Floating-point support in RISC-V

M. Sonza Reorda, Luca Sterpone,
M. Rebaudengo

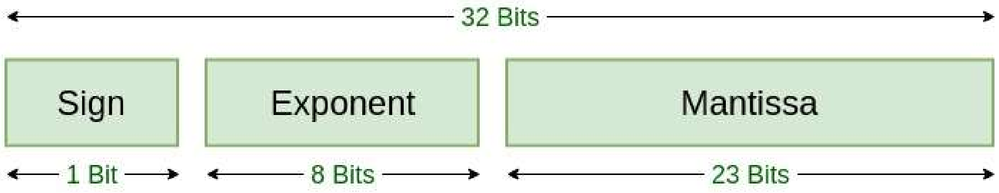
Politecnico di Torino
Dipartimento di Automatica e Informatica

RISC-V Floating-Point Extensions

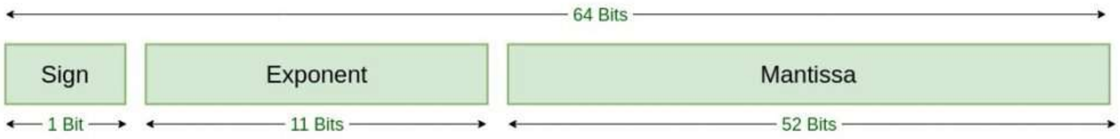
- RISC-V offers three floating-point extensions:
  - **RVF:** single-precision (32-bit)
    - 8 exponent bits, 23 fraction bits
  - **RVD:** double-precision (64-bit)
    - 11 exponent bits, 52 fraction bits
  - **RVQ:** quad-precision (128-bit)
    - 15 exponent bits, 112 fraction bits

Floating-point formats

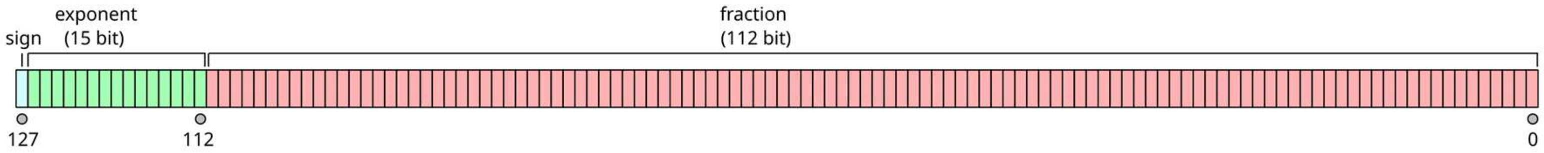
- According to the IEEE 754 standard, floating-point numbers can be represented in 3 formats
- Single precision (32 bits)



