COMP201
Computer Systems & Programming

Lecture #05 – Floating Point

Aykut Erdem // Koç University // Fall 2020
Good news, everyone!

• Assg1 is due Oct 16 18

• Assg1 will be out on Oct 16 18 (due Oct 26 28)

• Soon you will start to use linuxpool.ku.edu.tr cluster of CentOS 7 machines.
Recap

- Representing real numbers
- Fixed Point
- Floating Point
Plan For Today

• Example and Properties
• Floating Point Arithmetic
• Floating Point in C

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
—Randal E. Bryant and David R. O’Hallaron's CMU 15-213 class
Lecture Plan

• Example and Properties
• Floating Point Arithmetic
• Floating Point in C
Tiny Floating Point Example

- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next four bits are the exponent, with a bias of 7
  - the last three bits are the \texttt{frac}

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity
## Dynamic Range (Positive Only)

<table>
<thead>
<tr>
<th>s exp frac</th>
<th>E</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0000 000</td>
<td>−6</td>
<td>0</td>
</tr>
<tr>
<td>0 0000 001</td>
<td>−6</td>
<td>1/8*1/64 = 1/512</td>
</tr>
<tr>
<td>0 0000 010</td>
<td>−6</td>
<td>2/8*1/64 = 2/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0000 110</td>
<td>−6</td>
<td>6/8*1/64 = 6/512</td>
</tr>
<tr>
<td>0 0000 111</td>
<td>−6</td>
<td>7/8*1/64 = 7/512</td>
</tr>
<tr>
<td>0 0001 000</td>
<td>−6</td>
<td>8/8*1/64 = 8/512</td>
</tr>
<tr>
<td>0 0001 001</td>
<td>−6</td>
<td>9/8*1/64 = 9/512</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0110 110</td>
<td>−1</td>
<td>14/8*1/2 = 14/16</td>
</tr>
<tr>
<td>0 0110 111</td>
<td>−1</td>
<td>15/8*1/2 = 15/16</td>
</tr>
<tr>
<td>0 0111 000</td>
<td>0</td>
<td>8/8*1 = 1</td>
</tr>
<tr>
<td>0 0111 001</td>
<td>0</td>
<td>9/8*1 = 9/8</td>
</tr>
<tr>
<td>0 0111 010</td>
<td>0</td>
<td>10/8*1 = 10/8</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1110 110</td>
<td>7</td>
<td>14/8*128 = 224</td>
</tr>
<tr>
<td>0 1110 111</td>
<td>7</td>
<td>15/8*128 = 240</td>
</tr>
<tr>
<td>0 1111 000</td>
<td>n/a</td>
<td>inf</td>
</tr>
</tbody>
</table>

**Denormalized numbers**

**Normalized numbers**

\[ v = (-1)^s M 2^E \]

\[ n: E = \text{Exp} - \text{Bias} \]

\[ d: E = 1 - \text{Bias} \]
Distribution of Values

• 6-bit IEEE-like format
  • $e = 3$ exponent bits
  • $f = 2$ fraction bits
  • Bias is $2^{3-1}-1 = 3$

• Notice how the distribution gets denser toward zero.
Distribution of Values (close-up view)

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3
Special Properties of the IEEE Encoding

• FP Zero Same as Integer Zero
  • All bits = 0

• Can (Almost) Use Unsigned Integer Comparison
  • Must first compare sign bits
  • Must consider \(-0 = 0\)
  • NaNs problematic
    • Will be greater than any other values
    • What should comparison yield?
  • Otherwise OK
    • Denorm vs. normalized
    • Normalized vs. infinity
Lecture Plan

• Example and Properties
• Floating Point Arithmetic
• Floating Point in C
Demo: Float Arithmetic

float_arithmetic.c
Floating Point Arithmetic

Is this just overflowing? It turns out it’s more subtle.

```c
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %g\n", (a + b) - b); // prints 0
printf("3.14 + (1e20 - 1e20) = %g\n", a + (b - b)); // prints 3.14
```

Let's look at the binary representations for 3.14 and 1e20:

- **3.14:**
  ```
  \[
  \begin{array}{cccc}
  31 & 30 & 23 & 22 \\
  0 & 100000000 & 1001000111101011100011 \\
  \end{array}
  \]
  ```

- **1e20:**
  ```
  \[
  \begin{array}{cccc}
  31 & 30 & 23 & 22 \\
  0 & 110000001 & 01011010111100011101100 \\
  \end{array}
  \]
  ```
Floating Point Arithmetic

To add real numbers, we must align their binary points:

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.14:</td>
<td>0</td>
<td>10000000</td>
<td>10010001111010111000011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1e20:</td>
<td>0</td>
<td>11000001</td>
<td>01011010111100011101100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What does this number look like in 32-bit IEEE format?
Floating Point Arithmetic

Step 1: convert from base 10 to binary

What is 100000000000000000003.14 in binary? Let’s find out!


