Introduction to Computer Architecture
Chapter 3

Arithmetic for Computers

Hyungmin Cho

Department of Computer Science and Engineering
Sungkyunkwan University

Floating Point (1)

- What would be the output?

```c
double a, b, c, d;
a = 0.1;
b = 0.7;
c = 0.07;

d = a*b;

if(c==d) printf("%f is equal to %f\n",c,d);
else     printf("%f is NOT equal to %f\n",c,d);
```

```
0.070000 is NOT equal to 0.070000
```

Floating Point (2)

- Let’s see how a non-integer number is represented

```
float f1 = -58.0;

printf("f1 = %f\n", f1);
printf("f1 = 0x%X\n", f1);
```

```
warning: format ‘%X’ expects argument of type ‘long unsigned int’,
but argument 2 has type ‘double’
```

```
f1 = -58.000000
f1 =
```

Floating Point (3)

- Let's see how a non-integer number is represented

```
float f1 = -58.0;
unsigned int u1 = *((unsigned int *) &f1);

printf("f1 = %f\n", f1);
printf("u1 = 0x%X\n", u1);
```

```
f1 = -58.000000
u1 = 0xC2680000
```

Representing Non-Integer Numbers in Binary

$$10101.10101$$

$2^4=16$ $2^2=4$ $2^0=1$ $2^{-1}=0.5$ $2^{-3}=0.125$ $2^{-5}=0.03125$

$$= 16 + 4 + 1 + 0.5 + 0.125 + 0.03125 = 21.65625$$

- Question: How can we ‘encode’ the binary representation in computers?
  - Fixed-point format?
    - Fixed number of bits for integer / fractional parts
    - Limited range!

Floating Point

- Scientific notation (Significand + Magnitude)
  - -2.34×10^{56} $= -234 \times 10^{54}$ $= -0.234 \times 10^{57}$
  - $+0.002 \times 10^{-4}$ $= +0.2 \times 10^{-6}$ $= +2.0 \times 10^{-7}$
  - $+987.02 \times 10^{9}$ $= +9.8702 \times 10^{11}$ $= +98702 \times 10^{7}$

exponent

significand (or *coefficient*)

- Normalized (or *standard form*)
  - $1 \leq |\text{significand}| < 10$

Binary Representation

$$10101.10101$$

$$= 10101.10101 \times 2^0$$
$$= 1.010110101 \times 2^4$$

Range of the sigificand in binary normalized format:
  $1 \le |\text{significand}| < 2$

Binary number normalized format:
  $\pm 1.xxxxxxx_2 \times 2^{yyyy}$

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Single precision (32-bit)
- Double precision (64-bit)

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction (Mantissa)

$$x = (-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: **sign** bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is **Fraction** with the “1.” restored
- **Exponent**: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

Floating-Point Example

- Represent -0.75
  - $-0.75 = -\frac{3}{4} = -3\times\frac{1}{4} = -11_2 \times 2^{-2} = -1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent representation = -1 + Bias
    - Single: -1 + 127 = 126 = 01111110_2
    - Double: -1 + 1023 = 1022 = 01111111110_2

- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Example

- What number is represented by the single-precision float
  11000000101000…00
  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent representation = $10000001_2 = 129$

- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$
  $= (-1) \times 1.25 \times 2^2$
  $= -5.0$

Single-Precision Range

- Exponent representations “00000000” and “11111111” are reserved

- Smallest value
  - Exponent representation: 00000001 $\Rightarrow$ actual exponent = 1 - 127 = -126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$

- Largest value
  - Exponent representation: 11111110 $\Rightarrow$ actual exponent = 254 - 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Floating Point (4)

- What would be the output?

```c
float f2 = -0.1;
unsigned int u2 = *((unsigned int *) &f2);

printf("f2 = %f\n", f2);
printf("f2 = %3.20f\n", f2);
printf("u2 = 0x%X\n", u2);
```

```
f2 = -0.100000
f2 = -0.10000000149011611938
u2 = 0xBDCCCCCD
```

