Floating Point
CSE 351 Autumn 2017

Instructor:
Justin Hsia

Teaching Assistants:
Lucas Wotton  Michael Zhang  Parker DeWilde  Ryan Wong
Sam Gehman  Sam Wolfson  Savanna Yee  Vinny Palaniappan

http://xkcd.com/571/
Administrivia

- Lab 1 Prelim due tonight at 11:59pm
  - Only submit `bits.c`
- Lab 1 due Friday (10/13)
  - Submit `bits.c, pointer.c, lab1reflect.txt`
- Homework 2 released tomorrow, due 10/20
  - On Integers, Floating Point, and x86-64
Unsigned Multiplication in C

Operands:

\[ w \text{ bits} \]

\[ u \]

\[ * \]

\[ v \]

True Product:

\[ 2w \text{ bits} \]

\[ u \cdot v \]

Discard \( w \) bits:

\[ w \text{ bits} \]

\[ \text{UMult}_w(u \ , \ v) \]

- **Standard Multiplication Function**
  - Ignores high order \( w \) bits

- **Implements Modular Arithmetic**
  - \[ \text{UMult}_w(u \ , \ v) = u \cdot v \mod 2^w \]
Multiplication with shift and add

- Operation \( u \ll k \) gives \( u \times 2^k \)
  - Both signed and unsigned

**Operands**: \( w \) bits

**True Product**: \( w + k \) bits

**Discard** \( k \) bits: \( w \) bits

**Examples**:
- \( u \ll 3 \)  \( \equiv \)  \( u \times 8 \)
- \( u \ll 5 \)  \( - u \ll 3 \)  \( \equiv \)  \( u \times 24 \)  \( \rightarrow 32 - 8 \)
- \( u \ll 4 \)  \( + u \ll 3 \)  \( \rightarrow 16 + 8 \)
- Most machines shift and add faster than multiply
  - *Compiler generates this code automatically*
Number Representation Revisited

- What can we represent in one word?
  - Signed and Unsigned Integers
  - Characters (ASCII)
  - Addresses

- How do we encode the following:
  - Real numbers (e.g. 3.14159)
  - Very large numbers (e.g. 6.02×10^23) — Avogadro's number
  - Very small numbers (e.g. 6.626×10^{-34}) — Planck's constant
  - Special numbers (e.g. ∞, NaN)
Floating Point Topics

- Fractional binary numbers
- IEEE floating-point standard
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard...
Representation of Fractions

- “Binary Point,” like decimal point, signifies boundary between integer and fractional parts:

  Example 6-bit representation:

  \[ xx \cdot yyyyy \]

  \[ 2^1 \xrightarrow{2^0} 2^{-1} \xrightarrow{2^{-2}} 2^{-3} \xrightarrow{2^{-4}} \]

  \[ 0.5 \xrightarrow{0.25} 0.125 \xrightarrow{0.0625} \]

- Example: \(10.1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}\)

- Binary point numbers that match the 6-bit format above range from 0 (00.0000\(_2\)) to 3.9375 (11.1111\(_2\))

\[
= 4 - 2^{-4} + \frac{1}{100.0000}
\]
Scientific Notation (Decimal)

- **Normalized form**: exactly one digit (non-zero) to left of decimal point

- Alternatives to representing 1/1,000,000,000
  - **Normalized**: \( 1.0 \times 10^{-9} \)  
  - Not normalized: \( 0.1 \times 10^{-8}, 10.0 \times 10^{-10} \)
Scientific Notation (Binary)

- Computer arithmetic that supports this called **floating point** due to the “floating” of the binary point
  - Declare such variable in C as `float` (or `double`)
Scientific Notation Translation

- Convert from scientific notation to binary point
  - Perform the multiplication by shifting the decimal until the exponent disappears
    - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
    - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$

- Convert from binary point to normalized scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
    - Example: $1101.001_2 = 1.101001_2 \times 2^3$

- Practice: Convert $11.375_{10}$ to binary scientific notation (normalized)
  $8 + 2 + 1 + 0.25 + 0.125$
  $2^3 + 2^1 + 2^0 + 2^{-2} + 2^{-3} = 1011.011_2 = 1.011011 \times 2^3$

