Floating Point II
CSE 351 Autumn 2020

Instructor:
Justin Hsia

Teaching Assistants:
Aman Mohammed
Ami Oka
Callum Walker
Cosmo Wang
Hang Do
Jim Limprasert
Joy Dang
Julia Wang
Kaelin Laundry
Kyrie Dowling
Mariam Mayanja
Shawn Stanley
Yan Zhe Ong

http://www.smbc-comics.com/?id=2999
Administrivia

- hw6 due Friday, hw7 due Monday
- Lab 1a: last chance to submit is tonight @ 11:59 pm
  - Make sure you check the Gradescope autograder output!
  - Grades hopefully released by end of Sunday (10/18)
- Lab 1b due Monday (10/19)
  - Submit aisle_manager.c, store_client.c, and lab1Breflect.txt
- Section tomorrow on Integers and Floating Point
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?
  - 0x00000000
  - 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

<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 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: ± ∞
  - *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
New Representation Limits

- 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 $E = 0x01$
  - $a = 1.0...0_2 \times 2^{-126} = 2^{-126}$
  - $b = 1.0...01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - 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: $\pm 1.0...0_{\text{two}} \times 2^{-126} = \pm 2^{-126}$
  - 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

So much closer to 0
Floating Point Interpretation Flow Chart

FP Bits

What is the value of E?

all 1's

all 0's

What is the value of M?

all 0's

anything else

anything else

$(-1)^S \times \infty$

NaN

$(-1)^S \times 0. M \times 2^{1-bias}$

$(-1)^S \times 1. M \times 2^{E-bias}$

= special case
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...
Tiny Floating Point Representation

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

  
  ![Floating Point Representation Diagram](image)

  
  - 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

- 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 \( \text{Exp} = 0 \)?
  - What is this “step” when \( \text{Exp} = 100 \)?

- Distribution of values is denser toward zero
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 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
  - Not distributive: $100 \times (0.1 + 0.2) \neq 100 \times 0.1 + 100 \times 0.2$
    - 30.000000000000003553
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing
Floating Point Encoding Flow Chart

Value \( v \) to encode

Is \( v \) not a number?

Is \( |v| \), when rounded, \( \geq \) \( F_{\text{Over}} \)?

Is \( |v| \), when rounded, \( < \) \( F_{\text{Under}} \)?

Is \( |v| \), when rounded, \( < \) \( F_{\text{Denorm}} \)?

Yes

No

\( \pm \infty \)

\( E = \) all 1’s

\( M = \) all 0’s

\( \pm 0 \)

\( E = \) all 0’s

\( M = \) all 0’s

\( \text{NaN} \)

\( E = \) all 1’s

\( M \neq \) all 0’s

Denormed

\( E = \) all 0’s

0, \( M = \) Man

Normed

\( E = \) Exp + bias

1, \( M = \) Man

= special case
Limits of Interest

- The following thresholds will help give you a sense of when certain outcomes come into play, but don’t worry about the specifics:

  - **FOver** = $2^{\text{bias}+1} = 2^8$
    - This is just larger than the largest representable normalized number

  - **FDenorm** = $2^{1-\text{bias}} = 2^{-6}$
    - This is the smallest representable normalized number

  - **FUnder** = $2^{1-\text{bias}-m} = 2^{-9}$
    - $m$ is the width of the mantissa field
    - This is the smallest representable denormalized number

This is extra (non-testable) material
Polling Question 1

- Using our 8-bit representation, what value gets stored when we try to encode $2.625 = 2^1 + 2^{-1} + 2^{-3}$?

  - A. + 2.5
  - B. + 2.625
  - C. + 2.75
  - D. + 3.25
  - E. We’re lost...
Polling Question 2

- Using our **8-bit** representation, what value gets stored when we try to encode $384 = 2^8 + 2^7$?

A. + 256  
B. + 384  
C. + ∞  
D. NaN  
E. We’re lost...
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

  - `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)
Challenge Question

- We execute the following code in C. How many bytes are the same (value and position) between `i` and `f`?

  - No voting

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

A. 0 bytes
B. 1 byte
C. 2 bytes
D. 3 bytes
E. We’re lost...
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
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 E M</th>
<th>Exp</th>
<th>Value</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Denormalized numbers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0000 000</td>
<td>-6</td>
<td>0</td>
<td>closest to zero</td>
</tr>
<tr>
<td>0 0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td></td>
</tr>
<tr>
<td>0 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>
</tr>
<tr>
<td>0 0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
<td>largest denorm</td>
</tr>
<tr>
<td>0 0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
</tr>
<tr>
<td><strong>Normalized numbers</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td>smallest norm</td>
</tr>
<tr>
<td>0 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>
</tr>
<tr>
<td>0 0110 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td>closest to 1 below</td>
</tr>
<tr>
<td>0 0110 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
</tr>
<tr>
<td>0 0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td>closest to 1 above</td>
</tr>
<tr>
<td>0 0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
</tr>
<tr>
<td>0 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>
</tr>
<tr>
<td>0 1110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td>largest norm</td>
</tr>
<tr>
<td>0 1110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
</tr>
<tr>
<td>0 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