Floating Point
CSE 351 Spring 2017

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis
Administrivia

- Lab 1 due next Friday (4/14)
  - Prelim submission (3+ of `bits.c`) due on Monday (4/10)
  - Bonus slides at the end of today’s lecture have relevant examples
- HW2 coming soon!
Unsigned Multiplication in C

Operands:
\[ w \text{ bits} \]

True Product:
\[ 2w \text{ bits} \]

Discard \( w \) bits:
\[ w \text{ bits} \]

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

<table>
<thead>
<tr>
<th>Operands: $w$ bits</th>
<th>$u \times 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u$</td>
<td>$0 \ldots 010 \ldots 00$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>True Product: $w + k$ bits</th>
<th>$u \times 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \times 2^k$</td>
<td>$0 \ldots 010 \ldots 00$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Discard $k$ bits: $w$ bits</th>
<th>$u \times 2^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u \times 2^k$</td>
<td>$0 \ldots 010 \ldots 00$</td>
</tr>
</tbody>
</table>

- **Examples:**
  - $u \ll 3 \quad =\quad u \times 8$
  - $u \ll 5 \ - \ u \ll 3 \quad =\quad u \times 24$

- Most machines shift and add faster than multiply
  - *Compiler generates this code automatically*
Number Representation Revisited

- What can we represent so far?
  - 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)

Floating Point
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 Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “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!
Representation of Fractions

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

  Example 6-bit representation: 

  \[
  \begin{array}{c}
  2^1 \\
  2^0 \\
  2^{-1} \\
  2^{-2} \\
  2^{-3} \\
  2^{-4}
  \end{array}
  \]

- **Example:** \(10.1010_2 = 1\times2^1 + 1\times2^{-1} + 1\times2^{-3} = 2.625_{10}\)

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

- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number: \[ \sum_{k=-j}^{i} b_k \cdot 2^k \]
Fractional Binary Numbers

- **Value**
  - 5 and 3/4
  - 2 and 7/8
  - 47/64

- **Representation**
  - 101.11₂
  - 10.111₂
  - 0.101111₂

- **Observations**
  - Shift left = multiply by power of 2
  - Shift right = divide by power of 2
  - Numbers of the form 0.111111...₂ are just below 1.0
    - 1/2 + 1/4 + 1/8 + ... + 1/2ⁱ + ... \(\Rightarrow\) 1.0
    - Use notation 1.0 – \(\varepsilon\)
Limits of Representation

- Limitations:
  - Even given an arbitrary number of bits, can only exactly represent numbers of the form $x \times 2^y$ (y can be negative)
  - Other rational numbers have repeating bit representations

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary Representation:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1/3 = 0.333333..._{10}$</td>
<td>$0.01010101[01]..._2$</td>
</tr>
<tr>
<td>$1/5 = 0.001100110011[0011]..._2$</td>
<td></td>
</tr>
<tr>
<td>$1/10 = 0.0001100110011[0011]..._2$</td>
<td></td>
</tr>
</tbody>
</table>
Fixed Point Representation

- Implied binary point. Two example schemes:
  
  #1: the binary point is between bits 2 and 3
  \[ b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ [. \] \ b_2 \ b_1 \ b_0 \]

  #2: the binary point is between bits 4 and 5
  \[ b_7 \ b_6 \ b_5 \ [. \] \ b_4 \ b_3 \ b_2 \ b_1 \ b_0 \]

- Wherever we put the binary point, with fixed point representations there is a trade off between the amount of range and precision we have

- Fixed point = fixed range and fixed precision
  
  - range: difference between largest and smallest numbers possible
  - precision: smallest possible difference between any two numbers

- Hard to pick how much you need of each!
Floating Point Representation

- Analogous to scientific notation
  - In Decimal:
    - Not 12000000, but $1.2 \times 10^7$ In C: 1.2e7
    - Not 0.0000012, but $1.2 \times 10^{-6}$ In C: 1.2e-6
  - In Binary:
    - Not 11000.000, but $1.1 \times 2^4$
    - Not 0.000101, but $1.01 \times 2^{-4}$
- We have to divvy up the bits we have (e.g., 32) among:
  - the sign (1 bit)
  - the significand
  - the exponent
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

- **Practice:** Convert \(1/5\) to binary
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 Representation

- Numerical form:

\[ V_{10} = (-1)^s \times M \times 2^E \]

- Sign bit \( s \) determines whether number is negative or positive
- Significand (mantissa) \( M \) normally a fractional value in range \([1.0,2.0)\)
- Exponent \( E \) weights value by a (possibly negative) power of two
Floating Point Representation

- Numerical form:
  \[ V_{10} = (-1)^s \times M \times 2^E \]
  - Sign bit \( s \) determines whether number is negative or positive
  - Significand (mantissa) \( M \) normally a fractional value in range \([1.0, 2.0)\)
  - Exponent \( E \) weights value by a (possibly negative) power of two

- Representation in memory:
  - MSB \( s \) is sign bit \( s \)
  - \( \text{exp} \) field encodes \( E \) (but is \textit{not equal} to \( E \))
  - \( \text{frac} \) field encodes \( M \) (but is \textit{not equal} to \( M \))
Precisions

- **Single precision:** 32 bits
  
  1 bit | 8 bits | 23 bits

- **Double precision:** 64 bits
  
  1 bit | 11 bits | 52 bits

- **Finite representation means not all values can be represented exactly. Some will be approximated.**
Normalization and Special Values

\[ V = (-1)^s \cdot M \cdot 2^E \]

- "Normalized" = \( M \) has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 0.011 \( \times 2^5 \) and 1.1 \( \times 2^3 \) represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it

- How do we represent 0.0? Or special or undefined values like 1.0/0.0?
Normalizes and Special Values

\[ V = (-1)^{S} \times M \times 2^{E} \]

- "Normalized" = \( M \) has the form 1.xxxxx
  - As in scientific notation, but in binary
  - 0.011 x 2^5 and 1.1 x 2^3 represent the same number, but the latter makes better use of the available bits
  - Since we know the mantissa starts with a 1, we don't bother to store it.

- Special values:
  - \textbf{zero}:
    \[\text{exp} = 00\ldots0 \quad \text{frac} = 00\ldots0\]
  - \(+\infty, -\infty\):
    \[\text{exp} = 11\ldots1 \quad \text{frac} = 00\ldots0\]
    \[1.0/0.0 = -1.0/-0.0 = +\infty, \quad 1.0/-0.0 = -1.0/0.0 = -\infty\]
  - \textbf{NaN} ("Not a Number"): \(\text{exp} = 11\ldots1 \quad \text{frac} \neq 00\ldots0\)
    Results from operations with undefined result: \(\sqrt{-1}, \infty - \infty, \infty \times 0\), etc.
  - \textbf{Note}: \(\text{exp}=11\ldots1\) and \(\text{exp}=00\ldots0\) are reserved, limiting \text{exp} range...
Normalized Values

\[ V = (-1)^S \times M \times 2^E \]

- **Condition:** \( \exp \neq 000\ldots0 \) and \( \exp \neq 111\ldots1 \)
- **Exponent coded as biased value:** \( E = \exp - \text{Bias} \)
  - \( \exp \) is an unsigned value ranging from 1 to \( 2^{k-2} \) (\( k \) == # bits in \( \exp \))
  - \( \text{Bias} = 2^{k-1} - 1 \)
    - Single precision: 127 (so \( \exp \): 1...254, \( E \): -126...127)
    - Double precision: 1023 (so \( \exp \): 1...2046, \( E \): -1022...1023)
  - These enable negative values for \( E \), for representing very small values
- **Significand coded with implied leading 1:** \( M = 1.\ldots x \)
  - \( \ldots x \): the \( n \) bits of \( \text{frac} \)
  - Minimum when \( 000\ldots0 \) (\( M = 1.0 \))
  - Maximum when \( 111\ldots1 \) (\( M = 2.0 - \varepsilon \))
  - Get extra leading bit for “free”
Normalized Encoding Example

\[ V = (-1)^S \times M \times 2^E \]

網路:

- **Value:** `float f = 12345.0;`
  - `12345_{10} = 11000000111001_{2}`
  - `= 1.1000000111001_{2} \times 2^{13} \text{ (normalized form)}`