- Practice: Convert $1/5$ to binary
  $\frac{1}{5} - \frac{1}{8} = \frac{3}{40}$, $\frac{3}{40} - \frac{1}{16} = \frac{1}{80} = \frac{1}{16} \left( \frac{1}{5} \right)$
  $\frac{1}{5} = \frac{1}{8} + \frac{1}{16} + \frac{1}{16} \left( \frac{1}{5} \right)$
  $= 0.0011_2$
Floating Point Topics

- Fractional binary numbers
- **IEEE floating-point standard**
- Floating-point operations and rounding
- Floating-point in C

- There are many more details that we won’t cover
  - It’s a 58-page standard...
IEEE Floating Point

- **IEEE 754**
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - Now supported by all major CPUs

- Driven by numerical concerns
  - **Scientists**/numerical analysts want them to be as *real* as possible
  - **Engineers** want them to be *easy to implement* and *fast*
  - In the end:
    - Scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but...
    - Hard to make fast in hardware
    - **Float operations can be an order of magnitude slower than integer ops**

\[ \text{FLOPs} \]
Floating Point Encoding

- Use normalized, base 2 scientific notation:
  - Value: \( \pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}} \)
  - Bit Fields: \((-1)^{S} \times 1.M \times 2^{(E-\text{bias})}\)

- Representation Scheme:
  - Sign bit (0 is positive, 1 is negative)
  - Mantissa (a.k.a. significand) is the fractional part of the number in normalized form and encoded in bit vector \( M \)
  - Exponent weights the value by a (possibly negative) power of 2 and encoded in the bit vector \( E \)

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
</tbody>
</table>

3 separate fields within 32 bits
The Exponent Field

- **Use biased notation**
  - Read exponent as unsigned, but with bias of \(2^{w-1}-1 = 127\)
  - Representable exponents roughly \(\frac{1}{2}\) positive and \(\frac{1}{2}\) negative
  - Exponent 0 (\(\text{Exp} = 0\)) is represented as \(E = 0b \ 0111 \ 1111\)

- **Why biased?**
  - Makes floating point arithmetic easier
  - Makes somewhat compatible with two’s complement

- **Practice:** To encode in biased notation, add the bias then encode in unsigned:
  - \(\text{Exp} = 1\) → 128 → \(E = 0b \ 0100 \ 0000\)
  - \(\text{Exp} = 127\) → 254 → \(E = 0b \ 1111 \ 1110\)
  - \(\text{Exp} = -63\) → 64 → \(E = 0b \ 0100 \ 0000\)
The Mantissa (Fraction) Field

\[ (-1)^S \times (1 \cdot M) \times 2^{(E - \text{bias})} \]

- Note the implicit 1 in front of the M bit vector
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000 is read as 1.1₂ = 1.5₁₀, not 0.1₂ = 0.5₁₀
  - Gives us an extra bit of precision

- Mantissa “limits”
  - Low values near \( M = 0b0...0 \) are close to \( 2^{E} \)
  - High values near \( M = 0b1...1 \) are close to \( 2^{E+1} \)
Peer Instruction Question

What is the correct value encoded by the following floating point number?

\[ \begin{array}{c}
S & E & M \\
\text{0b 0 10000000 11000000000000000000000} \\
\text{(128 - 127)} & \text{Exp = 1} & \text{Man = 1.110...0 (implicit)} \\
\end{array} \]


A. +0.75
B. +1.5
C. +2.75
D. +3.5
E. We’re lost...

\[ +\frac{1.11_2}{2} \times 2^1 \]

\[ 1.11_2 = 2^1 + 2^0 + 2^{-1} = 3.5 \]
Precision and Accuracy

- **Precision** is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy

- **Accuracy** is a measure of the difference between the actual value of a number and its computer representation

  - *High precision permits high accuracy but doesn’t guarantee it. It is possible to have high precision but low accuracy.*

  - **Example:** float pi = 3.14;
    - pi will be represented using all 24 bits of the mantissa (highly precise), but is only an approximation (not accurate)
Need Greater Precision?

