Floating Point
CSE 351 Summer 2018

Instructor:
Justin Hsia

Teaching Assistants:
Josie Lee    Natalie Andreeva    Teagan Horkan

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: $(-1)^S \times 1.M \times 2^{(E-bias)}$

- Representation Scheme:
  - 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 $E = 0b\ 0111\ 1111$

- 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:
  - $Exp = 1 \rightarrow E = 0b$
  - $Exp = 127 \rightarrow E = 0b$
  - $Exp = -63 \rightarrow E = 0b$
The Mantissa (Fraction) Field

\[ (-1)^S \times (1 \cdot M) \times 2^{(E-\text{bias})} \]

- 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 \( M = 0b0\ldots0 \) are close to \( 2^{\text{Exp}} \)
  - High values near \( M = 0b1\ldots1 \) are close to \( 2^{\text{Exp}+1} \)
Peer Instruction Question

What is the correct value encoded by the following floating point number?

- 0b 0 10000000 110000000000000000000000

Vote at http://PollEv.com/justinh

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:**
  ```c
  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 $2^{10} - 1 = 1023$

- **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 =
  - *Special case:* $E$ and $M$ all zeros = 0
    - Two zeros! But at least 0x00000000 = 0 like integers

- New numbers closest to 0:
  - $a = 1.0\ldots0_2 \times 2^{-126} = 2^{-126}$
  - $b = 1.0\ldots01_2 \times 2^{-126} = 2^{-126} + 2^{-149}$
  - 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

- **$E = 0xFF, M = 0$:** $\pm \infty$
  - 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

- **New largest value (besides $\infty$)?**
  - $E = 0xFF$ 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>
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 Operations: Basic Idea

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

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

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 \(E\): \(E_1\)
- 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 \(S\): \(S_1 \oplus S_2\)
  - 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: $(3.14+1e100)-1e100 \neq 3.14+(1e100-1e100)$
    - $0 \quad 3.14$
  - Not distributive: $100*(0.1+0.2) \neq 100*0.1+100*0.2$
    - $30.000000000000003553 \quad 30$
  - 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!
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 `int`s 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 $T_{\text{min}}$ (even if the value is a very big positive)
#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
", *(int*)&f1, *(int*)&f2);

    printf("f1 = %10.9f
", 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;
}
Summary

 floatValueapproximates 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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000 000</td>
<td>000</td>
<td>-6</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0000 001</td>
<td>001</td>
<td>-6</td>
<td>$1/8 \times 1/64 = 1/512$</td>
</tr>
<tr>
<td>0</td>
<td>0000 010</td>
<td>010</td>
<td>-6</td>
<td>$2/8 \times 1/64 = 2/512$</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td>-6</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0000 110</td>
<td>110</td>
<td>-6</td>
<td>$6/8 \times 1/64 = 6/512$</td>
</tr>
<tr>
<td>0</td>
<td>0000 111</td>
<td>111</td>
<td>-6</td>
<td>$7/8 \times 1/64 = 7/512$</td>
</tr>
</tbody>
</table>

### Denormalized numbers
- closest to zero

### Normalized numbers
- smallest norm
- closest to 1 below
- closest to 1 above
- largest norm

0 0001 000 | 000 | -6 | $8/8 \times 1/64 = 8/512$ |
0 0001 001 | 001 | -6 | $9/8 \times 1/64 = 9/512$ |

... 0 0110 110 | 110 | -1 | $14/8 \times 1/2 = 14/16$ |
0 0110 111 | 111 | -1 | $15/8 \times 1/2 = 15/16$ |
0 0111 000 | 111 | 0 | $8/8 \times 1 = 1$ |
0 0111 001 | 111 | 0 | $9/8 \times 1 = 9/8$ |
0 0111 010 | 111 | 0 | $10/8 \times 1 = 10/8$ |

... 0 1110 110 | 110 | 7 | $14/8 \times 128 = 224$ |
0 1110 111 | 111 | 7 | $15/8 \times 128 = 240$ |
0 1111 000 | n/a | inf | inf |
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