Floating-Point Numbers

Z. Jerry Shi
Department of Computer Science and Engineering
University of Connecticut

CSE3666: Introduction to Computer Architecture

Outline

- Real numbers in binary
  - Decimal to binary
  - Binary to decimal
- IEEE 754 floating-point number standards
  - Single precision and double precision
- RISC-V support for floating-point numbers

Reading: Section 3.5, excluding hardware support for floating-point numbers.

0.00···001 0.9999···9

$-\infty$ 0 1 $+\infty$

Real numbers

- Computers need to deal with
  - Numbers with fractions (not just whole numbers)
  - Very big numbers
  - Very small numbers

Example of real numbers in decimal:

? $3.14159\ldots$

? -0.002×10^{-20} ← not normalized

? 9.4607×10^{15} (meters in a light year)

Normalized scientific notation: Only one non-zero digit to the left of the decimal point.

Binary number with fraction

exact match

- To represent fractions in binary, we use bits after the binary point

What is the value of the following binary number?

?

`101.1101`

Binary point

radix ?
(base) ?

$2^2 \; 2^1 \; 2^0 \, . \, 2^{-1} \; 2^{-2} \; 2^{-3}$

Binary to decimal

Example: `0b101.1101` $_{(2)} = 5.8125_{(10)}$

bits	1	0	1	1	1	0	1
weights	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

(base radix)

Multiply each bit with weight:

Integer part

Fractional part

0b101.1101

$$= 1 \times \overset{4}{2^2} + 0 \times \overset{2}{2^1} + 1 \times 2^0$$

$$+ 1 \times \underset{0.5}{2^{-1}} + 1 \times \underset{0.25}{2^{-2}} + 0 \times \underset{0.125}{2^{-3}} + 1 \times \underset{0.0625}{2^{-4}}$$

$$= 4 + 0 + 1 + 0.5 + 0.25 + 0 + 0.0625$$

$$= 5.8125$$

[Dec]

Decimal to binary

Example:

Convert the decimal number 0.8 to a binary number

0						
2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}

Converting decimal to binary

$0.8 = 0.11001100\cdots ?$

Decimal	Binary
0.8	0.
0.8 * 2 = 1.6	0.1
0.6 * 2 = 1.2	0.11
0.2 * 2 = 0.4	0.110
0.4 * 2 = 0.8	0.1100
0.8 * 2 = 1.6	0.11001…
Continue….	0.1100110011001100110 ….

Fraction .8 appears again. The pattern 1100 will repeat forever.

```
python
>>> float.hex(0.8)
'0x1.999999999999ap-1'
```

$1.1001\,1001\,1001 - \cdots \times 2^{-1}$ bin

$1.999 \cdots \times 2^{-1}$ hex

Normalized notation of binary numbers

- There are many representations as we move the binary point

$$101.1101 = 10.11101 \times 2^1 = \boxed{1.011101 \times 2^2} = 0.1011101 \times 2^3$$

Normalized binary representation

The normalized binary representation has a single 1 before the point

$$\underbrace{\pm 1.x}_{\text{Significand}} \times 2^{E}$$

Exponent is written in decimal for convenience

```
python
>>> float.hex(float.fromhex('5.d'))
'0x1.7400000000000p+2'
```

Encode floating-point numbers

- Given a number of bits, how do we represent

$$\pm 1.x \times 2^E$$

Design Problem ?

- What need to be encoded? ±, x, E
- How many bits for each?

1 bit + 29 + 2 = 32-64

28 3

26 5

23 8

FP

IEEE FP

FB (Meta) FP8 ML

$-\infty$ real numbers $+\infty$

Floating Point Standard (single and double precisions)

- Defined by IEEE Std 754-1985
  - Developed in response to divergence of representations
  - Solve the portability issues for scientific code
  - Now almost universally adopted

10^{100} ?

3.4×10^{38} (Nor largest IEEE FP)

[approximation]

Nor smallest positive 1.2×10^{-38}

$\times10^{-100}$?

64-bit

$-\infty$ IEEE Std FP $+\infty$

0 1 2

Denor FP

- Single precision (32-bit) and double precision (64-bit)
  - Double have more bits to represent exponent and fraction
  - They are types float and double in C

- Later versions of the standard include more types
  - E.g., 128-bit quad-precision

IEEE Floating-Point Format: single-precision

	8 bits	23 bits
S	Exponent	Fraction

Sign
0 : non-negative,
1: negative

Exponent in
excess-k representation
or biased representation

Bits after the binary point
There is a hidden 1 before the point!

$$\text{value} = (-1)^{S} \times (1.\text{Fraction}) \times 2^{E}$$

Exponent is in excess-127 representation. The Bias = 127.

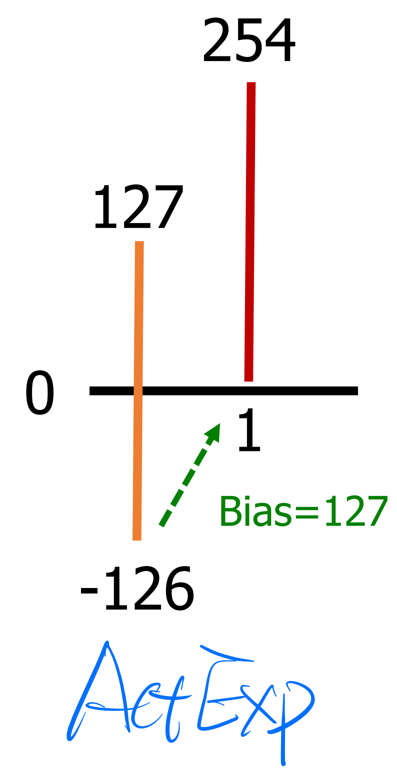
? ① ActEx = EncEx − 127

? ② EncodedExponent = ActualExponent + 127

Exponent field in single-precision

Enc Exp

- The exponent field has 8-bit, keeping a value in [0, 255]
  - 0 [1, 254]: A normal SP number
    255
  - We will discuss 0 and 255 soon = denormal
- The range of actual exponent: [-126, 127] : Act Exp Enc Exp
  - Excess-127 representation!



$$\pm 1.x \times 2^{E} \quad \text{and } E \in [-126, 127]$$

Encoded = E + 127

Bits in the exponent field
1 .. 254

Questions: Excess-127

Enc Exp

- Given the eight bits in the exponent field of single-precision FP numbers, find the actual exponents in decimal.

0111 1111 − 127

0000 0100 − 127

1000 0001 − 127

1001 0000 − 127

Example: Read Single-Precision FP numbers

- What number (in decimal) is represented by the following single-precision floating-point number?

1 10000001 010 0000 0000 0000 0000 0000

S E X

Enc Exp → ActExp

Bin → Dec

S = 1

Fraction = $01000\ldots00_2$

Encoded exponent = 10000001_2 = 129 (as 8-bit unsigned number)

Actual exponent = 129 – 127 = 2

Dec → Bin

ActExp → EcExp

The value is

$$(-1)^1 \times (1 + 0.01_2) \times 2^{(129-127)}$$

E

$$= (-1) \times 1.25 \times 2^2$$

$$= -5$$

Question

What is the actual exponent of the following single-precision floating-point number?

What is its value in decimal?

0x C1C0 0000

↓

bin

↓

S EncExp Frac

↓

Act Exp

Example: Convert to Single-Precision FP numbers

Represent 4.75 with a single precision floating-point number

$\pm 1.\underline{x} \times 2^{ActExp}$

$\pm 1.x \times 2^{EncExp-127}$

S EncExp Fraction

x

1, 8 bit, (23-bit)

Solutions

Represent 4.75 with a single precision floating-point number

$$4.75 = 100.11_2 = (-1)^0 \times 1.0011_2 \times 2^2$$

S = 0

Fraction = $0011000\ldots00_2$

EncodedExponent = 2 + Bias = 2 + 127 = 129 = 10000001_2

0 10000001 001 1000 0000 0000 0000 0000

0x4098 0000

Single-Precision Range (Normal Numbers)

- In normal SP FP numbers, encoded exponents are in [1, 254]
  - 00000000_2 and 11111111_2 are reserved

- What is the smallest positive value of normal SP FP numbers?

0 | 0 0000001 | 0000 00

- What is the largest positive value of normal SP FP numbers?

0 | 1111 1110 | 1111 111

1	8	23

Single-Precision Range (Normal Numbers)

- In normal SP FP numbers, exponents are from 1 to 254
  - 00000000_2 and 11111111_2 are reserved

- Smallest positive value
  - Exponent: $00000001_2 \Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0

$$1.0 \times 2^{-126} \approx 1.2 \times 10^{-38}$$

How do we represent 0.0?

- Largest positive value
  - Exponent: $11111110_2 \Rightarrow$ actual exponent = 254 – 127 = 127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0

$$2.0 \times 2^{+127} \approx 3.4 \times 10^{+38}$$

Denormalized/subnormal Numbers

- Denormalized number: the exponent field is 0
  - The actual exponent is always – 126 for single precision numbers
  - The hidden bit is 0

$$v = (-1)^S \times (0.\text{Fraction}) \times 2^{-126}$$

- Denormalized numbers can represent numbers smaller than normal numbers
  - Allow for gradually approaching to 0, with diminishing precision

0	0000 0000	000 0000 0000 0000 0001

$$0.00000000000000000000001 \times 2^{-126} = 2^{-129}$$

Representation of 0

- 0 is a denormalized number !

All bits in exponent and fraction are 0.

But the sign can be 0 or 1. So we have two 0's!

0	0000 0000	000 0000 0000 0000 0000

1	0000 0000	000 0000 0000 0000 0000

$$x = (-1)^S \times (0.0) \times 2^{-126} = \pm 0.0$$

Infinities and NaN

Exponent = 1111 1111 (255)

- If fraction = 000…0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- If fraction ≠ 000…0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

Try these in Python:

```
float('inf') + 1.0
float('inf') + float('-inf')
```

IEEE Floating-Point Format: double precision

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

Sign
0 : non-negative,
1: negative

Exponent in
excess-k representation

Bits after the binary point
There is a hidden 1 !

$$\text{value} = (-1)^{S} \times (1.\text{Fraction}) \times 2^{(\text{EncodedExponent} - \text{Bias})}$$

Exponent in single-precision: excess-127: Bias = 127.

Exponent in double-precision: excess-1023: Bias = 1023

Single precision vs double precision

IEEE FP

	Single	Double
Total number of bits	32	64
Number of bits in exponent	8	11
Number of bits in fraction	23	52
Bias	127	1023
Smallest positive value (normal values)	1.0×2^{-126} $\approx 1.18 \times 10^{-38}$	1.0×2^{-1022} $\approx 2.2 \times 10^{-308}$
Largest positive value	$2.0 \times 2^{+127}$ $\approx 3.4 \times 10^{+38}$	$2.0 \times 2^{+1023}$ $\approx 1.8 \times 10^{+308}$
Precision	23 bits $\approx$ 6 dec. digits	52 bits $\approx$ 16 dec. digits

F and D Extensions in RISC-V

RISC-V I32

CPU

I Reg File

FP Reg File

- F for float and D for double
  - D is a superset. If D is supported, F is supported
- Separate FP register file (RF) consisting of 32 FP registers
  - ~~In F,~~ each register can hold a float
  - In D, each register can hold a float or a double

`f0, f1, … f30, f31`

f0 is not a special register

- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact

FP register name and calling convention

FP Registers	Name	Usage
f0 - f7	ft0 - ft7	FP temporary registers. Not preserved
f8 - f9	fs0 - fs1	Callee saved registers. Preserved
f10 - f11	fa0 - fa1	First 2 arguments. Return values. Not preserved
f12 - f17	fa2 - fa7	6 more arguments. Not preserved
f18 - f27	fs2 - fs11	Callee saved registers. Preserved
f28 - f31	ft8 - ft11	FP temporary registers

12 callee saved registers. 12 temporary registers. 8 argument registers.

Load/store for FP numbers

- FP load and store instructions
  - w for SP and d for DP

```
flw, fsw, fld, fsd

# same memory addressing modes
# base address is an integer
flw     f8, 0(sp)              # single-precision
fsw     f8, 4(sp)

fld     f9, 8(s1)              # double-precision
fsd     f9, 16(s1)
```

FP Arithmetic

- Single-precision arithmetic

```
fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s

# f0 = f1 + f6
fadd.s      f0, f1, f6
```

Same as Int

- Double-precision arithmetic

```
fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d

# f1 = f2 * f3
fmul.d      f1, f2, f3
```

32 32

32?

Int Mult

Lowers 2

higher 32

FP Comparison and Branch

look for sure

- Single- and double-precision comparison

  `f.eq.s, f.lt.s, f.le.s`

  `f.eq.d, f.lt.d, c.le.d`

- Result, 0 or 1, is saved in an integer destination register
  - Use beq or bne to branch on comparison result

```
# if f3 < f4, goto loop
f.lt.d  t0, f3, f4      # t0 = f3 < f4
bne     t0, x0, loop    # if t0
```

Compare with x0
No need to compare with 1

Floating point precision

- Be mindful when you compare two FP numbers for equal

$$0.1 * 3 \mathrel{!=} 0.3$$

$\neq 0$ $\qquad 10^{-17}$

$$0.1 * 3 - 0.3 = 5.55111512312578E\text{-}17$$

0.1

```
python
>>> float.hex(0.1)
'0x1.999999999999ap-4'
>>> float.hex(0.1*3)
'0x1.3333333333334p-2'
>>> float.hex(0.3)
'0x1.3333333333333p-2'
```

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail
  - Need to validate parallel programs under varying degrees of parallelism

Example

$$(x + y) + z \neq x + (y + z)$$

If $(x + y)$ is computed first, the result is 0. After adding z, the result is 1.

If $(y + z)$ is computed first, the result is y (because y is much larger than z). After adding x, the result is 0.

		$(x+y)+z$	$x+(y+z)$
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	$y+z$ 1.50E+38
z	1.0	1.0	
		1.00E+00	0.00E+00

FP Example: °F to °C

C code:

```
float f2c (float fahr)
{
  return ((5.0/9.0)*(fahr - 32.0));
}
```

fahr in f10, return value in f10.