- **Double Precision** (vs. Single Precision) in 64 bits

- C variable declared as `double`
- Exponent bias is now $2^{10} - 1 = 1023$, $\text{bias} = 2^{10} - 1$
- **Advantages:** greater precision (larger mantissa), greater range (larger exponent)
- **Disadvantages:** more bits used, slower to manipulate
Representing Very Small Numbers

- But wait... what happened to zero?
  - Using standard encoding $0x00000000 = 1.0 \times 2^{-127} \neq 0$
  - **Special case:** $E$ and $M$ all zeros = 0
    - Two zeros! But at least $0x00000000 = 0$ like integers
      - $0x80000000 = -\infty$

- New numbers closest to 0:
  - $a = 1.0...0_2 \times 2^{-126} = 2^{-126}$
  - $b = 1.0...01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit 1 are to blame
  - **Special case:** $E = 0$, $M \neq 0$ are denormalized numbers
Denorm Numbers

- Denormalized numbers
  - No leading 1
  - Uses implicit exponent of $-126$ even though $E = 0x00$

- Denormalized numbers close the gap between zero and the smallest normalized number
  - Smallest norm: $\pm 1.0...0_{two} \times 2^{-126} = \pm 2^{-126}$
  - Smallest denorm: $\pm 0.0...01_{two} \times 2^{-126} = \pm 2^{-149}$
    - There is still a gap between zero and the smallest denormalized number
Other Special Cases

- \( E = 0xFF, M = 0: \pm \infty \)
  - e.g. division by 0
  - Still work in comparisons!

- \( E = 0xFF, M \neq 0: \) Not a Number (NaN)
  - e.g. square root of negative number, 0/0, \( \infty - \infty \)
  - NaN propagates through computations
  - Value of \( M \) can be useful in debugging

- New largest value (besides \( \infty \))?
  - \( E = 0xFF \) has now been taken!
  - \( E = 0xFE \) has largest: \( 1.1...12 \times 2^{127} = 2^{128} - 2^{104} \)
# Floating Point Encoding Summary

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0x00</td>
<td>non-zero</td>
<td>± denorm num</td>
</tr>
<tr>
<td>0x01 – 0xFE</td>
<td>anything</td>
<td>± norm num</td>
</tr>
<tr>
<td>0xFF</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>0xFF</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
Summary

- Floating point approximates real numbers:
  - Handles large numbers, small numbers, special numbers
  - Exponent in biased notation (bias = $2^{w-1}-1$)
    - Outside of representable exponents is overflow and underflow
  - Mantissa approximates fractional portion of binary point
    - Implicit leading 1 (normalized) except in special cases
    - Exceeding length causes rounding

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Mantissa</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0</td>
<td>± 0</td>
</tr>
<tr>
<td>0x00</td>
<td>non-zero</td>
<td>± denorm num</td>
</tr>
<tr>
<td>0x01 – 0xFE</td>
<td>anything</td>
<td>± norm num</td>
</tr>
<tr>
<td>0xFF</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>0xFF</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>
An example that applies the IEEE Floating Point concepts to a smaller (8-bit) representation scheme. These slides expand on material covered today, so while you don’t need to read these, the information is “fair game.”
Tiny Floating Point Example

- 8-bit Floating Point Representation
  - The sign bit is in the most significant bit (MSB)
  - The next four bits are the exponent, with a bias of $2^{4-1} - 1 = 7$
  - The last three bits are the mantissa

- Same general form as IEEE Format
  - Normalized binary scientific point notation
  - Similar special cases for 0, denormalized numbers, NaN, $\infty$
## Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>M</th>
<th>Exp</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 000</td>
<td>-6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
<td>largest denorm</td>
</tr>
<tr>
<td>0</td>
<td>0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td>smallest norm</td>
</tr>
<tr>
<td>0</td>
<td>0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td>closest to 1 below</td>
</tr>
<tr>
<td>0</td>
<td>0110 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td>closest to 1 above</td>
</tr>
<tr>
<td>0</td>
<td>0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td>largest norm</td>
</tr>
<tr>
<td>0</td>
<td>1110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111 000</td>
<td>n/a</td>
<td>inf</td>
<td></td>
</tr>
</tbody>
</table>
Special Properties of Encoding

- Floating point zero \((0^+)\) exactly the same bits as integer zero
  - All bits = 0

- Can (Almost) Use Unsigned Integer Comparison
  - Must first compare sign bits
  - Must consider \(0^- = 0^+ = 0\)
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity