Floating Point
CSE 351 Winter 2018

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi Parker, DeWilde, Emily Furst,
Sarah House, Waylon Huang, Vinny Palaniappan

http://xkcd.com/571/
Administrivia

- Lab 1 due Friday (1/19)
  - Submit `bits.c` and `pointer.c`
- Homework 2 out since 1/15, due 1/24
  - On Integers, Floating Point, and x86-64
Unsigned Multiplication in C

Operands: 
\(w\) bits

\[ u \cdot v \]

True Product: 
2\(w\) bits

Discard \(w\) bits: 
\(w\) bits

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:**
- $u \ll 3 = u \times 8$
- $u \ll 5 - u \ll 3 = u \times 24$
- 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. 6.02×10^{23})
  ▪ Very small numbers (e.g. 6.626×10^{-34})
  ▪ 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:

\[ xx \cdot yyyy \]

2^1  2^0  2^{-1}  2^{-2}  2^{-3}  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 $1/1,000,000,000$
  - 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`)

**Diagram:**
- **mantissa**: 1.01₂
- **exponent**: $2^{-1}$
- **radix (base)**: binary point
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: $1101.001_2 = 1.101001_2 \times 2^3$
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
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 bias of $2^{w-1}-1 = 127$
  ▪ Representable exponents roughly $\frac{1}{2}$ positive and $\frac{1}{2}$ negative
  ▪ Exponent 0 ($Exp = 0$) 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:
  ▪ $Exp = 1 \rightarrow E = 0b$
  ▪ $Exp = 127 \rightarrow E = 0b$
  ▪ $Exp = -63 \rightarrow E = 0b$
The Mantissa (Fraction) Field

\[(\pm) \times (1.M) \times 2^{(E-bias)}\]

- Note the implicit 1 in front of the M bit vector
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000 is read as 1.1_2 = 1.5_{10}, \textit{not} 0.1_2 = 0.5_{10}
  - Gives us an extra bit of \textit{precision}

- Mantissa “limits”
  - Low values near M = 0b0...0 are close to \(2^{\text{Exp}}\)
  - High values near M = 0b1...1 are close to \(2^{\text{Exp}+1}\)
What is the correct value encoded by the following floating point number?

0b 0 10000000 11000000000000000000000

A. + 0.75
B. + 1.5
C. + 2.75
D. + 3.5
E. We’re lost...
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 $2^{10} - 1 = 1023$
- **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:
  ▪ \( a = 1.0\ldots0 \times 2^{-126} = 2^{-126} \)
  ▪ \( b = 1.0\ldots01 \times 2^{-126} = 2^{-126} + 2^{-149} \)
  ▪ Normalization and implicit 1 are to blame
  ▪ *Special case:* E = 0, M ≠ 0 are denormalized numbers
Other Special Cases

❖ $E = 0xFF, \ M = 0$: ± ∞
  ▪ 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, −∞ ∞
  ▪ NaN propagates through computations
  ▪ Value of $M$ can be useful in debugging

❖ New largest value (besides ∞)?
  ▪ $E = 0xFF$ has now been taken!
  ▪ $E = 0xFE$ has largest: $1.1\ldots 1_2 \times 2^{127} = 2^{128} - 2^{104}$
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>
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...
Distribution of Values

❖ What ranges are NOT representable?
  ▪ Between largest norm and infinity  
  ▪ Between zero and smallest denorm  
  ▪ Between norm numbers?

❖ 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 = 100?

❖ Distribution of values is denser toward zero
Floating Point Operations: Basic Idea

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

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

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 \(\text{Exp}_1 > \text{Exp}_2\)

❖ 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 \(\text{Exp}\)
- If \(\text{Man} < 1\), shift \(\text{Man}\) left \(k\) positions, decrement \(\text{Exp}\) by \(k\)
- Over/underflow if \(\text{Exp}\) out of range
- Round \(\text{Man}\) to fit mantissa precision

Line up the binary points!
Floating Point Multiplication

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

- **Exact Result:** \(( -1)^S \times M \times 2^E\)
  - **Sign** \(S\): \(S_1 \times S_2\)
  - **Mantissa** \(M\): \(\text{Man}_1 \times \text{Man}_2\)
  - **Exponent** \(E\): \(\text{Exp}_1 + \text{Exp}_2\)

- **Adjustments:**
  - If \(\text{Man} \geq 2\), shift \(\text{Man}\) right, increment \(\text{Exp}\)
  - Over/underflow if \(\text{Exp}\) out of range
  - Round \(\text{Man}\) to fit mantissa precision
Mathematical Properties of FP Operations

