RISC-V ISA

Floating-point instructions

Sparsh Mittal

Courtesy

1. Prof Smruti Ranjan Sarangi
2. "The RISC-V Reader: An Open Architecture Atlas" by David Patterson and Andrew Waterman

Intro

- `F' and `D' extensions are meant for single precision and double precision floating point operations, respectively.
- Floating-point numbers are stored in the IEEE 754 format.

RV32F and RV32D

Floating-Point Computation

float { **add** / **sub**tract / **mul**tiply / **div**ide / **sq**uare **r**oo**t** / **min**imum / **max**imum } { **.s**ingle / **.d**ouble }

float { – / **n**egative } **m**ultiply { **add** / **sub**tract } { **.s**ingle / **.d**ouble }

float **m**o**v**e to **.s**ingle from .**x** register

float **m**o**v**e to .**x** register from **.s**ingle

Comparison

compare float { **eq**uals / **l**ess **t**han / **l**ess than or **e**quals } { **.s**ingle / **.d**ouble }

Load and Store

float { **l**oad / **s**tore } { **w**ord / **d**oubleword }

Conversion

float **con**ver**t** to { **.s**ingle / **.d**ouble } from **.w**ord { – / **u**nsigned }

float **con**ver**t** to **.w**ord { – / **u**nsigned } from { **.s**ingle / **.d**ouble }

float **con**ver**t** to **.s**ingle from **.d**ouble

float **con**ver**t** to **.d**ouble from **.s**ingle

Other instructions

float **sign** in**j**ection { – / **n**egative / e**x**clusive or } { **.s**ingle / **.d**ouble }

float **class**ify { **.s**ingle / **.d**ouble }

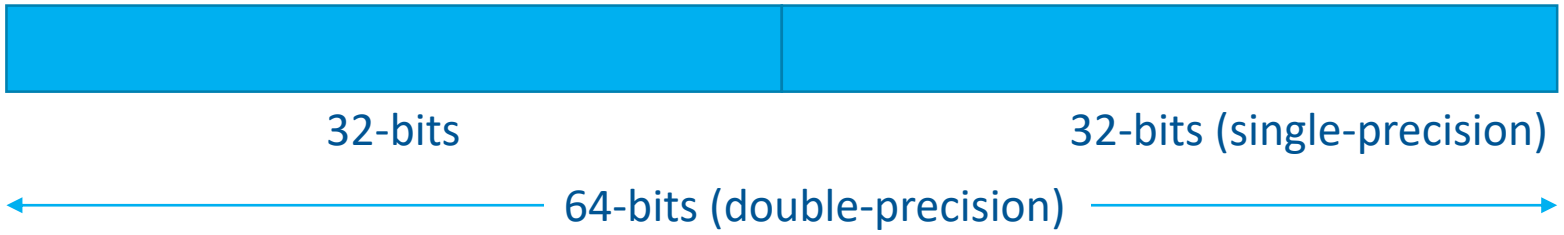
Registers

- There are 32 floating-point registers: f0 to f31. It is separate from integer registers x0 to x31.
- No register is hardwired to 0, but there is register usage convention.
- We cannot directly load an immediate into a floating point register.
- Like x86, floating point registers can only be initialized by loading values from memory.

Register	Mnemonic	Description
f0-7	ft0-7	Temporary registers
f8-9	fs0-1	Saved registers
f10-11	fa0-1	Arguments/return values
f12-17	fa2-7	Function arguments
f18-27	fs2-11	Saved registers
f28-31	ft8-11	Temporary registers

The main reason for the two sets of registers (integer and floating-point) is that processors can improve performance by doubling the register capacity and bandwidth by having two sets of registers without increasing the space for the register specifier in the cramped RISC-V instruction format.

If a processor has both RV32F and RV32D, the single-precision data uses only the lower 32 bits of the f registers



Load and Store Instructions

Single-precision load-store instructions

Semantics	Example	Explanation
flw rd, imm(rs1)	flw f1, 48(sp)	f1 ← mem[48 + sp]
fsw rs2, imm(rs1)	fsw f1, 48(sp)	mem[48 + sp] ← f1

The double-precision load is **fld** and store is **fsd**.
These instructions do not perform type conversion.

Load and store

- Floating point immediates cannot be directly loaded into registers
- Their contents need to be stored in memory first and then the 32-bit floating point value can be loaded into a floating point register.
- Most of the time, we need not load floating point immediates, other than while loading built-in constants such as π and e.
- In this case, we can use the assembler pseudoinstruction **la** to store the contents of the constant to memory and then load the address of the starting memory address to a register.
- Other than this, floating-point load/store are similar to integer counterparts.

