UNIVERSITY of WASHINGTON

CSS 422 Hardware and Computer Organization

Computer Arithmetic II

Professor: Yang Peng

The slides are re-produced by the courtesy of
Dr. Arnie Berger and Dr. Wooyoung Kim

Topic

Computer Arithmetic II

- IEEE floating point

- Chapter 2.4, 2.5 (Null)

How to represent a real number in binary?

From Real to Binary Numbers

- Let’s convert decimal number **3.8125** to a binary number
  - **Integer** part: the same as the integer binary $11_2 = 3$
  - **Fractional** part:

1. **Multiply the fraction by two**
2. Write down the **integer part on right**
3. Repeat 1 and 2 **until there is no fractional part on left**
4. **Read the integer** part on right, from **top to bottom**

0.8125 * 2 = 0.625 + 1
0.625*2 = 0.25 + 1
0.25 *2 = 0.5 + 0
0.5*2 = 0.0 + 1
0.0 STOP HERE

$\mathbf{3.8125_{10} = 0011.1101_2}$

From Binary to Real Numbers

Binary $I_m I_{m-1} \ldots I_1 I_0 . F_1 F_2 F_3 \ldots F_{n-1} F_n =$

Decimal $I*2^m + I*2^{m-1} + \ldots + I*2^0 + F*2^{-1} + F*2^{-2}$

$+ \ldots + F*2^{-(n-1)} + F*2^{-n}$

$10.101_2 = 1*2^1 + 0*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$

$= 2 + 0.5 + 0.125 = 2.625$

Real Numbers and Errors

- Many fractions are **repeating infinitely**

  E.g., convert 0.6 to binary

Integer

0.6 * 2 = 0.2 -----1

0.2 * 2 = 0.4 -----0

0.4 * 2 = 0.8 -----0

0.8 * 2 = 0.6 -----1

0.6 * 2 = 0.2 -----1

0.2 * 2 = 0.4 -----0

0.4 * 2 = 0.8 -----0

0.8 * 2 = 0.6 -----1

So, 0.6 → 0.1001100110011001….. (will be repeated infinitely)

Rounding and Truncation

- **Keep the number of bits finite**
  - **Truncation**: The simplest technique – just drop unwanted bits

    E.g., 0.1101101 → 0.1101

  - **Rounding**: Better technique, but a bit complicated

  If the **value of the lost digits** is **greater than half of the least-significant bit of the retained digits**, add 1 to the LSB; otherwise drop.

  E.g., 0.1101101: If I want to lose the last three bits, what shall I do?

  0.1101101 = 0.1101 + 0.0000101

  → 0.1101 + 0.0001

  = 0.11**10**

LSB of retained digits: 0.0001 = 2^{-4}
Lost digits: 0.0000101 = $(2^{-5} + 2^{-7}) > 2^{-4} / 2$

How to represent a real number in a computer system?

Real Numbers in a Computer System

- Two main approaches: Fixed-point vs. Floating-point
- Fixed-point representation (**NOT used** now)
  - Divide the bits into **integer** part and **fraction** part
  - **The "Point" is fixed**
  - Easier but less flexible
- Floating-point representation (**IEEE standard**)
  - Divide the bits into **sign**, **exponent** and **mantissa**
  - **The "Point" is floating**
  - Match with scientific notation
  - Flexible but more complex

Fixed-Point Representation

- Fixed-point representation
  - Divide the bits for integer part and fraction part
  - For example, $3.625_{10} = 11.101_2$

0	0	1	1	1	0	1	0

Integer part **Fractional part**

- Not flexible
  - What if you really need to represent $1.984 * 10^{(-123)}$ in computer?
  - How many bits will be needed? (more than 372 bits)

Floating-Point Representation

- Floating-point representation
  - Divide the bits into **sign**, **exponent** and **mantissa**

- IEEE floating-point format
  1. IEEE **short real** or **single precision**: **32** bits

Sign (1)	Exponent (8)	Mantissa(23)

  2. IEEE **long real** or **double precision**: **64** bits

Sign (1)	Exponent (11)	Mantissa(52)

Floating-Point Representation - Single Precision

Steps to convert a real number to IEEE **Single Precision** floating-point representation