Limited Precision

- Let's represent 0.1 in single-precision Floating Point
  - $0.1 = \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \frac{1}{4096} \ldots..$
  - Exponent = -4 → $-4+127 = 123 = 01111011_2$
  - Significand = 1.10011001100110011001100110011001100110011001100…
  - Mantissa = 10011001100110011001101
  - 0 01111011 10011001100110011001101
  - 0011 1101 1100 1100 1100 1100 1100 1101 → 0x3DCCCCCD
  - 0x3DCCCCCD → 0.100000001490116119384765625

Floating-Point Addition

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$

1. Align decimal points
   - Shift number with smaller exponent
   - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
2. Add significands
   - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
3. Normalize result & check for over/underflow
   - 1.0015×10^{2}
4. Round and renormalize if necessary
   - 1.002×10^{2}

Floating-Point Addition

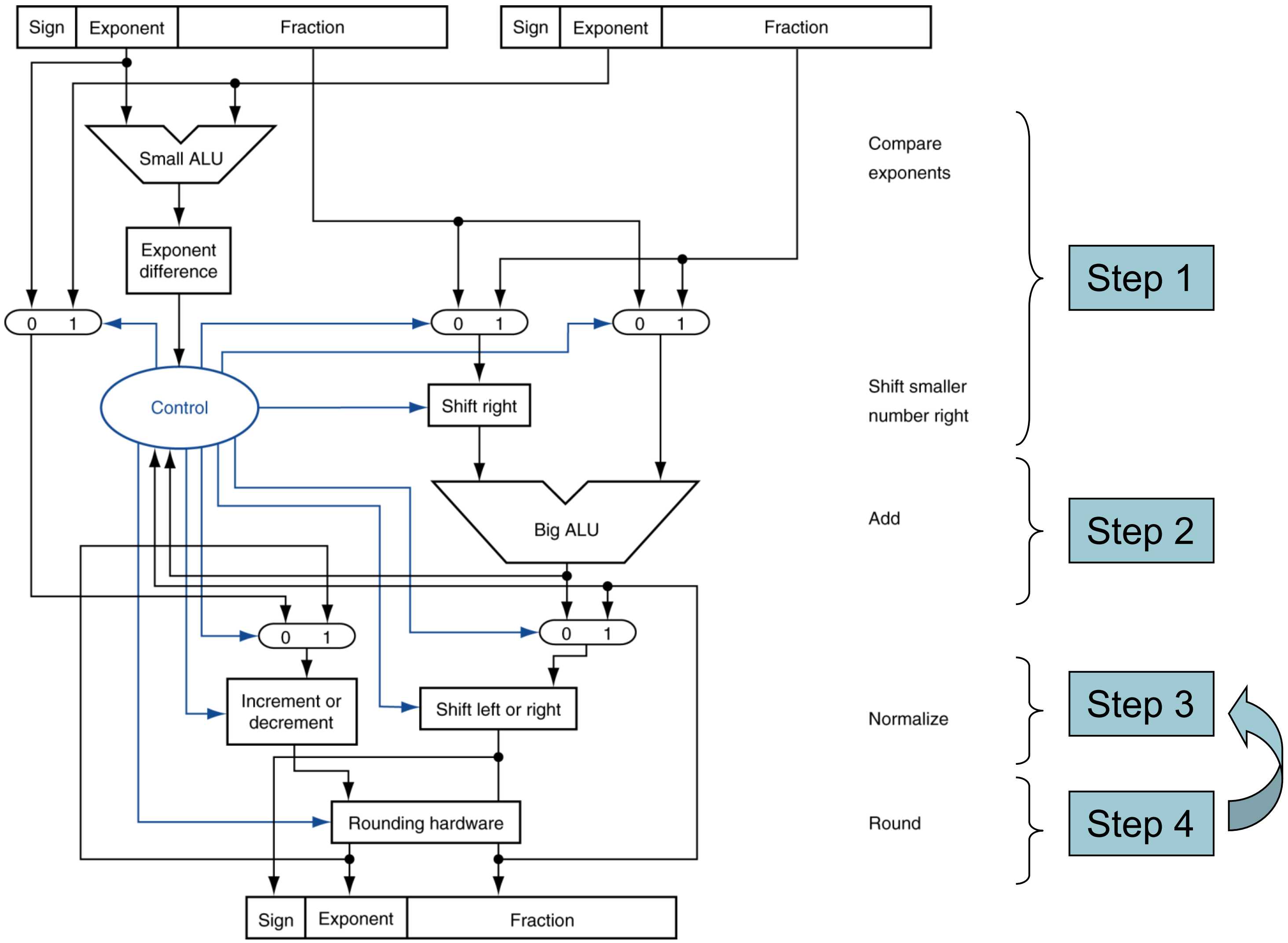
- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

1. Align binary points
   - Shift number with smaller exponent
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
2. Add significands
   - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
3. Normalize result & check for over/underflow
   - $1.000_2 \times 2^{-4}$, with no over/underflow
4. Round and renormalize if necessary
   - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than an integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



Why use Bias in Exponent?

- Consider two numbers: 2.5 and 0.75
  - $2.5 = 10.1_2 = 1.01_2 \times 2^1$
  - $0.75 = 0.11_2 = 1.1_2 \times 2^{-1}$

- If no bias is used (2's complement representation for exponent),
  - 2.5
    - Mantissa: $01000\ldots00_2$ Exponent representation: $1 = 00000001_2$
    - Binary representation: $000000001010000\ldots00_2$ = 0x00A00000
  - 0.75
    - Mantissa: $10000\ldots00_2$ Exponent representation : $-1 = 11111111_2$
    - Binary representation: $011111111110000\ldots00_2$ = 0x7FC00000

Which one is bigger?

Why use Bias in Exponent?

- Consider two numbers: 2.5 and 0.75
  - $2.5 = 10.1_2 = 1.01_2 \times 2^1$
  - $0.75 = 0.11_2 = 1.1_2 \times 2^{-1}$

- If bias (127) is used,
  - 2.5
    - Mantissa: $01000\ldots00_2$ Exponent representation : $1 + 127 = 10000000_2$
    - Binary representation: $01000000001000\ldots00_2$ = 0x40200000
  - 0.75
    - Mantissa: $10000\ldots00_2$ Exponent representation : $-1 + 127 = 01111110_2$
    - Binary representation: $00111111010000\ldots00_2$ = 0x3F400000

Which one is bigger?

Infinities and NaNs

- Exponent representation = 11111111, Fraction = 00000...00
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check

- Exponent = 11111111, Fraction ≠ 00000...00
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0

Denormalized Numbers

- What would be the smallest number without the denormalized numbers?
  - Suppose we can use "00000000" as a normal exponent: $1.0 \times 2^{-127} = 5.87\ldots \times 10^{-39}$
  - May not be small enough for some cases..
    - e.g., When $0 < |x - y| < 2^{-127}$, calculating $\frac{1}{x-y}$ would result in a divide by zero.

- How can we represent numbers smaller than 2^{-127} ?
  - First, reserve the exponent "00000000" as a special case
    - Now, the smallest number under the normalized form is 2^{-126}
  - For numbers smaller than 1.0×2^{-126}, use the "denormalized number" format.

Denormalized Numbers

- Exponent = 000...0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-(\text{Bias-1})}$$

- Denormalized numbers do not assume the significand starts with “1.0”.
  - The integer part of the significand is “0.”
  - The magnitude is always $\times 2^{-126}$
- Denormalized number example: $1.01_2 \times 2^{-130} = 0.000101_2 \times 2^{-126}$
  - Exponent = 00000000
  - Mantissa = $00010100\ldots00_2$
- The effective precision is decreased compared to the normalized numbers.