Solved example

Load the value of a constant val into a floating point register fs1.

```
val: .float 3.14
.main :
la a1 , val
flw fs1 , 0( a1)
```

Floating-point arithmetic instructions

Semantics	Example	Explanation
fadd.s rd, rs1, rs2	fadd.s f1, f2, f3	f1 ← f2 + f3
fsub.s rd, rs1, rs2	fsub.s f1, f2, f3	f1 ← f2 - f3
fmul.s rd, rs1, rs2	fmul.s f1, f2, f3	f1 ← f2 × f3
fdiv.s rd, rs1, rs2	fdiv.s f1, f2, f3	f1 ← f2 ÷ f3
fmin.s rd, rs1, rs2	fmin.s f1, f2, f3	f1 ← min(f2 , f3)
fmax.s rd, rs1, rs2	fmax.s f1, f2, f3	f1 ← max(f2 , f3)
fsqrt.s rd, rs1	fsqrt.s f1, f2	f1 ← $\sqrt{f2}$

Suffix .s ➔ single-precision
Suffix .d ➔ double-precision
Note that we do not have variants that accept immediates directly as source operands.
The immediates can only be loaded using flw instructions or converted from integers

- In integer arithmetic, the size of the product is double of its operands.
- In floating-point arithmetic, the size of the product from a floating-point multiply is the same as its operands.
- There are also instructions for finding minimum and maximum (fmin.s, fmin.d, fmax.s, fmax.d), values from the pair of source operands without using a branch instruction.

Solved example

$$\sqrt{\pi + e + \pi \times e},$$

Compute
and store the
result in fa0.

```
# declare the constants
pi: . float 3.14
e: . float 2.72

.main :
# load them into floating point registers
la a1 , pi
flw fs1 , 0( a1)
la a2 , e
flw fs2 , 0( a2)

fadd.s ft1, fs1 , fs2  # pi + e
fmul.s ft2, fs1 , fs2  # pi * e
fadd.s ft3, ft1 , ft2   # pi + e + pi * e
fsqrt.s fa0, ft3        # sqrt (pi + e + pi*e)
```

Fused addition and subtraction instructions

Semantics	Example	Explanation
fmadd.s rd, rs1, rs2, rs3	fmadd.s f1, f2, f3, f4	f1 ← f2 * f3 + f4
fmsub.s rd, rs1, rs2, rs3	fmsub.s f1, f2, f3, f4	f1 ← f2 * f3 - f4

- To support operations such as dot products and matrix multiplication, RISC-V supports a few fused arithmetic instructions such as the fused addition and subtraction operations.
- These fused multiply-add instructions are more accurate as well as faster than separate multiply and add instructions, because they round only once (after the add) rather than twice (after the multiply, then after the add).

Floating-point comparison instructions

Comparing FP numbers is not the same as comparing integers.
They cannot be directly given as arguments to conditional branch instructions.
In this case, the status of the comparison needs to be stored in an integer register.
This register can then be compared with the zero register using a regular conditional branch instruction.

These three FP comparison instructions store the result in an integer register.

Semantics	Example	Explanation
flt.s rd, rs1, rs2	flt.s s1, f2, f3	if (f2 $<$ f3) set s1 to 1
fle.s rd, rs1, rs2	fle.s s1, f2, f3	if (f2 $\leq$ f3) set s1 to 1
feq.s rd, rs1, rs2	feq.s s1, f2, f3	if (f2 == f3) set s1 to 1

Solved example

First, initialize a0 = 0, then set a0 = 17 if e < π .

```
pi: . float 3.14
e: . float 2.72

.main :
la a1 , pi       # load pi
flw fs1 , 0( a1)
la a2 , e        # load e
flw fs2 , 0( a2)

add a0 , zero , zero          # a0 = 0
flt.s t0, fs2 , fs1           # compare pi and e
beq t0 , zero , .end          # if (t0 == 0) jump to .end

addi a0 , zero , 17           # a0 = 17 because t0 == 1
.end:
```

FP-to-Integer and Integer-to-FP Conversion Instructions

RV32F and RV32D have instructions that perform all combinations of useful conversions between 32-bit signed integers, 32-bit unsigned integers, 32-bit floating point, and 64-bit floating point.

