Floating Point
CSE 351 Summer 2018

Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee       Natalie Andreeva       Teagan Horkan

signed overflow in 16 bits $\rightarrow$ short (in C)

http://xkcd.com/571/
Administrivia

- Lab 1a due tonight at 11:59 pm
  - Only submit `pointer.c`
  - Make sure to use `dlc.py`

- Lab 1b due Thursday (7/5)
  - Submit `bits.c, lab1reflect.txt`
  - Start *early* and come to office hours!

- Homework 2 due 7/11
  - On Integers, Floating Point, and x86-64
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...
IEEE Floating Point

- IEEE 754
  - Established in 1985 as uniform standard for floating point arithmetic
  - Main idea: make numerically sensitive programs portable
  - Specifies two things: representation and result of floating operations
  - Now supported by all major CPUs

- Driven by numerical concerns
  - **Scientists**/numerical analysts want them to be as **real** as possible
  - **Engineers** want them to be **easy to implement** and **fast**
  - In the end:
    - Scientists mostly won out
    - Nice standards for rounding, overflow, underflow, but...
    - Hard to make fast in hardware
    - Float operations can be an order of magnitude slower than integer operations
Floating Point Encoding

- Use normalized, base 2 scientific notation:
  - Value: \( \pm 1 \times \text{Mantissa} \times 2^{\text{Exponent}} \)
  - Bit Fields: \((-1)^S \times 1.M \times 2^{(E-\text{bias})}\)

- Representation Scheme: (3 separate fields within 32 bits)
  - **Sign bit** (0 is positive, 1 is negative)
  - **Mantissa** (a.k.a. significand) is the fractional part of the number in normalized form and encoded in bit vector \(M\)
  - **Exponent** weights the value by a (possibly negative) power of 2 and encoded in the bit vector \(E\)
The Exponent Field

- Use biased notation
  - Read $E$ as unsigned, but with bias of $2^{w-1}-1 = 127$
  - Representable exponents roughly $\frac{1}{2}$ positive and $\frac{1}{2}$ negative
  - Exponent 0 ($Exp = 0$) is represented as $E = 0b\ 0111\ 1111 = 2^8 - 1$
  - $E - bias = 0 = Exp$

- Why biased?
  - Makes floating point arithmetic easier
  - Makes somewhat compatible with two’s complement

- Practice: To encode in biased notation, add the bias then encode in unsigned:
  - $Exp = 1$ → $128$ → $E = 0b\ 1000\ 0000$
  - $Exp = 127$ → $254$ → $E = 0b\ 1111\ 1110$
  - $Exp = -63$ → $64$ → $E = 0b\ 0100\ 0000$
The Mantissa (Fraction) Field

- Note the implicit 1 in front of the M bit vector
  - Example: 0b 0011 1111 1100 0000 0000 0000 0000 0000 is read as 1.1₂ = 1.5₁₀, not 0.1₂ = 0.5₁₀
  - Gives us an extra bit of precision

- Mantissa “limits”
  - Low values near M = 0b0...0 are close to $2^{E_{-bias}}$
  - High values near M = 0b1...1 are close to $2^{E_{bias}+1}$
Peer Instruction Question

- What is the correct value encoded by the following floating point number?
  - 0b 0 10000000 11000000000000000000000

  \[ \text{Exp} = 1 - \text{bias} = 128 - 127 = 1 \]
  \[ \text{Man} = 1.110\ldots_2 \text{ implicit} \]

  \[ +1.11_2 \times 2^1 \]
  \[ 1.11_2 = 2^1 + 2^0 + 2^{-1} = 3.5 \]


A. + 0.75
B. + 1.5
C. + 2.75
D. + 3.5
E. We’re lost…
Precision and Accuracy

- **Precision** is a count of the number of bits in a computer word used to represent a value
  - Capacity for accuracy

- **Accuracy** is a measure of the difference between the actual value of a number and its computer representation

  - *High precision permits high accuracy but doesn’t guarantee it.*
    - *It is possible to have high precision but low accuracy.*

  - **Example:** \( \text{float } \pi = 3.14; \)
    - \( \pi \) will be represented using all 24 bits of the mantissa (highly precise), but is only an approximation (not accurate)
Need Greater Precision?

- **Double Precision** (vs. Single Precision) in 64 bits

- C variable declared as `double`
- Exponent bias is now $2^{10} - 1 = 1023$, \( \text{bias} = 2^{\text{w}-1} - 1 \)

- **Advantages:** greater precision (larger mantissa), greater range (larger exponent)
- **Disadvantages:** more bits used, slower to manipulate
Representing Very Small Numbers

- But wait... what happened to zero?
  - Using standard encoding \(0x00000000\) = \(1.0 \times 2^{-127} \neq 0\)
  - Special case: \(E\) and \(M\) all zeros = 0
    - Two zeros! But at least \(0x00000000 = 0\) like integers
      \(0x80000000 = -0\)