- **Significand:**
  - `M = \underline{1.1000000111001}_{2}`
  - `\text{frac} = \underline{1000000111001000000000000}_{2}`

- **Exponent:** `E = \text{exp} - \text{Bias}, \text{ so exp} = E + \text{Bias}`
  - `E = 13`
  - `Bias = 127`
  - `\text{exp} = 140 = \underline{10001100}_{2}`

- **Result:**
  - `\begin{array}{c|c|c}
    s & \text{exp} & \text{frac} \\
    \hline
    0 & \underline{10001100} & \underline{1000000011100100000000000}_{2}
  \end{array}`
Question

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

A. $+0.75$
B. $+1.5$
C. $+2.75$
D. $+3.5$
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 Operations

- Unlike the representation for integers, the representation for floating-point numbers is **not exact**
Floating Point Operations: Basic Idea

\[ V = (-1)^S \times M \times 2^E \]

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

- Basic idea for floating point operations:
  - First, compute the exact result
  - Then, round the result to make it fit into desired precision:
    - Possibly overflow if exponent too large
    - Possibly drop least-significant bits of significand to fit into \( \text{frac} \)
Floating Point Addition

\[(–1)^s_1 M_1 \ 2^{E_1} \ + \ (–1)^s_2 M_2 \ 2^{E_2}\]

Assume \(E_1 > E_2\)

- **Exact Result:** \((–1)^s M \ 2^E\)
  - Sign \(s\), significand \(M\):
    - Result of signed align & add
  - Exponent \(E\): \(E_1\)

- **Fixing**
  - If \(M \geq 2\), shift \(M\) right, increment \(E\)
  - if \(M < 1\), shift \(M\) left \(k\) positions, decrement \(E\) by \(k\)
  - Overflow if \(E\) out of range
  - Round \(M\) to fit \(\text{frac}\) precision

Line up the binary points
Floating Point Multiplication

\[ (-1)^{s_1} M_1 \ 2^{E_1} \ * \ (-1)^{s_2} M_2 \ 2^{E_2} \]

- **Exact Result:** \((-1)^{s} M \ 2^{E}\)
  - Sign \(s\): \(s_1 \ ^{\land} \ s_2\)
  - Significand \(M\): \(M_1 \ * \ M_2\)
  - Exponent \(E\): \(E_1 + E_2\)

- **Fixing**
  - If \(M \geq 2\), shift \(M\) right, increment \(E\)
  - If \(E\) out of range, overflow
  - Round \(M\) to fit \textit{frac} precision
Mathematical Properties of FP Operations

- Exponent overflow yields $\pm\infty$
- Floats with value $\pm\infty$, $-\infty$, and NaN can be used in operations
  - Result usually still $\pm\infty$, $-\infty$, or NaN; but not always intuitive
- Floating point operations do not work like real math, due to rounding!!
  - Not associative: $(3.14+1e100)-1e100 \neq 3.14+(1e100-1e100)$
    - 0
    - 3.14
  - Not distributive: $100\times(0.1+0.2) \neq 100\times0.1+100\times0.2$
    - 30.000000000000003553
    - 30
  - 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!

- Instead use `abs(f1 – f2) < 2^{-20}` or some other threshold
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; 52-bit frac)
  - **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 Tmin (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)
# include <stdio.h>

int main(int argc, char* argv[]) {
    int a = 33554435;
    printf("a = %d
(float) a = %f
\n", a, (float) a);

    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;
}
Floating Point Summary

- As with integers, floats suffer from the fixed number of bits available to represent them
  - Can get overflow/underflow, just like ints
  - Some “simple fractions” have no exact representation (e.g., 0.2)
  - Can also lose precision, unlike ints
    - “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 Puzzles

For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

```c
int x = ...;
float f = ...;
double d = ...;
double d2 = ...;
```

Assume neither `d` nor `f` is NaN

- `x == (int)(float) x`
- `x == (int)(double) x`
- `f == (float)(double) f`
- `d == (double)(float) d`
- `f == -(f)`
- `2/3 == 2/3.0`
- `(d+d2) - d == d2`