Floating Point II
CSE 351 Autumn 2019

Instructor: Justin Hsia

Teaching Assistants:
Andrew Hu
Diya Joy
Maurice Montag
Suraj Jagadeesh

Antonio Castelli
Ivy Yu
Melissa Birchfield

Cosmo Wang
Kaelin Laundry
Millicent Li

http://xkcd.com/899/
Administrivia

- hw6 due Friday, hw7 due Monday
- Lab 1a grades hopefully released by end of Sunday (10/13)
- Lab 1b due Monday (10/14)
  - Submit bits.c and lab1Breflect.txt
- Section tomorrow on Integers and Floating Point
Other Special Cases

- \( E = 0xFF, M = 0: \pm \infty \)
  - 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 (tells you cause of NaN)

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

<table>
<thead>
<tr>
<th>E</th>
<th>M</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 Interpretation Flow Chart

- **FP Bits**
  - What is the value of $E$?
    - all 1’s
      - What is the value of $M$?
        - all 0’s
          - all 0’s
            - $(-1)^S \times \infty$
          - anything else
            - NaN
        - anything else
          - normalized
            - $(-1)^S \times 1. M \times 2^{E-bias}$
          - anything else
            - denormalized
              - $(-1)^S \times 0. M \times 2^{1-bias}$

= 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:

  ![S E M diagram](image)

  - Encoding of $-0 = 0b \ 1 \ 0000 \ 000 = 0x80$
  - Encoding of $+\infty = 0b \ 0 \ 1111 \ 000 = 0x78$
  - Encoding of the largest (+) normalized # = $0b \ 0 \ 1110 \ 111 = 0x77$
  - Encoding of the smallest (+) normalized # = $0b \ 0 \ 0001 \ 000 = 0x08$
Distribution of Values

- What ranges are NOT representable?
  - Between largest norm and infinity \textbf{Overflow} (Exp too large)
  - Between zero and smallest denorm \textbf{Underflow} (Exp too small)
  - Between norm numbers? \textbf{Rounding}

- Given a FP number, what’s the bit pattern of the next largest representable number?
  - What is this “step” when Exp = 0? \(2^{-23}\)
  - What is this “step” when Exp = 100? \(2^{77}\)

- Distribution of values is denser toward zero

\[ \text{overflow} \quad \text{underflow} \quad \text{rounding} \]

- **Denormalized**  **Normalized**  **Infinity**
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 $M = 0b001$ (down)
  - Man = 1.001/11 rounded to $M = 0b010$ (up)
  - Man = 1.001/10 rounded to $M = 0b010$
  - Man = 1.000/10 rounded to $M = 0b000$ (down)
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 \textit{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: $100*(0.1+0.2) \neq 100*0.1+100*0.2$
    - $30 \neq 30.00000000000003553$
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing
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
Floating Point Encoding Flow Chart

Value \( v \) to encode → Is \( v \) not a number? → No

Yes → NaN
\( E = \) all 1’s
\( M \neq \) all 0’s

Is \( |v| \), when rounded, \( \geq \) FOver? → Yes

\( \pm \infty \)
\( E = \) all 1’s
\( M = \) all 0’s

No → Is \( |v| \), when rounded, \( \leq \) FDENORM? → No

Yes → Denormalized
\( E = \) all 0’s
\( 0.M = \) Man

No → Normed
\( E = \) Exp + bias
\( 1.M = \) Man

= special case
Example Question

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

\[
384 = 2^8 (1 + 2^{-1}) = 2^8 \times 1.1_2
\]

\( S = 0 \)\
\( E = \text{Exp + bias} = 8 + 7 = 15 = 0b1111 \)

- No voting

A. + 256
B. + 384
C. + \( \infty \)
D. NaN
E. We’re lost...

\[ +\infty \leftrightarrow 0b 0 1111 000 \text{ instead} \]
Polling Question

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

\[
\begin{align*}
S &= 0 \\
E &= \text{Exp} + \text{bias} \\
&= 1 + 7 = 8 \\
&= 0b\ 1000 \\
M &= 0b\ 0101/1
\end{align*}
\]


A. + 2.5
B. + 2.625
C. + 2.75
D. + 3.25
E. We’re lost...

Stored as: 0b 0 1000 010 = 2.5
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 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
  - `<float.h>` for additional 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}`
    - some arbitrary 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)
  - 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)
Polling Question

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

  - Vote at http://PollEv.com/justinh

```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...

\( i \) stored as 0x 00 00 01 80
\( f \) stored as 0x 43 C0 00 00
#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;
}
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 \texttt{ints}
    - Some “simple fractions” have no exact representation (\textit{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

- \textbf{Never} test floating point values for equality!
- \textbf{Careful} when converting between \texttt{ints} and \texttt{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>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>