- Exponent overflow yields $+\infty$ or $-\infty$
- Floats with value $+\infty$, $-\infty$, and NaN can be used in operations
  - Result usually still $+\infty$, $-\infty$, or NaN; but not always intuitive
- Floating point operations do not work like real math, due to rounding
  - Not associative: $(3.14+1e100) - 1e100 \neq 3.14 + (1e100 - 1e100)$
    
    \[
    \begin{array}{c}
    0 \\
    3.14
    \end{array}
    \]
  - Not distributive: $100 \times (0.1 + 0.2) \neq 100 \times 0.1 + 100 \times 0.2$
    
    \[
    \begin{array}{c}
    30.000000000000003553 \\
    30
    \end{array}
    \]
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing
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...
Floating Point in C

- C offers two (well, 3) levels of precision:
  - `float` 1.0f single precision (32-bit)
  - `double` 1.0 double precision (64-bit)
  - `long double` 1.0L (“double double” or quadruple) precision (64-128 bits)

- `#include <math.h>` to get `INFINITY` and `NAN` constants

- Equality (==) comparisons between floating point numbers are tricky, and often return unexpected results, so just avoid them!
Floating Point Conversions in C

❖ **Casting between int, float, and double changes the bit representation**

- **int → float**
  - May be rounded (not enough bits in mantissa: 23)
  - Overflow impossible

- **int or float → double**
  - Exact conversion (all 32-bit ints representable)

- **long → double**
  - Depends on word size (32-bit is exact, 64-bit may be rounded)

- **double or float → int**
  - Truncates fractional part (rounded toward zero)
  - “Not defined” when out of range or NaN: generally sets to $T_{\text{min}}$ (even if the value is a very big positive)
Number Representation Really Matters

- **1991**: Patriot missile targeting error
  - clock skew due to conversion from integer to floating point
- **1996**: Ariane 5 rocket exploded ($1 billion)
  - overflow converting 64-bit floating point to 16-bit integer
- **2000**: Y2K problem
  - limited (decimal) representation: overflow, wrap-around
- **2038**: Unix epoch rollover
  - Unix epoch = seconds since 12am, January 1, 1970
  - signed 32-bit integer representation rolls over to TMin in 2038
- **Other related bugs:**
  - 1982: Vancouver Stock Exchange 10% error in less than 2 years
  - 1994: Intel Pentium FDIV (floating point division) HW bug ($475 million)
  - 1997: USS Yorktown “smart” warship stranded: divide by zero
  - 1998: Mars Climate Orbiter crashed: unit mismatch ($193 million)
Floating Point Summary

❖ Floats also suffer from the fixed number of bits available to represent them
  ▪ Can get overflow/underflow
  ▪ “Gaps” produced in representable numbers means we can lose precision, unlike ints
    • Some “simple fractions” have no exact representation (e.g. 0.2)
    • “Every operation gets a slightly wrong result”

❖ Floating point arithmetic not associative or distributive
  ▪ Mathematically equivalent ways of writing an expression may compute different results

❖ Never test floating point values for equality!

❖ Careful when converting between ints and floats!
Floating Point Summary

- Converting between integral and floating point data types *does* change the bits
  - Floating point rounding is a HUGE issue!
    - Limited mantissa bits cause inaccurate representations
    - Floating point arithmetic is NOT associative or distributive
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: $\pm 1.0...0_{\text{two}} \times 2^{-126} = \pm 2^{-126}$
  - Smallest denorm: $\pm 0.0...01_{\text{two}} \times 2^{-126} = \pm 2^{-149}$
    - There is still a gap between zero and the smallest denormalized number

This is extra (non-testable) material
Floating Point and the Programmer

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0;
    float f2 = 0.0;
    int i;
    for (i = 0; i < 10; i++)
        f2 += 1.0/10.0;

    printf("0x%08x  0x%08x\n", *(int*)&f1, *(int*)&f2);
    printf("f1 = %10.9f\n", f1);
    printf("f2 = %10.9f\n\n", f2);

    f1 = 1E30;
    f2 = 1E-30;
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no");

    return 0;
}
```

```
$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119
f1 == f3? yes
```
An example that applies the IEEE Floating Point concepts to a smaller (8-bit) representation scheme.
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, ∞
## Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>M</th>
<th>Exp</th>
<th>Value</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>−6</td>
<td>0</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0001</td>
<td>−6</td>
<td>1/8*1/64 = 1/512</td>
<td>largest denorm</td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>0100</td>
<td>−6</td>
<td>2/8*1/64 = 2/512</td>
<td>smallest norm</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>1100</td>
<td>−6</td>
<td>6/8*1/64 = 6/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000</td>
<td>1110</td>
<td>−6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>0000</td>
<td>−6</td>
<td>8/8*1/64 = 8/512</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001</td>
<td>0001</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>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0110</td>
<td>1100</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</td>
<td>1111</td>
<td>−1</td>
<td>15/8*1/2 = 15/16</td>
<td>closest to 1 above</td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>0000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>0001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111</td>
<td>0100</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>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>1100</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110</td>
<td>1111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111</td>
<td>0000</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