Floating Point II
CSE 351 Autumn 2022

Instructor:
Justin Hsia

Teaching Assistants:
Angela Xu
Arjun Narendra
Armin Magness
Assaf Vayner
Carrie Hu
Clare Edmonds
David Dai
Dominick Ta
Effie Zheng
James Froelich
Jenny Peng
Kristina Lansang
Paul Stevans
Renee Ruan
Vincent Xiao

http://www.smbc-comics.com/?id=2999
Relevant Course Information

- hw6 due Friday, hw7 due Monday

- Lab 1a: last chance to submit is tonight @ 11:59 pm
  - One submission per partnership
  - Make sure you check the Gradescope autograder output!
  - Grades hopefully released by end of Sunday (10/16)

- Lab 1b due Monday (10/17)
  - Submit aisle_manager.c, store_client.c, and lab1Bsynthesis.txt

- Section tomorrow on Integers and Floating Point
Getting Help with 351

- Lecture recordings, readings, inked slides, section presentation recordings, worksheet solutions

- Form a study group!
  - Good for everything but labs, which should be done in pairs
  - Communicate regularly, use the class terminology, ask and answer each others’ questions, show up to OH together

- Attend office hours
  - Can also chat with other students—help each other learn!

- Post on Ed Discussion

- Request a 1-on-1 meeting
  - Available on a limited basis for special circumstances
Reading Review

❖ Terminology:
  ▪ Special cases
    • Denormalized numbers
    • $\pm\infty$
    • Not-a-Number (NaN)
  ▪ Limits of representation
    • Overflow
    • Underflow
    • Rounding

❖ Questions from the Reading?
Review Questions

❖ What is the value of the following floats?
  ▪ 0x0000000000
  ▪ 0xFF800000

❖ For the following code, what is the smallest value of \( n \) that will encounter a limit of representation?

```c
float f = 1.0;  // 2^0
for (int i = 0; i < n; ++i)
  f *= 1024;  // 1024 = 2^10
printf("f = %f\n", f);
```
# Floating Point Encoding Summary (Review)

<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
<th>Interpretation</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>
Special Cases

❖ But wait... what happened to zero?
   - *Special case:* $E$ and $M$ all zeros = 0
   - Two zeros! But at least $0x00000000 = 0$ like integers

❖ $E = 0xFF, M = 0$: $\pm \infty$
   - *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 Representation Limits (Review)

- New largest value (besides $\infty$)?
  - $E = 0xFF$ has now been taken!
  - $E = 0xFE$ has largest: $1.1...1_2 \times 2^{127} = 2^{128} - 2^{104}$

- New numbers closest to 0:
  - $E = 0x00$ taken; next smallest is $E = 0x01$
  - $a = 1.0...00_2 \times 2^{-126} = 2^{-126}$
  - $b = 1.0...01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - Normalization and implicit 1 are to blame
  - *Special case:* $E = 0$, $M \neq 0$ are denormalized numbers
Denorm Numbers (Review)

- 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
Floating Point Decoding Flow Chart

FP Bits \rightarrow \text{What is the value of } E? \rightarrow \text{all 0's} \rightarrow (−1)^S \times \infty
\rightarrow \text{all 1's} \rightarrow \text{What is the value of } M? \rightarrow \text{all 0's} \rightarrow (−1)^S \times 0. M \times 2^{1−bias}
\rightarrow \text{anything else} \rightarrow (−1)^S \times 1. M \times 2^{E−bias}

\text{NaN} \rightarrow \text{anything else}
Floating Point Topics

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

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

❖ We will use the following 8-bit floating point representation to illustrate some key points:

Assume that it has the same properties as IEEE floating point:

- bias =
- encoding of $-0 =$
- encoding of $+\infty =$
- encoding of the largest (+) normalized # =
- encoding of the smallest (+) normalized # =
Distribution of Values (Review)

- What ranges are NOT representable?
  - Between largest norm and infinity: **Overflow** (Exp too large)
  - Between zero and smallest denorm: **Underflow** (Exp too small)
  - Between norm numbers?

