Computer Architecture

ICT 1019Y Week 02 Lecture

Floating-Point Numbers

Recap

- **With your neighbor, write 23 in the following forms:**
  - **(1) Unsigned (2) Sign-magnitude**
    (3) One's complement (4) Two's complement
  - **What's the one thing I need to tell you first?**
    - **Let's say: 12 bit long representation**
- Convert |23| to binary: 10111 (i.e. 16+4+2+1)
- **Answers are all the same! 0000 0001 0111**
  - **Because number is positive**

Recap

- **With your neighbor, write -23 in the following forms using a 12-bit long representation:**
  - **(1) Unsigned (2) Sign-magnitude**
    (3) One's complement (4) Two's complement
- **Unsigned – No representation possible**
- **Sign-Magnitude: 1000 0001 0111**
- **One's complement: 1111 1110 1000**
  - ***(extend 23 to 12 bits, and then invert)***
- **Two's complement: 1111 1110 1001**
  - ***(one's complement plus 1)***

Range

- **What is the smallest and largest 8-bit two's complement number?**
  - $\mathtt{XXXXXXXX}_2$
  - Smallest (negative) # = $\mathtt{10000000}_2$ = **-128**
  - Largest (positive) # = $\mathtt{01111111}_2$ = **127**

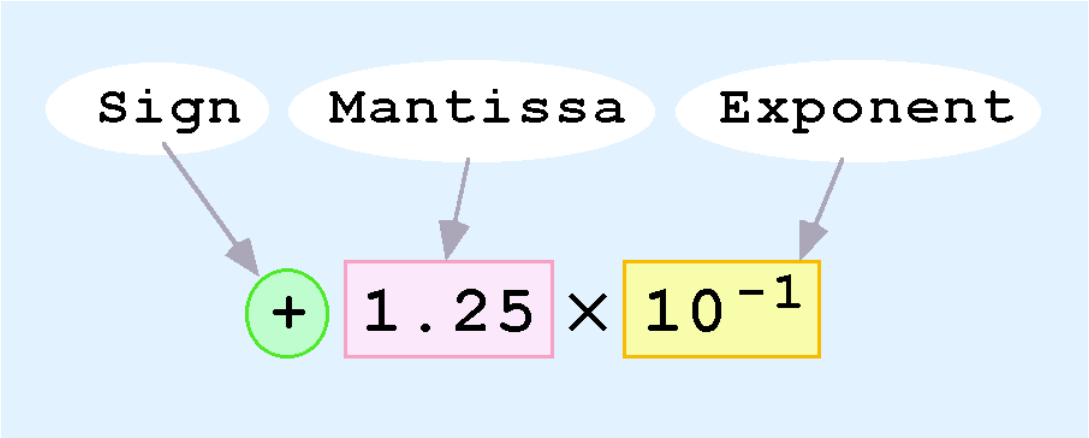
Floating-Point Numbers

Floating-Point Representation

- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in **scientific notation**
  - For example:
    - $0.125 = 1.25 \times 10^{-1}$
    - $5{,}000{,}000 = 5.0 \times 10^{6}$

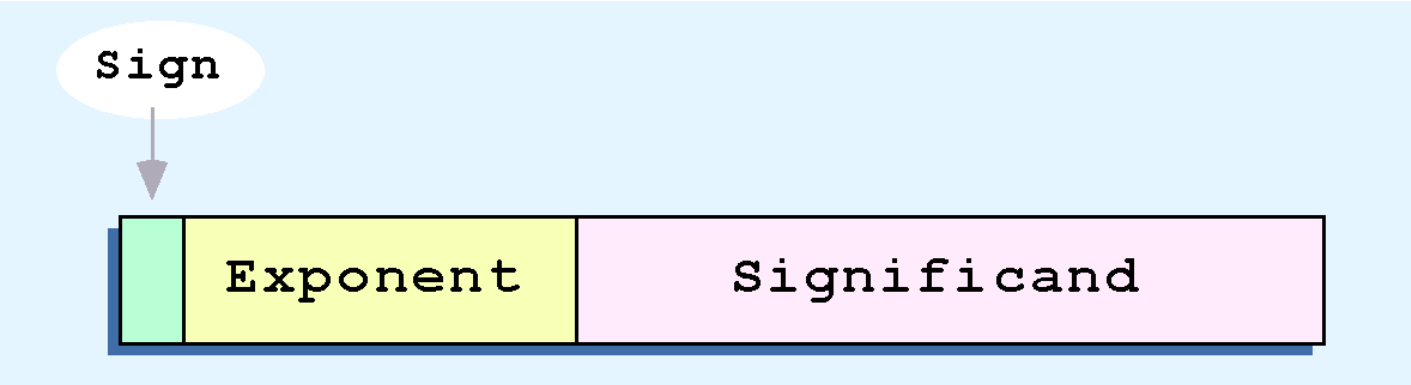
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



Floating-Point Representation

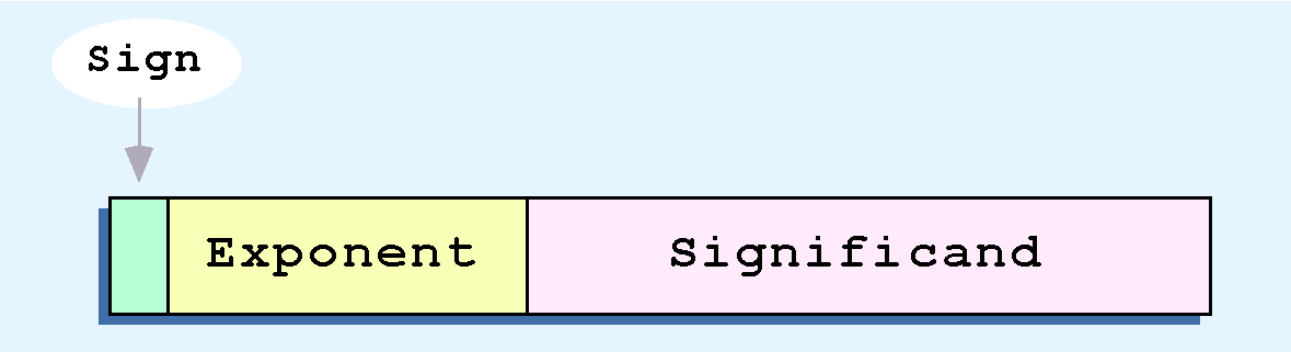
- Computer representation of a floating-point number consists of three fixed-size fields:
- This is the standard arrangement of these fields:



Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

Floating-Point Representation

- The one bit *sign* field is the sign of the stored value.
- The size of the *exponent* field determines the **range** of values that can be represented
- The size of the *significand* determines the **precision** of the representation



Floating-Point Errors

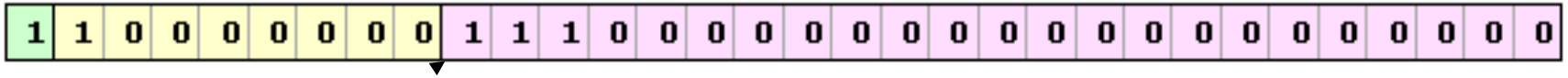
- When discussing floating-point numbers, it is important to understand the terms *range*, *precision*, and *accuracy*
- The **range** of a numeric integer format is the difference between the largest and smallest values that can be expressed
- **Accuracy** refers to how closely a numeric representation approximates a true value
- The **precision** of a number indicates how much information we have about a value

IEEE Floating-Point Representation

- The IEEE has established standards for floating-point numbers
- **IEEE-754 single precision** standard (**32 bits** long)
  - **8**-bit exponent (with a **bias of 127**)
  - **23**-bit significand
  - A “float” in C++
- **IEEE-754 double precision** standard (**64 bits** long)
  - **11**-bit exponent (with a **bias of 1023**)
  - **52**-bit significand
  - A “double” in C++

IEEE Floating-Point Representation

- Example: Express **-3.75** as a floating point number using IEEE *single* precision.
- Normalize according to IEEE rules:
  - $-3.75 = -11.11_2 = -1.111 \times 2^1$
  - The bias for *single precision* is 127, so add 127 + 1 = 128
    - This is the exponent saved to computer memory
  - The first 1 in the significand is implied, so we have:



(implied 1: not saved)

- To decode saved number with the implied 1 in the significand:
  - $-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$

IEEE Floating-Point Representation

- Using the IEEE-754 single precision floating point standard:
  - An exponent of 255 indicates a special value.
    - If the significand is zero, the value is ± infinity.
    - If the significand is nonzero, the value is NaN, “not a number,” often used to flag an error condition.
- Using the double precision standard:
  - An exponent of 2047 indicates a special value

- **Convert 12.5 to IEEE 754 single precision floating point:**

- Format requirements for single precision (32 bit total length):
  - 1 sign bit
  - 8 bit exponent (which uses a bias of 127)
  - 23 bit significant (which has an **implied 1. that is not stored in the field**)

- Convert 12.5 to binary: 1100.1×2^0
  - Normalize it in the IEEE way: 1.1001×2^3
  - Bias exponent: 3 + 127 = 130 (10000010 in binary)

- Result
  - Sign bit: **0**
  - Exponent (8 bits): **10000010**
  - Mantissa (23 bits): **10010000000000000000000** (padded out to 23 bits, leading 1 not shown!)
  - Thus, **0 | 10000010 | 10010000000000000000000**

Floating-Point Errors

- No matter how many bits we use in a floating-point representation, our model is finite
- Problem: Real numbers can be infinite, so our model can only *approximate* a real value
- At some point, every model breaks down, introducing errors into the calculations
- By using a greater number of bits in the model, we can reduce these errors, but we can never totally eliminate them

Floating-Point Errors

- Floating-point overflow and underflow can cause programs to crash
- **Overflow** occurs when there is no room to store the high-order bits resulting from a calculation
- **Underflow** occurs when a value is too small to store, possibly resulting in division by zero

Data Types

- **Where do I see all these data types in C/C++ programming?**
- `int` – Two's Complement number
- `unsigned int` – Plain old binary number
- `float` – IEEE single precision floating-point
- `double` – IEEE double precision floating-point