Floating Point II
CSE 351 Winter 2020

Instructor: Ruth Anderson
Teaching Assistants: Jonathan Chen, Justin Johnson, Porter Jones, Josie Lee, Jeffery Tian, Callum Walker, Eddy (Tianyi) Zhou

http://xkcd.com/899/
Administrivia

- hw6 due Friday, hw7 due Monday

- Lab 1a due last night. Lates accepted until Thurs.

- Lab 1b due Monday (1/27)
  - Submit `bits.c` and `lab1Breflect.txt`

- Section tomorrow on Integers and Floating Point
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

- New largest value (besides ∞)?
  - E = 0xFF has now been taken!
  - E = 0xFE has largest: 1.1...1₂ × 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: $(−1)^{S} \times \infty$
- All 1's: 
  - What is the value of M?
    - All 0's: Denormalized: $(−1)^{S} \times 0.M \times 2^{1−bias}$
    - Anything else: Normalized: $(−1)^{S} \times 1.M \times 2^{E−bias}$
- Anything else: NaN

= 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:

  - Assume that it has the same properties as IEEE floating point:
    - bias = $2^{\text{bias}-1} - 1 = 2^{4-1} - 1 = 7$
    - encoding of $-0 = 0\text{b 1 0000 000 = 0x80}$
    - encoding of $+\infty = 0\text{b 0 1111 1000 = 0x78}$
    - encoding of the largest (+) normalized $# = 0\text{b 0 1110 111 = 0x77}$
    - encoding of the smallest (+) normalized $# = 0\text{b 0 0001 000 = 0x08}$

Distribution of Values

- What ranges are NOT representable?
  - Between largest norm and infinity \(\text{Overflow} \) (Exp too large)
  - Between zero and smallest denorm \(\text{Underflow} \) (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 \(\text{Exp} = 0\)? \(2^{-23}\)
  - What is this “step” when \(\text{Exp} = 100\)? \(2^{\text{Exp} - 23}\)
- 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 = $1.00101$ rounded to $M = 0b001$ (down)
  - Man = $1.00111$ rounded to $M = 0b010$ (up)
  - Man = $1.00110$ rounded to $M = 0b010$ (up)
  - Man = $1.00010$ rounded to $M = 0b000$ (down)

This is extra (non-testable) material
Floating Point Operations: Basic Idea

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

\[ S \quad E \quad M \]

\[ \begin{align*}
\text{x} + f \text{y} &= \text{Round}(\text{x} + \text{y}) \\
\text{x} \times f \text{y} &= \text{Round}(\text{x} \times \text{y})
\end{align*} \]

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: \( 100 \times (0.1 + 0.2) \neq 100 \times 0.1 + 100 \times 0.2 \)
    - \( 30.000000000000003553 \neq 30 \)
  - 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:
  - \(\text{FOver} = 2^{\text{bias}+1} = 2^8\)
    - This is just larger than the largest representable normalized number
  - \(\text{FDenorm} = 2^{1-\text{bias}} = 2^{-6}\)
    - This is the smallest representable normalized number
  - \(\text{FUnder} = 2^{1-\text{bias}-m} = 2^{-9}\)
    - \(m\) is the width of the mantissa field
    - This is the smallest representable denormalized number
Floating Point Encoding Flow Chart

Value ν to encode → Is ν not a number? → No → Is |ν|, when rounded, ≥ FOver? → Yes → NaN → E = all 1’s, M ≠ all 0’s → No

Yes → ±∞ → E = all 1’s, M = all 0’s

Yes → ±0 → E = all 0’s, M = all 0’s

Yes → Is |ν|, when rounded, < FUnder? → Normed → E = Exp + bias, 1.M = Man → No

No → Denormed → E = all 0’s, 0.M = Man → No

= special case
Example Question

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

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

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

- No voting

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

\[ S = 0 \]
\[ E = \text{Exp} + \text{bias} \]
\[ = 8 + 7 = 15 \]
\[ = 0b1111 \]

this falls outside of the normalized exponent range!

this number is too large, so we store

\[ +\infty \leftrightarrow 0b01111000 \]

instead
Polling Question

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

\[
\begin{array}{c|c|c|c}
S & E & M \\
1 & 4 & 3 \\
\end{array}
\]

\[
2^1 \times (1 + 2^{-2} + 2^{-4}) = 2^1 \times 1.0101_2
\]

- 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…

\[
\text{stored as: } \overline{\text{0b 0 1000 010} = 2.5}
\]
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}`

  `\uparrow` 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/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…

```
i stored as 0x 00 00 01 80
```

```
f stored as 0x 43 CO 00 00
```
Floating Point and the Programmer

#include <stdio.h>

int main(int argc, char* argv[]) {
    float f1 = 1.0; // specify float constant
    float f2 = 0.0;
    int i;
    for (i = 0; i < 10; i++)
        f2 += 1.0/10.0;
        // f2 should == \frac{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" );

    return 0;
}

$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119
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

<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 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