Constants 5.0, 9.0, and 32.0 are stored in (global) memory.

RISC-V code:

```
f2c: flw     f0, 0(gp)         # load 5
     flw     f1, 4(gp)         # load 9
     fdiv.s  f0, f0, f1        # compute 5/9
     flw     f1, 8(gp)         # load 32
     fsub.s  f10, f10, f1      # compute fahr - 32
     fmul.s  f10, f0, f10      # multiply with 5/9
     jalr    x0, 0(ra)
```

Frequency of RISC-V instructions in SPEC CPU2006

Figure 3.22

17 most popular instructions
76% of all instr. executed

RISC-V Instruction	Name	Frequency	Cumulative
Add immediate	addi	14.36%	14.36%
Load doubleword	ld	8.27%	22.63%
Load fl. pt. double	fld	6.83%	29.46%
Add registers	add	6.23%	35.69%
Load word	lw	4.38%	40.07%
Store doubleword	sd	4.29%	44.36%
Branch if not equal	bne	4.14%	48.50%
Shift left immediate	slli	3.65%	52.15%
Fused mul-add double	fmadd.d	3.49%	55.64%
Branch if equal	beq	3.27%	58.91%
Add immediate word	addiw	2.86%	61.77%
Store fl. pt. double	fsd	2.24%	64.00%
Multiply fl. pt. double	fmul.d	2.02%	66.02%
Load upper immediate	lui	1.56%	67.59%
Store word	sw	1.52%	69.10%
Jump and link	jal	1.38%	70.49%
Branch if less than	blt	1.37%	71.86%
Add word	addw	1.34%	73.19%
Subtract fl. pt. double	fsub.d	1.28%	74.47%
Branch if greater/equal	bge	1.27%	75.75%

Summary

- Support for data types and arithmetic are part of ISA design
- RISC-V
  - Base supports integer add and sub
  - M extension supports mul and div
  - F and D extensions support FP operations
- Exceptions during arithmetic
  - Operations can overflow
  - Need to handle error with hardware and/or software
  - Floating-point has bounded range and precision
- Bits can be interpreted in many ways
  - Signed, unsigned, instruction, characters, FP numbers

Denormalized Numbers Examples

In the table, only the first number is a normal number

Exponent	Fraction	Actual exponent in decimal	Value
0000 0001	00000…00	-126	1.0×2^{-126} (normal number)
0000 0000	10000…00	-126	$0.1 \times 2^{-126} = 2^{-127}$
0000 0000	01000…00	-126	$0.01 \times 2^{-126} = 2^{-128}$
…			
0000 0000	00000…01	-126	$0.0…01 \times 2^{-126} = 2^{-149}$
0000 0000	00000…00	-126	$0.0…00 \times 2^{-126} = 0$

Conversion between datatypes

- Many conversion instructions. Study the reference card

  `fcvt.s.w, fcvt.d.w, fcvt.d.s, …`

```
addi       t0, x0, 5
fcvt.s.w   ft0, t0       # word to single-precision
fcvt.d.w   ft1, t0       # word to double-precision
# ft0 is a single-precision 5.0
# ft1 is a double-precision 5.0
```

Loading constants from memory:
[cse3666/91-f2c.s at master · zhijieshi/cse3666 (github.com)]
Using conversion instructions:
[cse3666/91-f2c-v2.s at master · zhijieshi/cse3666 (github.com)]

Question

Convert the decimal number 0.9 to a binary number

0						
2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}

Converting decimal to binary Example

Decimal	Binary
0.9	0.
0.9 * 2 = 1.8	0.1
0.8 * 2 = 1.6	0.11
0.6 * 2 = 1.2	0.111
0.2 * 2 = 0.4	0.1110

We can find the first 4 digits after the binary point by the following steps:

$0.9 * 2^4 = 14.4$

Convert 14 to 4-bit binary number and we get 1110.

Example: Convert to Single-Precision FP numbers

Represent –0.75 with a single precision floating-point number

$$-0.75 = -\,0.11_2 = (-1)^1 \times 1.1_2 \times 2^{-1}$$

S = 1

Fraction = $1000\ldots00_2$

EncodedExponent = –1 + Bias = –1 + 127 = 126 = 01111110_2

1 01111110 100 0000 0000 0000 0000 0000

0xBF40 0000

Reading Single-Precision FP Number - Solutions

```
0x C1C0 0000
1100 0001 1100 0000 0000 0000 0000 0000
```

$S = 1$

Fraction $= 10000\ldots00_2$

Encoded Exponent $= 10000011_2 = 131$ (as unsigned)

Actual exponent $= 131 – 127 = 4$

The value is

$$(-1)^1 \times (1 + 0.1_2) \times 2^{(131 - 127)}$$
$$= -1 \times 1.5 \times 2^4$$
$$= -24$$