Floating Point II
CSE 351 Autumn 2019

Instructor: Justin Hsia
Teaching Assistants: Andrew Hu, Antonio Castelli, Cosmo Wang
Diya Joy, Ivy Yu, Kaelin Laundry
Maurice Montag, Melissa Birchfield, 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

- $E = 0xFF, M = 0$: ± $\infty$
  - 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 largest value (besides $\infty$)?
  - $E = 0xFF$ has now been taken!
  - $E = 0xFE$ has largest: $1.1...1_2 \times 2^{127} = 2^{128} - 2^{104}$
# 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 $E$?
- What is the value of $M$?
- all 0's
  - $(-1)^S \times \infty$
- all 1's
  - anything else
    - NaN
- anything else
  - $(-1)^S \times 0. M \times 2^{1-bias}$
  - $(-1)^S \times 1. M \times 2^{E-bias}$

= 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 bit pattern of 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 $M = 0b010$
Floating Point Operations: Basic Idea

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

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

- 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:**
    \[
    100\times(0.1+0.2) \neq 100\times0.1+100\times0.2
    \]
    \[
    30.000000000000003553 \neq 30
    \]
  - **Not cumulative**
    - Repeatedly adding a very small number to a large one may do nothing
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** = $2^{\text{bias}} = 2^7$
    - 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, \( \geq F_{\text{Over}} \)?

No

Yes

\( \pm \infty \)

\( E = \) all 1’s

\( M = \) all 0’s

Yes

Yes

\( \pm 0 \)

\( E = \) all 0’s

\( M = \) all 0’s

No

No

Denormed

\( E = \) all 0’s

\( 0.M = \) Man

Yes

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

Yes

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

No

Normed

\( E = \) Exp + bias

\( 1.M = \) Man

Yes

\( E = \) all 1’s

\( M = \) all 0’s

\( E = \) all 1’s

\( M = \) all 0’s

\( E = \) all 0’s

\( M = \) all 0’s

\( = \) special case
Example Question

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

  - No voting

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

- 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...
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

- 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` 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 \( f \)?

```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...
#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 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