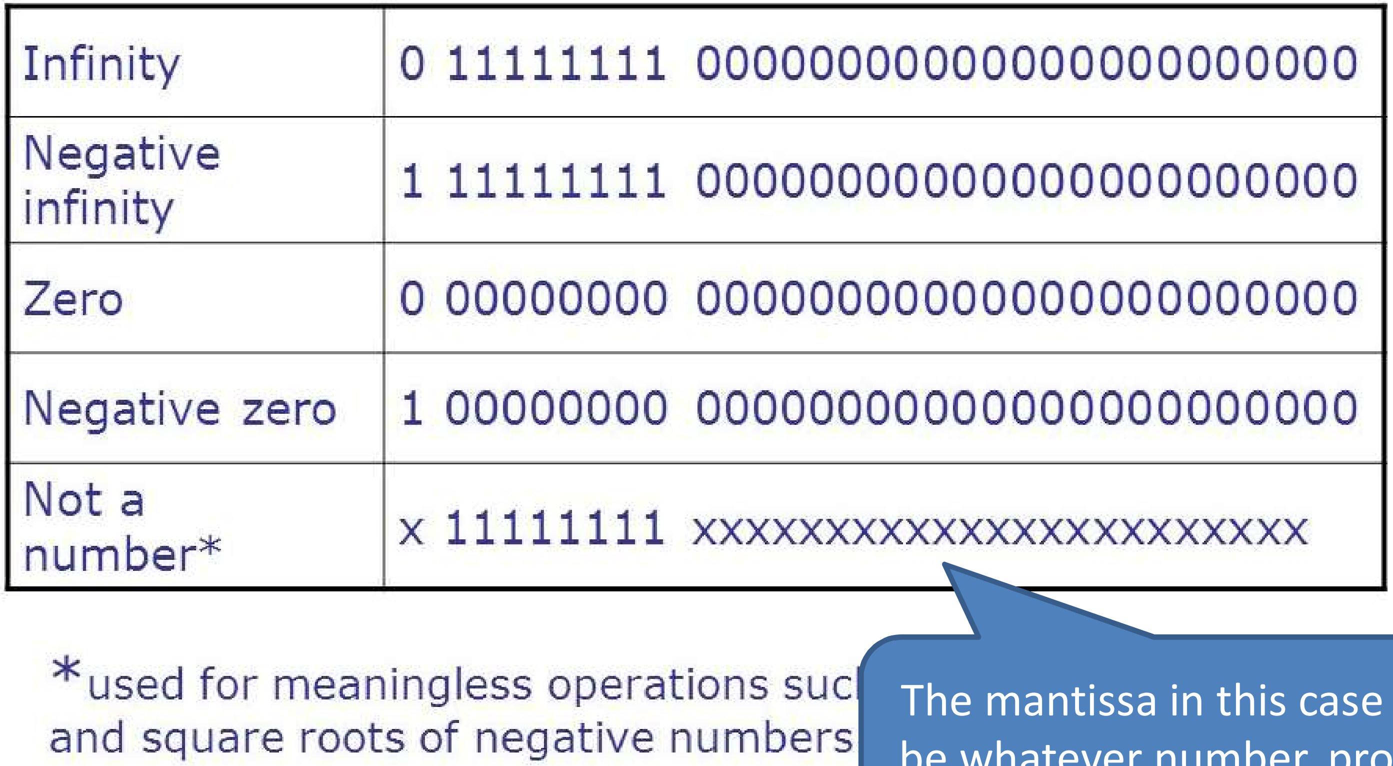
- Double precision (64 bits)



- Quadruple precision (128 bits)



Special cases



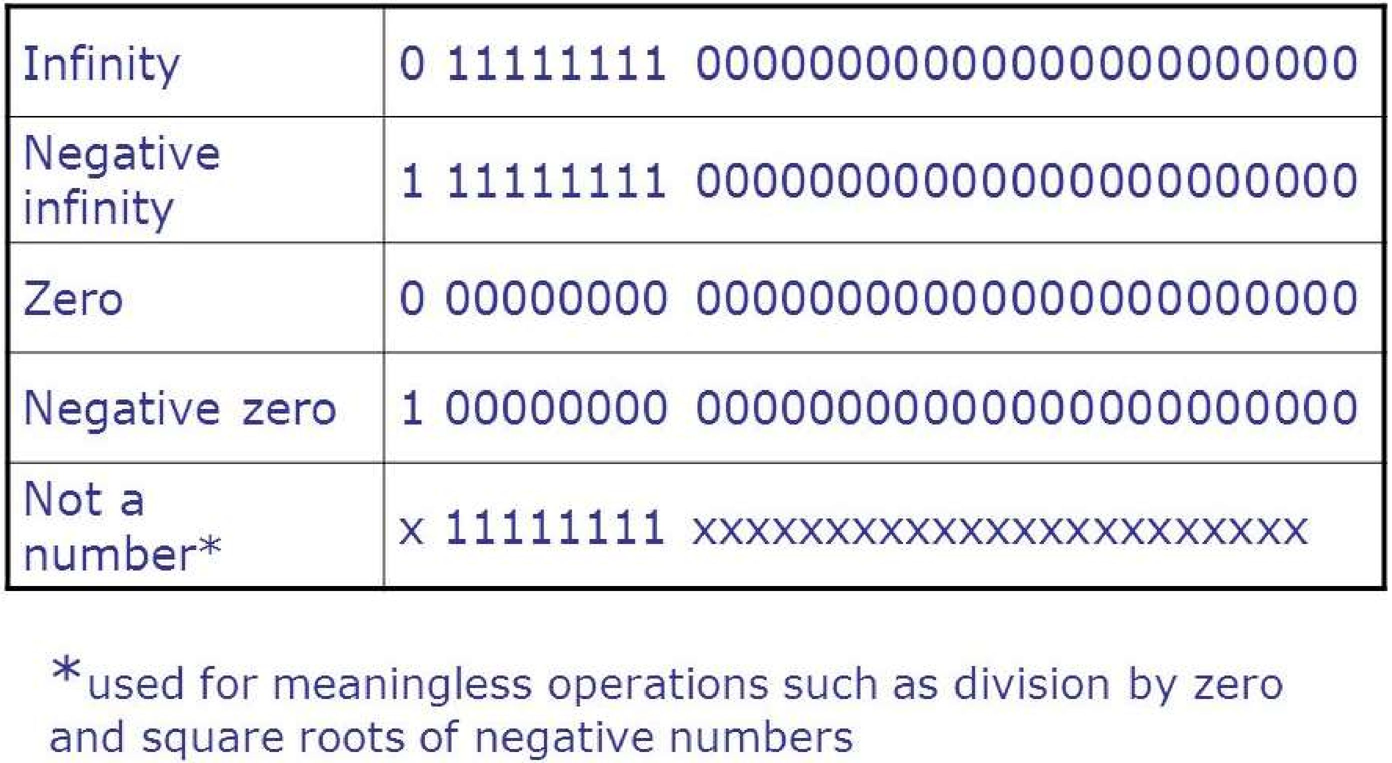
Infinity	0 11111111 00000000000000000000000
Negative infinity	1 11111111 00000000000000000000000
Zero	0 00000000 00000000000000000000000
Negative zero	1 00000000 00000000000000000000000
Not a number*	x 11111111 xxxxxxxxxxxxxxxxxxxxxxx

