Floating Point II
CSE 351 Spring 2020

Instructor: Ruth Anderson

Teaching Assistants:
Alex Olshanskyy
Connie Wang
Eddy (Tianyi) Zhou
Jonathan Chen
Millicent Li

Callum Walker
Diya Joy
Eric Fan
Joseph Schafer
Porter Jones

Chin Yeoh
Edan Sneh
Jeffery Tian
Melissa Birchfield
Rehaan Bhimani

http://xkcd.com/899/
Adminstrivia

- Lab 1a due TONIGHT (4/13) at 11:59 pm
  - Submit `pointer.c` and `lab1Areflect.txt`
- hw6 due Wednesday – 11am
- Lab 1b due Monday (4/20)
  - Submit `bits.c` and `lab1Breflect.txt`
- You must log on with your @uw google account to access!!
  - Google doc for 11:30 Lecture: [https://tinyurl.com/351-04-13A](https://tinyurl.com/351-04-13A)
- Week 2 Feedback Survey
  - [https://catalyst.uw.edu/webq/survey/rea2000/388285](https://catalyst.uw.edu/webq/survey/rea2000/388285)
Other Special Cases

- **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 (tells you cause of NaN)

- New largest value (besides ∞)?
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: \(1.1\ldots1_2 \times 2^{127} = 2^{128} - 2^{104}\)
# 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 0's
    - (−1)^S × ∞
  - all 1's
    - NaN
  - anything else
    - (−1)^S × 0. M × 2^{1−bias}
- anything else
  - (−1)^S × 1. M × 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:

  ![Tiny Floating Point Representation Diagram]

- Assume that it has the same properties as IEEE floating point:
  - bias = $2^{\omega-1} - 1 = 2^{4-1} - 1 = 7$
  - encoding of $-0 = 0b\ 1\ 0000\ 000 = 0\times80$
  - encoding of $+\infty = 0b\ 0\ 1111\ 000 = 0\times78\ 1\ 1111_2 \times 2^{14-7}$
  - encoding of the largest (+) normalized # = $0b\ 0\ 1110\ 111 = 0\times77$
  - encoding of the smallest (+) normalized # = $0b\ 0\ 0001\ 000 = 0\times08\ 1\ 00001_2 \times 2^{-17}$
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 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

![Distribution Diagram]

Legend:
- 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 = 1.00101 rounded to $M = 0b001$ (down)
  - Man = 1.00111 rounded to $M = 0b010$ (up)
  - Man = 1.00110 rounded to $M = 0b010$
  - Man = 1.00010 rounded to $M = 0b000$ (down)
Floating Point Operations: Basic Idea

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

\[
\begin{array}{c|c|c}
S & E & M \\
\end{array}
\]

- \(x +_f y = \text{Round}(x + y)\)
- \(x *_f y = \text{Round}(x * y)\)

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: $(3.14+1e100)-1e100 \neq 3.14+(1e100-1e100)$
  - Not distributive: $100*(0.1+0.2) \neq 100*0.1+100*0.2$
  - Not cumulative
    - Repeatedly adding a very small number to a large one may do nothing
Aside: 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

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

\( \pm \infty \)

E = all 1’s
M = all 0’s

No

Is \( |v| \), when rounded, \( < FUnder? \) Yes

\( \pm 0 \)

E = all 0’s
M = all 0’s

Yes

Is \( |v| \), when rounded, \( < FDenorm\)? No

Denormed
E = all 0’s
0.M = Man

Yes

Normed
E = Exp + bias
1.M = Man

\( \Box \) = special case
Example Question [FP II - a]

Using our 8-bit representation, what value gets stored when we try to encode $384 = 2^8 + 2^7$? $= 2^8(1 + 2^{-1})$

\[ = 2^8 \times 1.1_2 \]

\[ S = 0 \]

\[ E = \text{Exp} + \text{bias} = 8 + 7 = 15 \]

\[ = 0b1111 \]

\[ \text{this falls outside of the normalized exponent range!} \]

No voting

A. + 256

B. + 384

C. + ∞ **(Correct Answer)**

D. NaN

E. We’re lost...

**Note:**

This number is too large, so we store $+\infty \leftrightarrow 0b01111000$ instead.
Polling Question [FP II - b]

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

\[
\begin{align*}
S &= 0 \\
E &= \text{Exp} + \text{bias} \\
&= 1 + 7 = 8 \\
&= \text{Ob 1000} \\
M &= \text{Ob 010101}_2 \\
\text{stored as: Ob 0 1000 010 = 2.5}
\end{align*}
\]

- Vote at [http://pollev.com/rea](http://pollev.com/rea)

A. + 2.5
B. + 2.625
C. + 2.75
D. + 3.25
E. We’re lost…
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 \( \text{abs}(f_1 - f_2) < 2^{-20} \) 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 [FP II - c]

- 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/rea](http://pollev.com/rea)

```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...

\[\begin{align*}
\text{i stored as } & \quad 0x \quad 00 \quad 00 \quad 01 \quad 80 \\
\text{f stored as } & \quad 0x \quad 43 \quad C0 \quad 00 \quad 00
\end{align*}\]
Floating Point and the Programmer

#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0f; // specify float constant
    float f2 = 0.0f;
    int i;
    for (i = 0; i < 10; i++)
        f2 += 1.0/10.0;
    f2 should == 10×1/10 = 1
    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; 10^{30}
    f2 = 1E-30; 10^{-30}
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );
    10^{30} == 10^{30} + 10^{-30}
    return 0;
}

$ ./a.out
0x3f800000 0x3f800001
f1 = 1.000000000
f2 = 1.0000000119
f1 == f3? yes
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