- Given a FP number, what’s 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

- Diagram:
  - Denormalized
  - Normalized
  - Infinity
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 the specified precision (width of M)
  - Possibly over/underflow if exponent outside of range
Mathematical Properties of FP Operations

- **Overflow** yields $\pm \infty$ and **underflow** yields 0
- Floats with value $\pm \infty$ and **NaN** can be used in operations
  - Result usually still $\pm \infty$ or NaN, but not always intuitive
- Floating point operations do not work like real math, due to **rounding**
  - Not associative: 
    \[
    (3.14+1e100)-1e100 \neq 3.14+(1e100-1e100) \\
    0 \neq 3.14
    \]
  - Not distributive: 
    \[
    100\times(0.1+0.2) \neq 100\times0.1+100\times0.2 \\
    30.000000000000003553 \neq 30
    \]
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing
Floating Point Rounding

❖ The IEEE 754 standard actually specifies different rounding modes:

- Round to nearest, ties to nearest even digit
- Round toward $+\infty$ (round up)
- Round toward $-\infty$ (round down)
- Round toward 0 (truncation)

❖ In our tiny example:
- Man = 1.001 01 rounded to $M = 0b001$
- Man = 1.001 11 rounded to $M = 0b010$
- Man = 1.001 10 rounded to $M = 0b010$
- Man = 1.000 10 rounded to $M = 0b000$
Floating Point in C

❖ Two common levels of precision:

- `float 1.0f` single precision (32-bit)
- `double 1.0` double precision (64-bit)

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

❖ `#include <float.h>` for additional 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 (tries to preserve the value)
  - int → float
    - May be rounded (not enough bits in mantissa: 23)
    - Overflow impossible
  - int or float → double
    - Exact conversion (all 32-bit ints are 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 TMin (even if the value is a very big positive)
Casting Example

- We execute the following code in C. How are \( i \) and \( f \) represented in hex?

```c
int i = 384;    // 2^8 + 2^7
float f = (float) i;
```
Discussion Questions

❖ How do you feel about floating point?
   ▪ Do you feel like the limitations are acceptable?

   ▪ Does this affect the way you’ll think about non-integer arithmetic in the future?

   ▪ Are there any changes or different encoding schemes that you think would be an improvement?
More on Floating Point History

❖ Early days
  ▪ First design with floating-point arithmetic in 1914 by Leonardo Torres y Quevedo
  ▪ Implementations started in 1940 by Konrad Zuse, but with differing field lengths (usually not summing to 32 bits) and different subsets of the special cases

❖ IEEE 754 standard created in 1985
  ▪ Primary architect was William Kahan, who won a Turing Award for this work
  ▪ Standardized bit encoding, well-defined behavior for all arithmetic operations
Floating Point in the “Wild”

- 3 formats from IEEE 754 standard widely used in computer hardware and languages
  - In C, called float, double, long double
- Common applications:
  - 3D graphics: textures, rendering, rotation, translation
  - “Big Data”: scientific computing at scale, machine learning
- Non-standard formats in domain-specific areas:
  - **Bfloat16**: training ML models; range more valuable than precision
  - **TensorFloat-32**: Nvidia-specific hardware for Tensor Core GPUs

<table>
<thead>
<tr>
<th>Type</th>
<th>S bits</th>
<th>E bits</th>
<th>M bits</th>
<th>Total bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Half-precision</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>Bfloat16</td>
<td>1</td>
<td>8</td>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>TensorFloat-32</td>
<td>1</td>
<td>8</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>Single-precision</td>
<td>1</td>
<td>8</td>
<td>23</td>
<td>32</td>
</tr>
</tbody>
</table>
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!
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)
Summary

Floating point encoding has many limitations
- Overflow, underflow, rounding
- Rounding is a HUGE issue due to limited mantissa bits and gaps that are scaled by the value of the exponent
- Floating point arithmetic is NOT associative or distributive

Converting between integral and floating point data types *does* change the bits

<table>
<thead>
<tr>
<th>E</th>
<th>M</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b0...0</td>
<td>anything</td>
<td>± denorm num (including 0)</td>
</tr>
<tr>
<td>anything else</td>
<td>anything</td>
<td>± norm num</td>
</tr>
<tr>
<td>0b1...1</td>
<td>0</td>
<td>± ∞</td>
</tr>
<tr>
<td>0b1...1</td>
<td>non-zero</td>
<td>NaN</td>
</tr>
</tbody>
</table>