To	From			
	32b signed integer (`w`)	32b unsigned integer (`wu`)	32b floating point (`s`)	64b floating point (`d`)
32b signed integer (`w`)	–	–	`fcvt.w.s`	`fcvt.w.d`
32b unsigned integer (`wu`)	–	–	`fcvt.wu.s`	`fcvt.wu.d`
32b floating point (`s`)	`fcvt.s.w`	`fcvt.s.wu`	–	`fcvt.s.d`
64b floating point (`d`)	`fcvt.d.w`	`fcvt.d.wu`	`fcvt.d.s`	–

Semantics	Example	Explanation
fcvt.s.w rd, rs1	fcvt.s.w f1, x5	f1 ← (float) x5
fcvt.w.s rd, rs1	fcvt.w.s x5, f1	x5 ← (int) f1

Solved example

Compute π*e + 4, and store the result in fa0. Convert the result to an integer and store the result in a0

```
pi: . float 3.14
e: . float 2.72

.main :
la a1 , pi                  # load pi
flw fs1 , 0( a1)
la a2 , e                   # load e
flw fs2 , 0( a2)

Li t1, 4                    # load 4.0 in a register
fcvt.s.w ft1 , t1

fmadd.s ft0 , fs1 , fs2 , ft1 # pi * e + 4
fcvt.w.s a0 , ft0                # convert to int
```

Comparison of ISAs for DAXPY

```
void daxpy(size_t n, double a, const double x[], double y[])
{
  for (size_t i = 0; i < n; i++) {
    y[i] = a*x[i] + y[i];
  }
}
```

C code of Y = aX+Y (a is scalar, X and Y are vectors)

Number of instructions and code size of DAXPY for four ISAs. RISC-V version again has about the same or fewer instructions

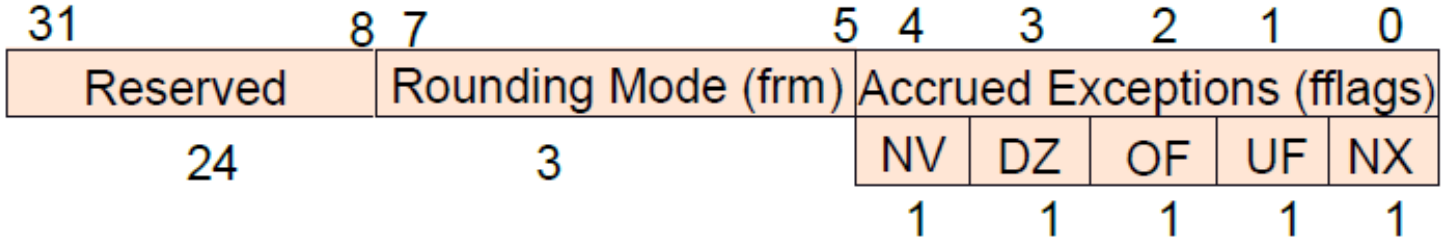
ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32FD	RV32FD+RV32C
Instructions	10	10	12	12	16	11	11
Per Loop	6	6	7	7	6	7	7
Bytes	40	28	48	32	50	44	28

RV32D code for DAXPY

```
# RV32FD (7 insns in loop; 11 insns/44 bytes total; 28 bytes RVC)
# a0 is n, a1 is pointer to x[0], a2 is pointer to y[0], fa0 is a
  0: 02050463 beqz    a0,28           # if n == 0, jump to Exit
  4: 00351513 slli    a0,a0,0x3       # a0 = n*8
  8: 00a60533 add     a0,a2,a0        # a0 = address of x[n] (last element)
Loop:
  c: 0005b787 fld     fa5,0(a1)       # fa5 = x[]
 10: 00063707 fld     fa4,0(a2)       # fa4 = y[]
 14: 00860613 addi    a2,a2,8         # a2++ (increment pointer to y)
 18: 00858593 addi    a1,a1,8         # a1++ (increment pointer to x)
 1c: 72a7f7c3 fmadd.d fa5,fa5,fa0,fa4 # fa5 = a*x[i] + y[i]
 20: fef63c27 fsd     fa5,-8(a2)      # y[i] = a*x[i] + y[i]
 24: fea614e3 bne     a2,a0,c         # if i != n, jump to Loop
Exit:
 28: 00008067         ret             # return
```

Floating point control and status register (fcsr),

There is an additional special register called fcsr



Its lower 8 bits encode important information.

The lower 5 bits store exceptional conditions encountered since these bits were last reset. This is known as the fflags field.

The rest of the 3 bits store the rounding mode.

Accrued Exception Flags (fflags)

Fflags stores five flags, which are also known as the accrued exception flags.
Inexact flag is set when the result cannot exactly be stored in a floating point register and some rounding was required.

