rd	rc	rb	op

`ld8`	`[rc+rb],ra`	Load gr(ra) from mem(gr(rb)+gr(rc),8)
`ld16`	`[rc+rb],ra`	Load gr(ra) from mem(gr(rb)+gr(rc),16)
`ld32`	`[rc+rb],ra`	Load gr(ra) from mem(gr(rb)+gr(rc),32)
`ld64`	`[rc+rb],ra`	Load gr(ra) from mem(gr(rb)+gr(rc),64)
`ld16`	`[rc+rb*2],ra`	Load gr(ra) from mem(gr(rb)+2*gr(rc),16)
`ld32`	`[rc+rb*4],ra`	Load gr(ra) from mem(gr(rb)+4*gr(rc),32)
`ld64`	`[rc+rb*8],ra`	Load gr(ra) from mem(gr(rb)+8*gr(rc),64)

op	rd	rc	simm (12)

`ld8`	`rd,[rc+off]`	Load gr(rd) from mem(gr(rc)+simm,8)
`ld16`	`rd,[rc+off]`	Load gr(rd) from mem(gr(rc)+2*simm,16)
`ld32`	`rd,[rc+off]`	Load gr(rd) from mem(gr(rc)+4*simm,32)
`ld64`	`rd,[rc+off]`	Load gr(rd) from mem(gr(rc)+8*simm,64)

op	op	rc	rb	ra

`st8`	`[rc+rb],ra`	Store gr(ra) bits 7...0 to mem(gr(rb)+gr(rc),8)
`st16`	`[rc+rb],ra`	Store gr(ra) bits 16...0 to mem(gr(rb)+gr(rc),16)
`st32`	`[rc+rb],ra`	Store gr(ra) bits 31...0 to mem(gr(rb)+gr(rc),32)
`st64`	`[rc+rb],ra`	Store gr(ra) to mem(gr(rb)+gr(rc),64)
`st16`	`[rc+rb*2],ra`	Store gr(ra) bits 16...0 to mem(gr(rb)+2*gr(rc),16)
`st32`	`[rc+rb*4],ra`	Store gr(ra) bits 31...0 to mem(gr(rb)+4*gr(rc),32)
`st64`	`[rc+rb*8],ra`	Store gr(ra) to mem(gr(rb)+8*gr(rc),64)

op	simm (12)	rb	ra

`st8`	`[rb+off],ra`	Store gr(ra) bits 7...0 to mem(gr(rb)+simm,8)
`st16`	`[rb+off],ra`	Store gr(ra) bits 16...0 to mem(gr(rb)+2*simm,16)
`st32`	`[rb+off],ra`	Store gr(ra) bits 31...0 to mem(gr(rb)+4*simm,32)
`st64`	`[rb+off],ra`	Store gr(ra) to mem(gr(rb)+8*simm,64)

Since the offset is scaled differently depending on operand size, the offset range varies from -2048...2047 (for 8-bit operations) to -16384...16376 (for 64-bit operations).

2.5 Prefetch Instructions

op	op	rc	rb	ra

`pfr8`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),8) to mem(ra)
`pfr16`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),16) to mem(ra)
`pfr32`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),32) to mem(ra)
`pfr64`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),64) to mem(ra)
`pfrw8`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),8) to mem(ra)
`pfrw16`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),16) to mem(ra)
`pfrw32`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),32) to mem(ra)
`pfrw64`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),64) to mem(ra)
`pfw8`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),8) to mem(ra)
`pfw16`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),16) to mem(ra)
`pfw32`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),32) to mem(ra)
`pfw64`	`[rc+rb],ra`	Prefetch from mem(gr(rb)+gr(rc),64) to mem(ra)

op	rd	rc	simm (12)

`pfr8`	`[rc+off],ra`	Prefetch from mem(gr(rc)+simm,8) to mem(ra)
`pfr16`	`[rc+off],ra`	Prefetch from mem(gr(rc)+2*simm,16) to mem(ra)
`pfr32`	`[rc+off],ra`	Prefetch from mem(gr(rc)+4*simm,32) to mem(ra)
`pfr64`	`[rc+off],ra`	Prefetch from mem(gr(rc)+8*simm,64) to mem(ra)
`pfrw8`	`[rc+off],ra`	Prefetch from mem(gr(rc)+simm,8) to mem(ra)
`pfrw16`	`[rc+off],ra`	Prefetch from mem(gr(rc)+2*simm,16) to mem(ra)
`pfrw32`	`[rc+off],ra`	Prefetch from mem(gr(rc)+4*simm,32) to mem(ra)
`pfrw64`	`[rc+off],ra`	Prefetch from mem(gr(rc)+8*simm,64) to mem(ra)
`pfw8`	`[rc+off],ra`	Prefetch from mem(gr(rc)+simm,8) to mem(ra)
`pfw16`	`[rc+off],ra`	Prefetch from mem(gr(rc)+2*simm,16) to mem(ra)
`pfw32`	`[rc+off],ra`	Prefetch from mem(gr(rc)+4*simm,32) to mem(ra)
`pfw64`	`[rc+off],ra`	Prefetch from mem(gr(rc)+8*simm,64) to mem(ra)

Prefetch between the index expression and the address contained in register `ra`. Start at the first address and continue in the direction indicated by `ra`.

When using any of the `pfw` instructions, the contents of the entire area between the address expressions is declared as containing undefined data.

2.6 Handling of Unimplemented Instructions

Implementations of FISA are allowed to be partial, i.e., some instructions might be missing. Such instructions invoke MIS handlers. Before the handler is invoked, the hardware sets things up like this:

The trap goes to R[63] + 64 × op.

R[62] contains the address of the instruction following the trapped instruction.

R[61] is an alias of the destination register. Writes to R[61] goes to the destination register of the trapped instruction.

R[60] is a copy of R[ra].

R[59] is a copy of R[rb].

R[58] is a copy of R[rc].

R[57] contains the instruction word in its low 32 bits. This can be used for decoding the instruction.

Register R[56] through R[61] can be used as scratch in the handler.

2.7 Yet-to-be Documented Instructions

The following instructions have not yet been documented.

`ldm`	`mask`	
`stm`	`mask`	
`smask`	`mask`	
`scall`	`code`	
`ld32`	`rd,[rc+off]`	Load fr(rd) from mem(gr(rc)+4*simm,32)
`ld64`	`rd,[rc+off]`	Load fr(rd) from mem(gr(rc)+4*simm,64)
`st32`	`[rc+off],ra`	Store fr(ra) to mem(gr(rb)+4*simm,32)
`st64`	`[rc+off],ra`	Store fr(ra) to mem(gr(rb)+4*simm,64)
`cmpeq`	`rd,rc,rb`	gr(rd) $\leftarrow$ fr(rb) = fr(rc)
`cmpne`	`rd,rc,rb`	gr(rd) $\leftarrow$ fr(rb) $\neq$ fr(rc)
`cmplt`	`rd,rc,rb`	gr(rd) $\leftarrow$ fr(rb) $<$ fr(rc)
`cmple`	`rd,rc,rb`	gr(rd) $\leftarrow$ fr(rb) $\leq$ fr(rc)
`cmpgt`	`rd,rc,rb`	gr(rd) $\leftarrow$ fr(rb) $>$ fr(rc)
`cmpge`	`rd,rc,rb`	gr(rd) $\leftarrow$ fr(rb) $\geq$ fr(rc)
`cvtXX`	`rd,rc`	Convert between integer and floating-point
`implver`	`rd,rc`	

In addition to the above table, floating-point instructions and privileged instructions also need to be documented.