*used for meaningless operations suc
and square roots of negative numbers

The mantissa in this case should be whatever number, providedit is different than all 0s

- The standard also allows fo
subnormal) representations. In such a case
  - The exponent is the minimum one
  - The mantissa MSB is a 0 (instead of an implicit 1).

Special cases



Infinity	0 11111111 00000000000000000000000
Negative infinity	1 11111111 00000000000000000000000
Zero	0 00000000 00000000000000000000000
Negative zero	1 00000000 00000000000000000000000
Not a number*	x 11111111 xxxxxxxxxxxxxxxxxxxxxxx

*used for meaningless operations such as division by zero and square roots of negative numbers

- The standard also allows for denormalized (or *subnormal*) representations. In such a case
  - The exponent is the minimum one
  - The mantissa MSB is a 0 (instead of an implicit 1).

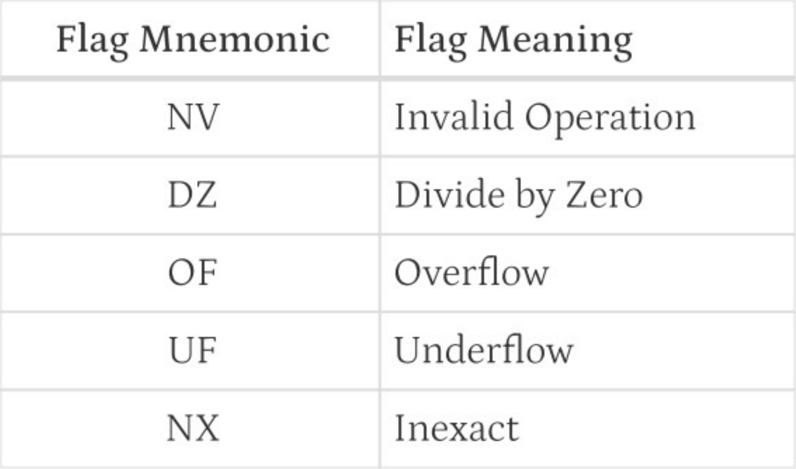
NaN

- A NaN (Not-a-Number) is a symbolic entity encoded in floating-point format. There are 2 types of NaN:
  - *Signalling NaN*: it signals an invalid operation exception if used in an arithmetic operation
  - *Quiet NaN*: it propagates through almost every arithmetic operation without signalling an exception.

- A number is considered NaN if the exponent is all 1s:
  - Quiet NaN: the MSB of the mantissa is 1
  - Signalling NaN: the MSB of the mantissa is 0.

Exceptions and FP instructions

- Floating-point instructions never trigger an exception due to their result
- Rather, they may set one of the following flags, that can be read from the Floating-Point Control and Status Register (`fcsr`)
- `fcsr` is a CSR that can be accessed via special instructions (e.g., FRCSR reads `fcsr` by copying it into an integer register rd).



Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

Floating-Point Registers

- **32** Floating-point registers
- **Width** is highest precision – for example, if RVQ is implemented, registers are 128 bits wide
- When multiple floating-point extensions are implemented, the lower-precision values occupy the lower bits of the register

Floating-Point Registers

Name	Register Number	Usage
`ft0-7`	f0-7	Temporary variables
`fs0-1`	f8-9	Saved variables
`fa0-1`	f10-11	Function arguments/Return values
`fa2-7`	f12-17	Function arguments
`fs2-11`	f18-27	Saved variables
`ft8-11`	f28-31	Temporary variables

Floating-Point Instructions

- Append .s (single), .d (double), .q (quad) for precision:
  - i.e., `fadd.s`, `fadd.d`, and `fadd.q`
- **Arithmetic operations**:

  `fadd`, `fsub`, `fdiv`, `fsqrt`, `fmin`, `fmax`, `multiply-add` (`fmadd`, `fmsub`, `fnmadd`, `fnmsub`)
- **Other instructions:**

  move (`fmv.x.w`, `fmv.w.x`)
  convert (`fcvt.w.s`, `fcvt.s.w`, etc.)
  comparison (`feq`, `flt`, `fle`)
  classify (`fclass`)
  sign injection (`fsgnj`, `fsgnjn`, `fsgnjx`)

Floating-Point Instructions

- FADD.S (Floating-point Add Single-Precision): Adds two single-precision (32-bit) floating-point numbers and stores the result in a floating-point register
- FSUB.S (Floating-point Sub Single-Precision): Subtracts two single-precision (32-bit) floating-point numbers and stores the result in a floating-point register
- FMUL.S (Floating-point Multiply Single-Precision): Multiplies two single-precision (32-bit) floating-point numbers and stores the result in a floating-point register
- FDIV.S (Floating-point Divide Single-Precision): Divides two single-precision (32-bit) floating-point numbers and stores the result in a floating-point register.

Floating-Point Instructions

- FMIN.S (Floating-point Min Single-Precision)
  - `fmin.s rd,rs1,rs2`
  - writes into rd the smaller out of the two single precision data in rs1 and rs2
- FMAX.S (Floating-point Max Single-Precision)
  - `fmax.s rd,rs1,rs2`
  - writes into rd the larger out of the two single precision data in rs1 and rs2
- FSQRT.S (Floating-point Square Root Single-Precision)
  - `fsqrt.s rd,rs1`
  - performs single-precision square root, `f[rd] = sqrt(f[rs1])`

Floating-Point Fused Multiply-Add

- FMADD multiplies the values in rs1 and rs2, adds the value in rs3, and writes the final result to rd
- it requires four register operands.
  - `fmadd.s f1, f2, f3, f4    # f1 = f2 x f3 + f4`

- Furthermore
  - `fmsub.s f1, f2, f3, f4         # f1 = f2 x f3 - f4`
  - `fnmadd.s f1, f2, f3, f4        # f1 = - f2 x f3 + f4`
  - `fnmsub.s f1, f2, f3, f4        # f1 = - f2 x f3 - f4`

Floating-Point Instructions: Load and Store

- FLW (Floating-point Load Word): loads a 32-bit word from memory into the specified floating-point register
  - `flw rd,offset(rs1)`
- FSW (Floating-point Store Word): stores the value of the specified floating-point register as a 32-bit word in memory
  - `fsw rs2,offset(rs1)`

Floating-Point Instructions: Convert

- FCVT.W.S (Floating-point Convert to Signed Word Single-Precision): converts a single-precision (32-bit) floating-point number to a 32-bit signed integer and stores the result in an integer register
- FCVT.S.W (Floating-point Convert from Signed Word Single-Precision): converts a 32-bit signed integer to a single-precision (32-bit) floating-point number in a floating point register
- Furthermore, unsigned versions:
  - FCVT.WU.S (Floating-point Convert to Unsigned Word Single-Precision): converts a single-precision (32-bit) floating-point number to a 32-bit unsigned integer and stores the result in an integer register
  - FCVT.S.WU (Floating-point Convert from Unsigned Word Single-Precision): converts a 32-bit unsigned integer to a single-precision (32-bit) floating-point number in a floating point register

Floating-Point Instructions: Convert

- FCVT.W.S (Floating-point Convert to Signed Word Single-Precision): converts a single-precision (32-bit) floating-point number to a 32-bit signed integer and stores the result in an integer register
- FCVT.S.W (Floating-point Convert from Signed Word Single-Precision): converts a 32-bit signed integer to a single-precision (32-bit) floating-point number in a floating-point register
- Furthermore, unsigned versions:
  - FCVT.WU.S (Floating-point Conver Precision): converts a single-pre number to a 32-bit unsigned integ integer register
  - FCVT.S.WU (Floating-point Convert Precision): converts a 32-bit unsign (32-bit) floating-point number in a

When performing conversion, the 3-bit `rm` field in the instruction encoding defines the rounding mode (e.g., round to nearest, round towards zero, round down, round up, etc.)

Floating-Point Instructions: Sign Injection

- FSGNJ.S (Floating-point Sign-Injection Single-Precision): performs a sign injection on a single-precision (32-bit) floating-point number and stores the result in a floating-point register
  - `fsgnj.s rd,rs1,rs2          # rd = abs(rs1) * sgn(rs2)`

- Furthermore:
  - `fsgnjn.s rd,rs1,rs2         # rd = abs(rs1) * -sgn(rs2)`
  - `fsgnjx.s rd,rs1,rs2         # rd = rs1 * -sgn(rs2)`

Floating-Point Instructions: Sign Injection

- FSGNJ.S (Floating-point Sign-Injection Single-Precision): performs a sign injection on a single-precision (32-bit) floating-point number and stores the result in a floating-point register
  - `fsgnj.s rd,rs1,rs2          # rd = abs(rs1) * sgn(rs2)`

- Furthermore:
  - `fsgnjn.s rd,rs1,rs2         # rd = abs(rs1) * -sgn(rs2)`
  - `fsgnjx.s rd,rs1,rs2         # rd = rs1 * -sgn(rs2)`

In practice

- for FSGNJ, the result's sign bit is rs2's sign bit;
- for FSGNJN, the result's sign bit is the opposite of rs2's sign bit;
- for FSGNJX, the sign bit is the XOR of the sign bits of rs1 and rs2.

Floating-Point Instructions: Move

- Instructions are provided to move bit patterns between the floating-point and integer registers
- The bits are not modified in the transfer
- `fmv.w.x fd, rs`
  - move the single-precision value encoded in IEEE 754-2008 standard encoding from the integer register rs to the floating-point register fd
- `fmv.x.w rd, fs`
  - move the single-precision value in floating-point register fs represented in IEEE 754-2008 encoding to the integer register rd.

Floating-Point Instructions: compare

- Floating-point compare instructions perform the specified comparison (equal, less than, less than or equal) between floating-point registers rs1 and rs2
- They record the Boolean result in integer register rd (writing 1 if the condition holds, and 0 otherwise)
  - `feq.s rd,rs1,rs2`
  - `flt.s rd,rs1,rs2`
  - `fle.s rd,rs1,rs2`

Floating-Point Instructions: classify

- `fclass.s rd,rs1`
- It examines the value in floating-point register rs1 and writes to integer register rd a 10-bit mask that indicates the class of the floating-point number. The corresponding bit in rd will be set if the property is true and clear otherwise.

rd bit	Meaning
0	rs1 is -infinity
1	rs1 is a negative normal number
2	rs1 is a negative subnormal number
3	rs1 is −0
4	rs1 is +0
5	rs1 is a positive subnormal number
6	rs1 is a positive normal number
7	rs1 is +infinity
8	rs1 is a signaling NaN
9	rs1 is a quiet NaN

Floating-Point Example

C Code

```
int i;
float scores[200];

for (i=0; i<200; i=i+1)

  scores[i]=scores[i]+10;
```

RISC-V assembly code

```
# s0 = scores base address, s1 = i
  addi s1, zero, 0          # i = 0
  addi t2, zero, 200        # t2 = 200
  addi t0, zero, 10         # ft0 = 10.0
  fcvt.s.w ft0,  t0
for:
  bge    s1, t2, done       # i>=200? done
  slli   t0, s1, 2          # t0 = i*4
  add    t0, t0, s0         # scores[i] address
  flw    ft1, 0(t0)         # ft1=scores[i]
  fadd.s ft1, ft1, ft0      # ft1=scores[i]+10
  fsw    ft1, 0(t0)         # scores[i] = t1
  addi   s1, s1, 1          # i = i+1
  j      for                # repeat
done:
```

Floating-Point Instruction Formats

- Use R-, I-, and S-type formats
- Introduce another format for multiply-add instructions that have 4 register operands: R4-type

R4-Type

31:27	26:25	24:20	19:15	14:12	11:7	6:0
rs3	funct2	rs2	rs1	funct3	rd	op
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits