CPT_S 260 Intro to Computer Architecture
Lecture 10

Floating Point Arithmetic
February 2, 2022

Ganapati Bhat
School of Electrical Engineering and Computer Science
Washington State University

Announcements

- **Homework 2 is online**
  - Due next Friday

Recap: Fractional Part in Binary Format

- Repeatedly multiply the fraction by 2 and save the resulting integer digits. The digits for the binary number are the 0,1 in order of their computation.

- Convert 46.6875 to binary!

Recap: IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

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

- **S: sign bit (0 ⇒ non-negative, 1 ⇒ 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

Recap: Bias Exponent Representation

- **2's complement makes it difficult to compare exponents**
- **-1 is (111..111) where is 1 is (000….001). If we just look at 2's complement number, we cannot tell which has higher exponent**
- **IEEE 754 uses a bias of 127 for single precision, so an exponent of -1 is represented by the bit pattern of the value -1 + 127_{ten}, or 126_{ten} = $0111\ 1110_{two}$**
- **+1 → 1 + 127, or 128_{ten} = $1000\ 0000_{two}$**
- **The exponent bias for *double* precision is 1023.**

Single-Precision Range

- **Exponents 00000000 and 11111111 reserved**
- **Smallest value**
  - Exponent (also called biased exponent): 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 (also called biased exponent): 11111110 $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- **Exponents 0000…00 and 1111…11 reserved (next slides show for what these exponents are used)**
  - Smallest value
  - Exponent: 00000000001 $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
  - Exponent: 11111111110 $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Special Cases

- **Zero**
  - Sign bit = 0; biased exponent = all 00 bits; and the fraction = all 00 bits;
- **Positive and Negative Infinity**
  - Sign bit = 00 for positive infinity, 11 for negative infinity; biased exponent = all 11 bits; and the fraction = all 00 bits;
- **NaN (Not-A-Number)**
  - Sign bit = 0 or 1; biased exponent = all 11 bits; and the fraction is anything but all 00 bits. NaN's occurs when one does an invalid operation on a floating point value, such as dividing by zero, or taking the square root of a negative number.

Therefore, for Infinities and NaNs, we have

- **Exponent = 111…1, Fraction = 000…0**
  - ±Infinity –sign bit =0 for (+); sign bit =1 for (-)
  - Can be used in subsequent calculations, avoiding need for overflow check
- **Exponent = 111…1, Fraction ≠ 000…0**
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - » e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

Floating-Point Precision

- **Relative precision**
  - All fraction bits are significant
- **Single: approx 2^{-23}**
  - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- **Double: approx 2^{-52}**
  - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

Floating-Point Example #1

- **Represent –0.75**
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = 1000…00
  - Exponent = –1 + Bias
    - Single: –1 + 127 = 126 = 01111110
    - Double: –1 + 1023 = 1022 = 01111111110
- **Single: 1011111101000…00**
- **Double: 1011111111101000…00**

Floating-Point Example #2

- **What number is represented by the single-precision float**
- **11000000101000…00**
  - S = 1
  - Fraction = 01000…00
  - Exponent = 10000001 = 129
- **$x = (–1) \times (1 + .01) \times 2^{(129-127)}$**
  - $= (–1) \times 1.25 \times 2^2$
  - $= –5.0$

Overflow and Underflow

- Overflow occurs when a number is larger than the largest number that can be represented
- Underflow occurs when a number is smaller than the smallest number that can be represented

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}

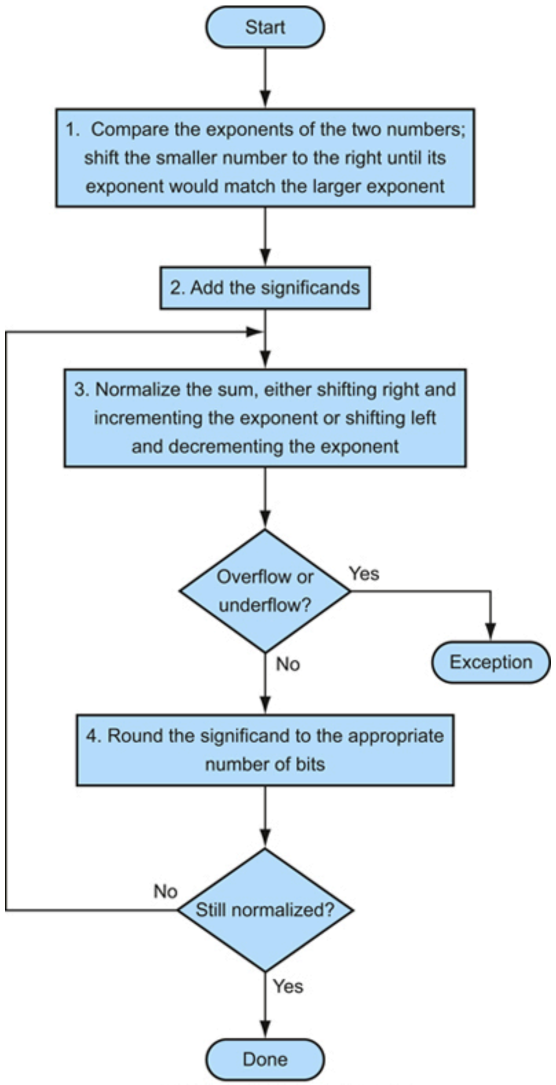
Floating-Point Addition

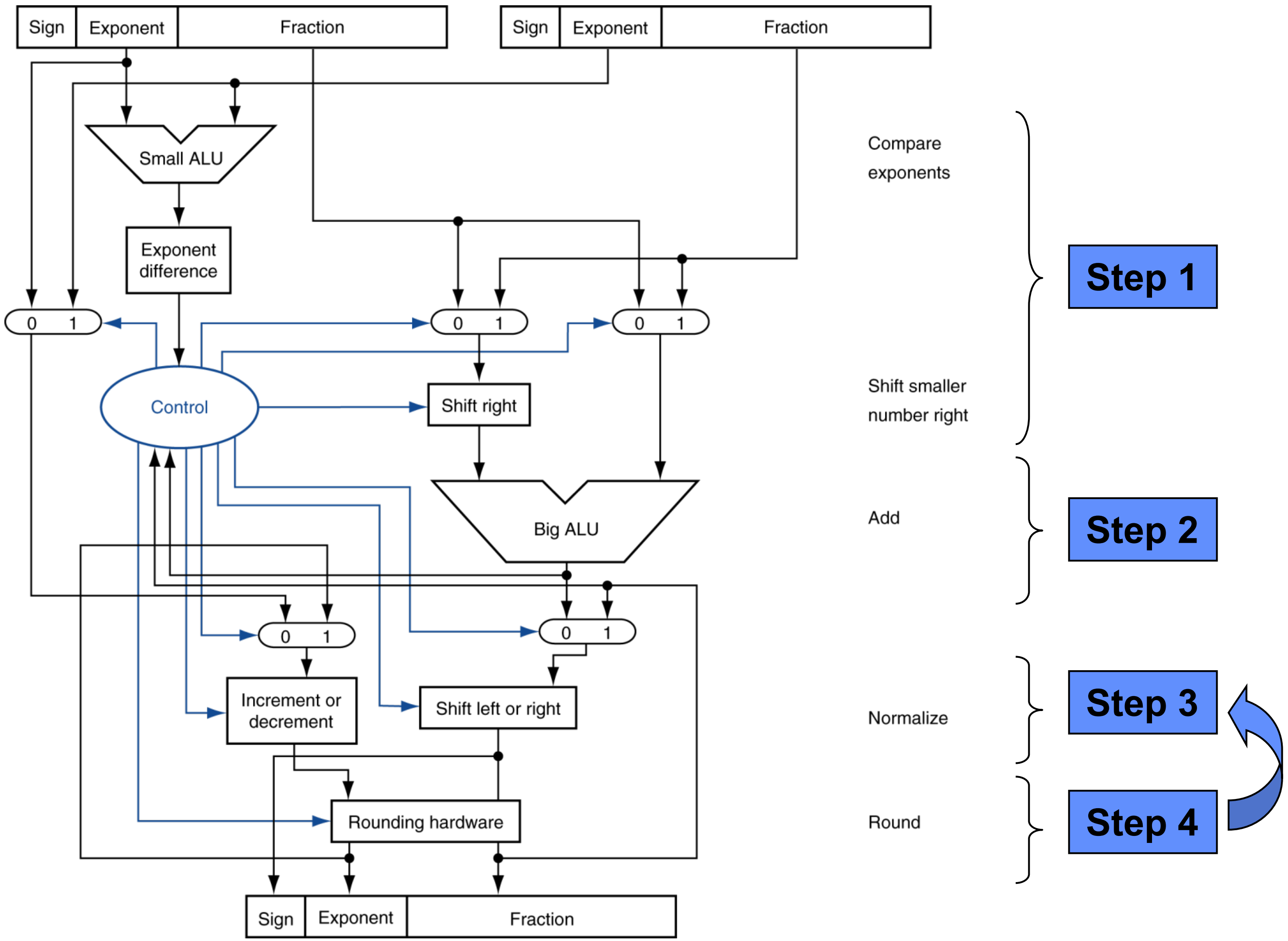
- **Now consider a 4-digit binary example**
  - $1.000 \times 2^{-1} + -1.110 \times 2^{-2}$ (0.5 + –0.4375)
- **1. Align binary points**
  - Shift number with smaller exponent
  - $1.000 \times 2^{-1} + -0.111 \times 2^{-1}$
- **2. Add significands**
  - $1.000 \times 2^{-1} + -0.111 \times 2^{-1} = 0.001 \times 2^{-1}$
- **3. Normalize result & check for over/underflow**
  - 1.000×2^{-4}, with no over/underflow
- **4. Round and renormalize if necessary**
  - $1.000_2 \times 2^{-4}$ (no change) $= 0.0625_{10}$