Mnemonic	Explanation
NV	Invalid operation
DZ	Divide by zero
OF	Overflow
UF	Underflow
NX	Inexact

Rounding modes

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to nearest, prefer even LSBs
001	RTZ	Round towards zero.
010	RDN	Round down (towards -∞). Floor function
011	RUP	Round up (towards +∞). Ceiling function
100	RMM	Round to nearest, prefer the number with the maximum magnitude
101		Invalid. Reserved for future use.
110		Invalid. Reserved for future use.
111	DYN	Selects a rounding mode dynamically (stored in the frm field of the $fcsr$)

Rounding modes

- RISC-V instructions can use a static rounding mode (encoded in the instruction) or a dynamic rounding mode (encoded in the fcsr's frm field).
- The default rounding mode is RNE.
- We round the result to the nearest value that can be represented in the IEEE 754 format.
- If the real value is between two representable values, then the result is rounded to the value that has an even LSB.

Rounding modes

RTZ (round to zero): it is same as truncation; the bits that cannot be fit in the format are simply removed.

RMM rounding mode is similar to RNE. However, if the result is between two representable values, then we round towards the number that has the higher magnitude (away from zero).

RV32F opcode map

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	010	rd	0000111	I flw
imm[11:5]		rs2	rs1	010	imm[4:0]	0100111	S fsw
rs3	00	rs2	rs1	rm	rd	1000011	R4 fmadd.s
rs3	00	rs2	rs1	rm	rd	1000111	R4 fmsub.s
rs3	00	rs2	rs1	rm	rd	1001011	R4 fnmsub.s
rs3	00	rs2	rs1	rm	rd	1001111	R4 fnmadd.s
0000000		rs2	rs1	rm	rd	1010011	R fadd.s
0000100		rs2	rs1	rm	rd	1010011	R fsub.s
0001000		rs2	rs1	rm	rd	1010011	R fmul.s
0001100		rs2	rs1	rm	rd	1010011	R fdiv.s
0101100		00000	rs1	rm	rd	1010011	R fsqrt.s
0010000		rs2	rs1	000	rd	1010011	R fsgnj.s
0010000		rs2	rs1	001	rd	1010011	R fsgnjn.s
0010000		rs2	rs1	010	rd	1010011	R fsgnjx.s
0010100		rs2	rs1	000	rd	1010011	R fmin.s
0010100		rs2	rs1	001	rd	1010011	R fmax.s
1100000		00000	rs1	rm	rd	1010011	R fcvt.w.s
1100000		00001	rs1	rm	rd	1010011	R fcvt.wu.s
1110000		00000	rs1	000	rd	1010011	R fmv.x.w
1010000		rs2	rs1	010	rd	1010011	R feq.s
1010000		rs2	rs1	001	rd	1010011	R flt.s
1010000		rs2	rs1	000	rd	1010011	R fle.s
1110000		00000	rs1	001	rd	1010011	R fclass.s
1101000		00000	rs1	rm	rd	1010011	R fcvt.s.w
1101000		00001	rs1	rm	rd	1010011	R fcvt.s.wu
1111000		00000	rs1	000	rd	1010011	R fmv.w.x

RV32D opcode map

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
imm[11:0]			rs1	011	rd	0000111	I fld
imm[11:5]		rs2	rs1	011	imm[4:0]	0100111	S fsd
rs3	01	rs2	rs1	rm	rd	1000011	R4 fmadd.d
rs3	01	rs2	rs1	rm	rd	1000111	R4 fmsub.d
rs3	01	rs2	rs1	rm	rd	1001011	R4 fnmsub.d
rs3	01	rs2	rs1	rm	rd	1001111	R4 fnmadd.d
0000001		rs2	rs1	rm	rd	1010011	R fadd.d
0000101		rs2	rs1	rm	rd	1010011	R fsub.d
0001001		rs2	rs1	rm	rd	1010011	R fmul.d
0001101		rs2	rs1	rm	rd	1010011	R fdiv.d
0101101		00000	rs1	rm	rd	1010011	R fsqrt.d
0010001		rs2	rs1	000	rd	1010011	R fsgnj.d
0010001		rs2	rs1	001	rd	1010011	R fsgnjn.d
0010001		rs2	rs1	010	rd	1010011	R fsgnjx.d
0010101		rs2	rs1	000	rd	1010011	R fmin.d
0010101		rs2	rs1	001	rd	1010011	R fmax.d
0100000		00001	rs1	rm	rd	1010011	R fcvt.s.d
0100001		00000	rs1	rm	rd	1010011	R fcvt.d.s
1010001		rs2	rs1	010	rd	1010011	R feq.d
1010001		rs2	rs1	001	rd	1010011	R flt.d
1010001		rs2	rs1	000	rd	1010011	R fle.d
1110001		00000	rs1	001	rd	1010011	R fclass.d
1100001		00000	rs1	rm	rd	1010011	R fcvt.w.d
1100001		00001	rs1	rm	rd	1010011	R fcvt.wu.d
1101001		00000	rs1	rm	rd	1010011	R fcvt.d.w
1101001		00001	rs1	rm	rd	1010011	R fcvt.d.wu