Floating Point
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

http://xkcd.com/899/
Administrivia

- Lab 1 due today at 5pm (prelim) and Friday at 5pm
  - Use Makefile and DLC and GDB to check & debug
- Homework 1 (written problems) released tomorrow
Integers

- Binary representation of integers
  - Unsigned and signed
  - Casting in C
- Consequences of finite width representations
  - Overflow, sign extension
- Shifting and arithmetic operations
- Multiplication
Multiplication

- What do you get when you multiply 9 x 9?

- What about $2^{30} \times 3$?

- $2^{30} \times 5$?

- $-2^{31} \times -2^{31}$?
Unsigned Multiplication in C

**Operands:**
- $w$ bits

**True Product:**
- $2w$ bits

**Discard $w$ bits:**
- $w$ bits

- **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 $u << k$ gives $u \times 2^k$
  - Both signed and unsigned

**Operands:** $w$ bits

**True Product:** $w + k$ bits

**Discard $k$ bits:** $w$ bits

- **Examples:**
  - $u << 3 = u \times 8$
  - $u << 5 - u << 3 = u \times 24$
  - 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. $6.02 \times 10^{23}$)
  - Very small numbers (e.g. $6.626 \times 10^{-34}$)
  - Special numbers (e.g. $\infty$, NaN)
Goals of Floating Point

- Support a wide range of values
  - Both very small and very large
- Keep as much precision as possible
- Help programmer with errors in real arithmetic
  - Support $+\infty$, $-\infty$, Not-A-Number (NaN), exponent overflow and underflow
- Keep encoding that is somewhat compatible with two’s complement
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: $\text{xx.yyyy}$

- $2^1 \quad 2^0 \quad 2^{-1} \quad 2^{-2} \quad 2^{-3} \quad 2^{-4}$

- 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$_2$)
Scientific Notation (Decimal)

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

- Alternatives to representing 1/1,000,000,000,000
  - 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`
Scientific Notation Translation

- Consider the number $1.011_2 \times 2^4$
  - To convert to ordinary number, shift the decimal to the right by 4
    - Result: $10110_2 = 22_{10}$
  - For negative exponents, shift decimal to the left
    - $1.011_2 \times 2^{-2} \Rightarrow 0.01011_2 = 0.34375_{10}$
  - Go from ordinary number to scientific notation by shifting until in *normalized* form
    - $1101.001_2 \rightarrow 1.101001_2 \times 2^3$

- **Practice:** Convert $11.375_{10}$ to binary scientific notation

- **Practice:** Convert $1/5$ to binary
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
Floating Point Encoding

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

- 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 \( E \)
The Exponent Field

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

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

- **Practice:** To encode in biased notation, subtract the bias (add 127) then encode in unsigned:
  - Exp = 1 \(\rightarrow\) \texttt{E = 0b}
  - Exp = 127 \(\rightarrow\) \texttt{E = 0b}
  - Exp = -63 \(\rightarrow\) \texttt{E = 0b}
The Mantissa 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 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 \( 2^{\text{Exp}} \)
  - High values near \( M = 0b1...1 \) are close to \( 2^{\text{Exp}+1} \)
Precision and Accuracy

- **Precision** is a count of the number of bits in a computer word used to represent a value
  - Capability 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 significand (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 $-(2^{10}-1) = -1023$

- **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 =
  - *Special case:* E and M all zeros = 0
    - Two zeros! But at least 0x00000000 = 0 like integers

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

- Denormalized numbers
  - No leading 1
  - Careful! *Implicit exponent is \(-126\) (not \(-127\)) even though \(E = 0x00\)*

- Now what do the gaps look like?
  - Smallest norm: \(\pm 1.0...0_{\text{two}} \times 2^{-126} = \pm 2^{-126}\) \(\text{No gap}\)
  - Largest denorm: \(\pm 0.1...1_{\text{two}} \times 2^{-126} = \pm (2^{-126} - 2^{-149})\)
  - Smallest denorm: \(\pm 0.0...01_{\text{two}} \times 2^{-126} = \pm 2^{-149}\) \(\text{So much closer to 0}\)
Other Special Cases

- **E = 0xFF, M = 0:** ±∞
  - e.g., division by 0
  - Still work in comparisons!

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

- **Largest value (besides ∞)?**
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: \(1.1\ldots1_2\times2^{127} = 2^{128} - 2^{104}\)
## 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>
Distribution of Values

- What ranges are NOT representable?
  - Between largest norm and infinity: Overflow
  - Between zero and smallest denorm: Underflow
  - Between norm numbers?: Rounding

- Given a FP number, what’s the bit pattern of the next largest representable number?
  - What is this “step” when Exp = 0?
  - What is this “step” when Exp = 200?