FP Adder Hardware

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

FP Adder Hardware





Floating-Point Multiplication

- **Consider a 4-digit decimal example**
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- **1. Add exponents**
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- **2. Multiply significands**
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- **3. Normalize result & check for over/underflow**
  - 1.0212×10^{6}
- **4. Round and renormalize if necessary**
  - 1.021×10^{6}
- **5. Determine sign of result from signs of operands**
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- **Now consider a 4-digit binary example**
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- **1. Add exponents**
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- **2. Multiply significands**
  - $1.000_2 \times 1.110_2 = 1.110_2 \;\Rightarrow\; 1.110_2 \times 2^{-3}$
- **3. Normalize result & check for over/underflow**
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- **4. Round and renormalize if necessary**
  - $1.110_2 \times 2^{-3}$ (no change)
- **5. Determine sign: +ve × –ve ⇒ –ve**
  - $-1.110_2 \times 2^{-3} = -0.21875$

Accurate Arithmetic

- **IEEE Std 754 specifies additional rounding control**
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- **Not all FP units implement all options**
  - Most programming languages and FP libraries just use defaults
- **Trade-off between hardware complexity, performance, and market requirements**

Guard and Round Bits

- **Used to hold intermediate operands before truncating**
- **Guard is the first of two extra bits**
- **Round is the second extra bit**

Number	Guard	Round

- **Example**
- $2.56 \times 10^{0} + 2.34 \times 10^{2}$

IEEE 754 Rounding Modes

- **Always round up (Towards infinity)**
- **Always round down (Towards negative infinity)**
- **Truncate**
- **Round to nearest even**
- **Tricky in the half way case (0.5)**
  - If the LSB retained would be odd, add one
  - If the LSB would be even, truncate
  - Gives a zero in the LSB

Sticky Bit

- **A bit used to track whenever there are non-zero bits to the right of the round bit**
- **Helps differentiate between 0.50000000…0 and 0.500000…1 when rounding**
- $5.01 \times 10^{-1} + 2.34 \times 10^{2}$

Sticky Bit

- A bit used to track whenever there are non-zero bits to the right of the round bit
- Helps differentiate between 0.50000000…0 and 0.500000…1 when rounding
- $5.01 \times 10^{-1} + 2.34 \times 10^{2}$

- **Without sticky bit**
- Add 0.0050 and 2.34
- Sum of 2.3450
- Round off to 2.34

- **With sticky bit**
- Add 0.0050 and 2.34
- Sticky bit would be set
- Sum of 2.3450 with sticky bit 1
- Round off to 2.35