- New numbers closest to 0:
  - \(E = 0x01, \ Exp = -126\)
    - \(a = 1.0...0_2 \times 2^{-126} = 2^{-126}\)
    - \(b = 1.0...01_2 \times 2^{-126} = 2^{-126} + 2^{-149}\)
  - 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: $\pm 1.0\ldots0_{\text{two}} \times 2^{-126} = \pm 2^{-126}$
  - Smallest denorm: $\pm 0.0\ldots01_{\text{two}} \times 2^{-126} = \pm 2^{-149}$
    - There is still a gap between zero and the smallest denormalized number
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...1₂ × 2¹²⁷ = 2¹²⁸ − 2¹⁰⁴
## 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>

- **smallest E (all 0's)**
- **everything else**
- **largest E (all 1's)**
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: 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
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 desired precision:
  - Possibly over/underflow if exponent outside of range
  - Possibly drop least-significant bits of mantissa to fit into M bit vector
Floating Point Addition

- \((-1)^{S_1} \times \text{Man}_1 \times 2^{\text{Exp}_1} + (-1)^{S_2} \times \text{Man}_2 \times 2^{\text{Exp}_2}\)
  - Assume \(\text{Exp}_1 > \text{Exp}_2\)

- Exact Result: \((-1)^S \times \text{Man} \times 2^{\text{Exp}}\)
  - Sign \(S\), mantissa \(\text{Man}\):
    - Result of signed align & add
  - Exponent \(E\): \(\text{E}_1\)

- Adjustments:
  - If \(\text{Man} \geq 2\), shift \(\text{Man}\) right, increment \(\text{Exp}\)
  - If \(\text{Man} < 1\), shift \(\text{Man}\) left \(k\) positions, decrement \(\text{Exp}\) by \(k\)
  - Over/underflow if \(\text{Exp}\) out of range
  - Round \(\text{Man}\) to fit mantissa precision
Floating Point Multiplication

\[ (-1)^{S_1} \times \text{Man}_1 \times 2^{\text{Exp}_1} \times (-1)^{S_2} \times \text{Man}_2 \times 2^{\text{Exp}_2} \]

- Exact Result: \((-1)^{S} \times \text{M} \times 2^{E}\)
  - Sign \(S\): \(S_1 \text{ ^ } S_2\)
  - Mantissa \(\text{Man}\): \(\text{Man}_1 \times \text{Man}_2\)
  - Exponent \(\text{Exp}\): \(\text{Exp}_1 + \text{Exp}_2\)

- Adjustments:
  - If \(\text{Man} \geq 2\), shift \(\text{Man}\) right, increment \(\text{Exp}\)
  - Over/underflow if \(\text{Exp}\) out of range
  - Round \(\text{Man}\) to fit mantissa precision
Mathematical Properties of FP Operations

- Exponent overflow yields $+\infty$ or $-\infty$
- Floats with value $+\infty$, $-\infty$, and NaN can be used in operations
  - Result usually still $+\infty$, $-\infty$, 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
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

- C offers two (well, 3) levels of precision
  - float 1.0f single precision (32-bit)
  - double 1.0 double precision (64-bit)
  - long double 1.0L ("double double" or quadruple) precision (64-128 bits)

- #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!

  instead use \( \text{abs}(f_1 - f_2) < 2^{-20} \)

  \( \uparrow \) 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 $T_{min}$ (even if the value is a very big positive)
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 == 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; \times 10^{30}
    f2 = 1E-30; \times 10^{-30}
    float f3 = f1 + f2;
    printf("f1 == f3? %s\n", f1 == f3 ? "yes" : "no" );
    \times 10^{30} == \times 10^{30} + \times 10^{-30}
    return 0;
}
```

$ ./a.out
0x3f800000  0x3f800001
f1 = 1.000000000
f2 = 1.000000119
f1 == f3? yes
Summary

- Floating point approximates real numbers:
  - Handles large numbers, small numbers, special numbers
  - Exponent in biased notation (bias = $2^{w-1} - 1$)
    - Outside of representable exponents is overflow and underflow
  - Mantissa approximates fractional portion of binary point
    - Implicit leading 1 (normalized) except in special cases
    - Exceeding length causes rounding

<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>
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</th>
<th>E</th>
<th>M</th>
<th>Exp</th>
<th>Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 000</td>
<td>-6</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 001</td>
<td>-6</td>
<td>1/8*1/64 = 1/512</td>
<td>closest to zero</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 010</td>
<td>-6</td>
<td>2/8*1/64 = 2/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 110</td>
<td>-6</td>
<td>6/8*1/64 = 6/512</td>
<td>largest denorm</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 111</td>
<td>-6</td>
<td>7/8*1/64 = 7/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001 000</td>
<td>-6</td>
<td>8/8*1/64 = 8/512</td>
<td>smallest norm</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0001 001</td>
<td>-6</td>
<td>9/8*1/64 = 9/512</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Denormalized numbers

<table>
<thead>
<tr>
<th>S</th>
<th>E</th>
<th>M</th>
<th>Exp</th>
<th>Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0111 110</td>
<td>-1</td>
<td>14/8*1/2 = 14/16</td>
<td>closest to 1 below</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 111</td>
<td>-1</td>
<td>15/8*1/2 = 15/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
<td>closest to 1 above</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
<td>largest norm</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1111 000</td>
<td>n/a</td>
<td>inf</td>
<td></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