- Distribution of values is denser toward zero
Peer Instruction Question

Let $FP[1,2) = \#$ of representable floats between 1 and 2
Let $FP[2,3) = \#$ of representable floats between 2 and 3

❖ Which of the following statements is true?
   - Vote at http://PollEv.com/justinh

(A) $FP[1,2) > FP[2,3)$
(B) $FP[1,2) == FP[2,3)$
(C) $FP[1,2) < FP[2,3)$
(D) It depends
Floating Point Operations: Basic Idea

Value = \((-1)^s \times \text{Mantissa} \times 2^{\text{Exponent}}\)

\[
\begin{array}{c|c|c}
S & E & M \\
\end{array}
\]

- \(x +_f y = \text{Round}(x + y)\)
- \(x \times_f y = \text{Round}(x \times y)\)

Basic idea for floating point operations:
- First, compute the exact result
- Then round the result to make it fit into desired precision:
  - Possibly over/underflow if exponent outside of range
  - Possibly drop least-significant bits of mantissa to fit into M bit vector
Floating Point Addition

- \((-1)^{S_1} \times \text{Man}_1 \times 2^{\text{Exp}_1} + (-1)^{S_2} \times \text{Man}_2 \times 2^{\text{Exp}_2}\)

  - Assume \(E_1 > E_2\)

- Exact Result: \((-1)^S \times \text{Man} \times 2^{\text{Exp}}\)
  - Sign \(S\), mantissa \(\text{Man}\):
    - Result of signed align & add
  - Exponent \(E\): \(E_1\)

- Adjustments:
  - If \(\text{Man} \geq 2\), shift \(\text{Man}\) right, increment \(E\)
  - If \(\text{Man} < 1\), shift \(\text{Man}\) left \(k\) positions, decrement \(E\) by \(k\)
  - Over/underflow if \(E\) out of range
  - Round \(\text{Man}\) to fit mantissa precision

Line up the binary points
Floating Point Multiplication

- \((-1)^{S_1} \times M_1 \times 2^{E_1} \times (-1)^{S_2} \times M_2 \times 2^{E_2}\)

- Exact Result: \((-1)^S \times M \times 2^E\)
  - Sign \(S\): \(s_1 \ ^\land \ s_2\)
  - Mantissa \(\text{Man}\): \(M_1 \times M_2\)
  - Exponent \(E\): \(E_1 + E_2\)

- Adjustments:
  - If \(\text{Man} \geq 2\), shift \(\text{Man}\) right, increment \(E\)
  - Over/underflow if \(E\) out of range
  - Round \(\text{Man}\) to fit mantissa precision
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>
More details for the curious. 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
- Distribution of Values
Visualization: Floating Point Encodings
Tiny Floating Point Example

- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit.
  - the next four bits are the exponent, with a bias of 7.
  - the last three bits are the frac

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity
## Dynamic Range (Positive Only)

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

- **Denormalized numbers**: Numbers closest to zero and largest denorm.
- **Normalized numbers**: Numbers closest to 1 below and closest to 1 above, as well as those closest to the largest norm (inf).
Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 23-1-1 = 3

- Notice how the distribution gets denser toward zero.

Diagram showing distribution of values with a 6-bit IEEE-like format: 1 bit for sign, 3 bits for exponent, and 2 bits for fraction. The distribution gets denser toward zero.
Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - $e = 3$ exponent bits
  - $f = 2$ fraction bits
  - Bias is 3

![Diagram showing distribution of values with 6-bit IEEE-like format]
## Interesting Numbers

<table>
<thead>
<tr>
<th>Description</th>
<th>exp</th>
<th>frac</th>
<th>Numeric Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>00...00</td>
<td>00...00</td>
<td>0.0</td>
</tr>
<tr>
<td>Smallest Pos. Denorm.</td>
<td>00...00</td>
<td>00...01</td>
<td>$2^{-23,52} \times 2^{-126,1022}$</td>
</tr>
<tr>
<td>Single</td>
<td>$1.4 \times 10^{-45}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$4.9 \times 10^{-324}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Largest Denormalized</td>
<td>00...00</td>
<td>11...11</td>
<td>$(1.0 - \varepsilon) \times 2^{-126,1022}$</td>
</tr>
<tr>
<td>Single</td>
<td>$1.18 \times 10^{-38}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$2.2 \times 10^{-308}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest Pos. Norm.</td>
<td>00...01</td>
<td>00...00</td>
<td>$1.0 \times 2^{-126,1022}$</td>
</tr>
<tr>
<td>Just larger than largest denormalized</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>One</td>
<td>01...11</td>
<td>00...00</td>
<td>1.0</td>
</tr>
<tr>
<td>Largest Normalized</td>
<td>11...10</td>
<td>11...11</td>
<td>$(2.0 - \varepsilon) \times 2^{127,1023}$</td>
</tr>
<tr>
<td>Single</td>
<td>$3.4 \times 10^{38}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>$1.8 \times 10^{308}$</td>
<td></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