1. Convert decimal to binary
2. Normalize: moving the point left or right
3. Add **127** to the exponent
4. Mantissa is the one **after the floating point** in the normalized form
   – If the mantissa part is less than 23 bits, **add zeros at the end**
5. Put the corresponding numbers into each field

IEEE Floating point converter:

http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html

Floating-Point Representation - Single Precision

- **32-bit (single precision)** format

Sign (1)	Exponent (8)	Mantissa(23)

- Let's represent a real number to floating-point format

  E.g., $\mathbf{-3.8125_{10}}$

  $= \mathbf{-11.1101_2}$ (note that the integer part is **not** 2's complement)

  $= \mathbf{-1}.11101*2^1$ (**normalize**: scientific notation)

- **Sign** bit = 1, because this is a negative number
- **Exponent** bits = 1 + **127** (biased) = 128 = 10000000_2
- **Mantissa** bits: 111 0100 0000 0000 0000 0000

Steps of movement:
Move right/left: positive/negative

Therefore, the real number in floating-point representation is:

1 100 0000 0 111 01 00 0000 0000 0000 0000_2 = $C0740000

Floating-Point Representation - Single Precision

- **Normalization**
  - **"1" shall always appear as an integer part**
  - No need to represent this bit in the format -> save one bit

- **Biased exponent**
  - The exponent has 8 bits, meaning it can range from -127 to 127 (Here we assume that -128 will never appear).
  - Therefore, if we add 127 to the exponent, it will always be a non-negative number.
  - Assuming such a representation, 0~254 is then available for the exponent filed. How about 255?

Floating-Point Representation - Double Precision

Steps to convert a real number to IEEE **Double Precision** floating-point representation

1. Convert decimal to binary
2. Normalize: moving the point left or right
3. Add **1023** to the exponent
4. Mantissa is the one **after the floating point** in the normalized form
   – If the mantissa part is less than 52 bits, **add zeros at the end**
5. Put the corresponding numbers into each field

IEEE Floating point converter:

http://babbage.cs.qc.cuny.edu/IEEE-754.old/Decimal.html

Floating-Point Representation - Double Precision

- **64-bit (double precision)** format

Sign (1)	Exponent (11)	Mantissa(52)

- Let's represent a real number to floating-point format

E.g., **-3.8125_{10}**

$= \mathbf{-11.1101_2}$ (note that the integer part is **not** 2's complement)

$= \mathbf{-1}.11101*2^1$ (normalize: scientific notation)

- **Sign** bit = 1, since negative
- **Exponent** = 1 + **1023** (biased) = 1024 = $100\ 0000\ 0000_2$
- **Mantissa**: 1110 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Therefore in floating-point representation,

1 100 0000 0000 1110 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000_2 = $C00E800000000000

More about real numbers

- Why using biased exponent?
  - **Effect**: changing **negative** exponent value **to positive** value
  - **Motivation**: for **quick comparison** (bit-by-bit) of two real numbers

- Why adding 127 for single-precision floating numbers?
  - **Effect**: positive numbers in the rage of **0 to 254**
  - **Motivation**: **reserve 255 for special number usage**

Sign	Exponent (e)	Fraction (f)	Value
0	00⋯00	00⋯00	+0
0	00⋯00	00⋯01 ⋮ 11⋯11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00⋯01 ⋮ 11⋯10	XX⋯XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11⋯11	00⋯00	$+\infty$
0	11⋯11	00⋯01 ⋮ 01⋯11	SNaN
0	11⋯11	1X⋯XX	QNaN
1	00⋯00	00⋯00	−0
1	00⋯00	00⋯01 ⋮ 11⋯11	Negative Denormalized Real $-0.f \times 2^{(-b+1)}$
1	00⋯01 ⋮ 11⋯10	XX⋯XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
1	11⋯11	00⋯00	$-\infty$
1	11⋯11	00⋯01 ⋮ 01⋯11	SNaN
1	11⋯11	1X⋯XX	QNaN

- **NaN**: Not A Number
- **QNaN**: **Quiet** NaN
  - generated from an operation when the result is not mathematically defined
  - denote ***indeterminate*** operations
- **SNaN**: **Signaling** NaN
  - used to signal an exception when used in operations
  - can be to assign to uninitialized variables to trap premature usage
  - denote ***invalid*** operations