National Yang Ming Chiao Tung University
Computer Architecture & System Lab

# Lecture 7: Floating Point

## CS10014 Computer Organization

Tsung Tai Yeh
Department of Computer Science
National Yang Ming Chiao University

# Acknowledgements and Disclaimer

- Slides were developed in the reference with
  - CS 61C at UC Berkeley
    - https://inst.eecs.berkeley.edu/~cs61c/sp23/
  - CS 252 at UC Berkeley
    - https://people.eecs.berkeley.edu/~culler/courses/cs252-s05/
  - E85 at HMC
    - https://pages.hmc.edu/harris/class/e85/old/fall21/

# Outline

- Review of Numbers
- Floating-point numbers
- The implicit leading 1
- Floating-point number addition

# Number Systems

- Fixed-point notation
  - Has an implied binary point between the integer and fraction bits

0110.1100

Integer bits    Fraction bits

$$= 2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$$

# Number Systems

- Fixed-point notation
  - Signed fixed-point numbers can use either two's complement or sign/magnitude notation
  - E.g. How to represent -2.375 using fixed-point notations with four integer and four fraction bits

(a) 2.375 = 0010.0110 (absolute value)
(b) 1010.0110 (sign and magnitude)
(c) -2.375 = 1101.1010 (two's complement)

Two's complement inverts the bit of the absolute value and adding a 1 to the LSB

# Saturating Arithmetic

- Fixed point **overflow** is usually bad
  - Produces undesired artifacts:
    - Video: dark pixel in middle of bright pixels
    - Audio: clicking sounds
- **Saturating arithmetic**
  - Instead of overflowing, use largest value
  - In U4.4: 11000000 + 01111000 = 11111111

    12 + 7.5 = 15.9375

# Floating vs. Fixed Point Numbers

- **Floating Point** is preferred for general-purpose computing where programming time is most important
- **Fixed Point** is preferred for signal processing performance, power, and hardware cost matter most
  - Machine learning, video

# Number Systems

- Floating-point number is composed of a
  - Sign
  - Mantissa (M)
  - Base (B)
  - Exponent (E)

# Floating-Point Numbers

- Binary point floats to the right of the most significant 1
- Similar to decimal scientific notation

- For example, write $273_{10}$ in scientific notation:

$$273 = 2.73 \times 10^2$$

- In general, a number is written in scientific notation as:

$$\pm M \times B^E$$

  - **M** = mantissa
  - **B** = base
  - **E** = exponent
  - In the example, M = 2.73, B = 10, and E = 2

# Floating-Point Numbers

- IEEE 754 32-bit Floating-point format
  - What is the floating-point representation of decimal number 228?
    - $228_{10} = 11100100_2 = 1.11001_2 \times 2^7$
    - The sign bit is positive (0)
    - The 8 exponent bits give the value 7
    - The remaining 23 bits are mantissa (111 001….)

$$\pm M \times B^E$$

| 1 bit | 8 bits | 23 bits |
|---|---|---|
| 0 | 00000111 | 111 0010 0000 0000 0000 0000 |
| **Sign** | **Exponent** | **Mantissa** |

# Floating-Point Numbers

- IEEE 754 32-bit Floating-point format
  - The exponent needs to represent both positive and negative exponents
    - Uses a biased exponent, which is the original exponent plus a constant bias
    - 32-bit floating-point uses a bias of 127

$$value = (-1)^{s} \times 1.M \times 2^{E-127}$$

# Floating-Point Numbers

- **How to convert 10.875 to IEEE 754 FP Format ?**
  - Step 1: Write the number in binary
    - $1010.111_2 = 1.010111000... * 2^3$
  - Step 2: Determine Sign/Exponent/Mantissa
    - Sign = Positive ->0
    - Exponent: 3-(-127) = 130 -> $1000\ 0010_2$
    - Mantissa: 1010 1110 0000 0000 0000 0000
  - Step 3: Concatenate
    - 0100 0001 0101 0111 0000 0000 0000 0000

| S | Exponent | Mantissa |
|---|---|---|
| 0 | 100 0001 0 | 101 0111 0000 0000 0000 0000 |

# Floating-Point Numbers

- IEEE 754 32-bit Floating-point format
  - How to represent $1.11001_2 \times 2^7$ in IEEE 754 format?
    - For the exponent 7, the biased exponent is $7 - (-127) = 134 = 10000110_2$
    - With an implicit leading one

$$value = (-1)^s \times 1.M \times 2^{E-127}$$

| 1 bit | 8 bits | 23 bits |
|---|---|---|
| 0 | 10000110 | 110 0100 0000 0000 0000 0000 |
| **Sign** | **Biased Exponent** | **Fraction** |

# Floating-Point Numbers

- **How to convert 0xC3CC0000 to decimal?**
  - Step 1: Write the number in binary
    - 1100 0011 1100 1100 0000 0000 0000 0000
    - C 3 C C 0 0 0 0
  - Step 2: Determine Sign/Exponent/Mantissa
    - Sign = Negative ->1
    - Exponent: $1000\ 0111_2$ -> 135-(127) = 8
    - Mantissa: 1001 1000 … -> 1.001 1000 = $1 + 2^{-3} + 2^{-4}$
    - $(1 + 2^{-3} + 2^{-4}) * 2^8 = 2^8 + 2^5 + 2^4 = 304$

# The implicit Leading 1

- Our mantissa is guaranteed not to have any leading zeros
  - If we wanted to write $0.234*10^5$, we'd instead write it as $2.34*10^4$
- In binary, every digit is only either 1 or 0
  - Since the MSB can't be 0, it must therefore be 1
  - **If the first bit will always be 1, we don't need to store it!**
  - We can save 1 bit (or alternatively add another bit of precision) to the mantissa by not including the MSB of the mantissa
    - This is known as the implicit 1
    - The resulting mantissa is a "normalized" number

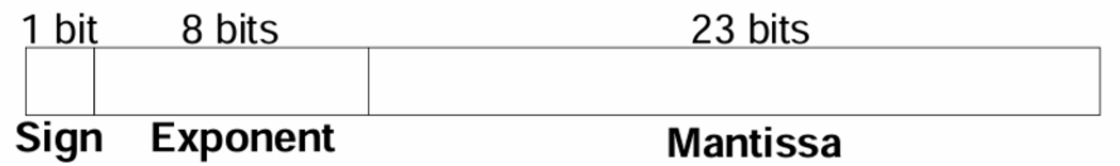
# The implicit Leading 1

- **How to convert 10.875 to IEEE 754 FP Format?**
  - Step 1: Write the number in binary
    - $1010.111_2 = 1.010111000... * 2^3$
  - Step 2: Determine Sign/Exponent/Mantissa
    - Sign = Positive ->0
    - Exponent: 3-(-127) = 130 -> $1000\ 0010_2$
    - Mantissa: 0101 1100 0000 0000 0000 000
  - Step 3: Concatenate
    - 0100 0001 0010 1110 0000 0000 0000 0000

**S** **Exponent** **Mantissa**

# Takeaway Question

- How to represent $-58.25_{10}$ in the IEEE 754 format with implicit leading 1?



# Takeaway Question

- How to represent $-58.25_{10}$ in the IEEE 754 format with implicit leading 1?

Write **$-58.25_{10}$** in floating point (IEEE 754)

1. Convert magnitude of decimal to binary:

   **$58.25_{10}$ = $111010.01_2$**

2. Write in binary scientific notation:

   **$1.1101001 \times 2^5$**

3. Fill in fields:

   **Sign bit:** 1 (negative)
   **8 exponent bits:** (127 + 5) = 132 = **$10000100_2$**
   **23 fraction bits:** **110 1001 0000 0000 0000 0000**

in hexadecimal: **0xC2690000**

# Floating-Point Special Cases

| Number | Sign | Exponent | Fraction |
|---|---|---|---|
| 0 | X | 00000000 | 00000000000000000000000 |
| ∞ | 0 | 11111111 | 00000000000000000000000 |
| - ∞ | 1 | 11111111 | 00000000000000000000000 |
| NaN | X | 11111111 | non-zero |

# Floating-Point Precision

- **Single-Precision:**
  - 32-bit
  - 1 sign bit, 8 exponent bits, 23 fraction bits
  - bias = 127

| Format | Total Bits | Sign Bits | Exponent Bits | Fraction Bits |
|---|---|---|---|---|
| single | 32 | 1 | 8 | 23 |
| double | 64 | 1 | 11 | 52 |

- **Double-Precision:**
  - 64-bit
  - 1 sign bit, 11 exponent bits, 52 fraction bits
  - bias = 1023

# Rounding & Overflow

- **Overflow:** number too large to be represented
- **Underflow:** number too small to be represented
- **Rounding modes:**
  - Down
  - Up
  - Toward zero
  - To nearest

# Rounding & Overflow

- **Example:** round 1.100101 (1.578125) to only 3 fraction bits
  - **Down:** 1.100
  - **Up:** 1.101
  - **Toward zero:** 1.100
  - **To nearest:** 1.101 (1.625 is closer to 1.578125 than 1.5 is)

# Floating-Point Addition

## 1. Extract exponent and fraction bits

| | 1 bit | 8 bits | 23 bits |
|---|---|---|---|
| 0x3FC00000 | 0 | 01111111 | 100 0000 0000 0000 0000 0000 |
| | Sign | Exponent | Fraction |

| | 1 bit | 8 bits | 23 bits |
|---|---|---|---|
| 0x40500000 | 0 | 10000000 | 101 0000 0000 0000 0000 0000 |
| | Sign | Exponent | Fraction |

For first number (N1): S = 0, E = 127, F = .1

For second number (N2): S = 0, E = 128, F = .101

## 2. Prepend leading 1 to form mantissa

N1: 1.1

N2: 1.101

# Floating-Point Addition

**3. Compare exponents**

127 – 128 = -1, so shift N1 right by 1 bit

**4. Shift smaller mantissa if necessary**

shift N1's mantissa: 1.1 >> 1 = 0.11 $(\times 2^1)$

**5. Add mantissas**

$$\begin{array}{r} 0.11 \times 2^1 \\ + \; 1.101 \times 2^1 \\ \hline 10.011 \times 2^1 \end{array}$$

# Floating-Point Addition

6. **Normalize mantissa and adjust exponent if necessary**

   $10.011 \times 2^1 = 1.0011 \times 2^2$

7. **Round result**

   No need (fits in 23 bits)

8. **Assemble exponent and fraction back into floating-point format**

   $S = 0, E = 2 + 127 = 129 = 10000001_2, F = 001100\ldots$

in hexadecimal: **0x40980000**

| 1 bit | 8 bits | 23 bits |
|---|---|---|
| 0 | 10000001 | 001 1000 0000 0000 0000 0000 |
| **Sign** | **Exponent** | **Fraction** |