10101101011110001110101110010110101100011000100000000000000000011.0010001111010111000010100011…
Floating Point Arithmetic

**Step 2:** find most significant 1 and take the next 23 digits for the fractional component, rounding if needed.

1 01011010111100011101100

1010110101111000111010111000101101011000110001000000000000000001.001000111101100001010011…

1 01011010111100011101100
Floating Point Arithmetic

**Step 3:** find how many places we need to shift **left** to put the number in 1.xxx format. This fills in the exponent component.

1010110101111000111010111000101101011000110001011010110001100010000000000000000011.001000111101011100001010011…

66 shifts -> 66 + 127 = 193
Floating Point Arithmetic

Step 4: if the sign is positive, the sign bit is 0. Otherwise, it’s 1.

Sign bit is 0.
Floating Point Arithmetic

The binary representation for $1\times10^{20} + 3.14$ thus equals the following:

```
<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>23</th>
<th>22</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11000001</td>
<td>01011010111100011101100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

This is the same as the binary representation for $1\times10^{20}$ that we had before!

We didn’t have enough bits to differentiate between $1\times10^{20}$ and $1\times10^{20} + 3.14$. 
Floating Point Arithmetic

Is this just overflowing? It turns out it’s more subtle.

```c
float a = 3.14;
float b = 1e20;
printf("(3.14 + 1e20) - 1e20 = %g\n", (a + b) - b);  // prints 0
printf("3.14 + (1e20 - 1e20) = %g\n", a + (b - b));  // prints 3.14
```

Floating point arithmetic is not associative. The order of operations matters!

• The first line loses precision when first adding 3.14 and 1e20, as we have seen.
• The second line first evaluates 1e20 − 1e20 = 0, and then adds 3.14
Demo: Float Equality

`float_equality.c`
Floating Point Arithmetic

Float arithmetic is an issue with most languages, not just C!

• http://geocar.sdf1.org/numbers.html
Let’s Get Real

What would be nice to have in a real number representation?

• Represent widest range of numbers possible ✔
• Flexible “floating” decimal point ✔
• Represent scientific notation numbers, e.g. 1.2 x 10^6 ✔
• Still be able to compare quickly ✔
• Have more predictable over/under-flow behavior ✔
Lecture Plan

• Example and Properties
• Floating Point Arithmetic
• Floating Point in C
Floating Point in C

• C Guarantees Two Levels
  • `float`  single precision
  • `double`  double precision

• Conversions/Casting
  • Casting between `int`, `float`, and `double` changes bit representation
  • `double/float → int`
    • Truncates fractional part
    • Like rounding toward zero
    • Not defined when out of range or NaN: Generally sets to TMin
  • `int → double`
    • Exact conversion, as long as `int` has $\leq 53$ bit word size
  • `int → float`
    • Will round according to rounding mode
Ariane 5: A Bug and A Crash

- On June 4, 1996, Ariane 5 rocket self destructed just after 37 seconds after liftoff
- **Cost:** $500 million
- **Cause:** An overflow in the conversion from a 64 bit floating point number to a 16 bit signed integer
- A design flaw:
  - 5 times faster than Ariane 4
  - Reused same software specifications from Ariane 4
  - Ariane 4 assumes horizontal velocity would never overflow a 16-bit number

Practice Problem 2.54

Assume variables $x$, $f$, and $d$ are of type `int`, `float`, and `double`, respectively. Their values are arbitrary, except that neither $f$ nor $d$ equals $+\infty$, $-\infty$, or NaN.

For each of the following C expressions, either argue that it will always be true (i.e., evaluate to 1) or give a value for the variables such that it is not true (i.e., evaluates to 0).

A. $x == (int)(double) x$
B. $x == (int)(float) x$
C. $d == (double)(float) d$
D. $f == (float)(double) f$
E. $f == -(-f)$
F. $1.0/2 == 1/2.0$
G. $d*d >= 0.0$
H. $(f+d)-f == d$
Floating Point Puzzles

• For each of the following C expressions, either:
  • Argue that it is true for all argument values
  • Explain why not true

```c
int x = ...;
float f = ...;
double d = ...;
```

Assume neither `d` nor `f` is NaN

<table>
<thead>
<tr>
<th>Expression</th>
<th>Truth</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x == (int)(float) x</code></td>
<td>False</td>
</tr>
<tr>
<td><code>x == (int)(double) x</code></td>
<td>True</td>
</tr>
<tr>
<td><code>f == (float)(double) f</code></td>
<td>True</td>
</tr>
<tr>
<td><code>d == (float) d</code></td>
<td>False</td>
</tr>
<tr>
<td><code>f == -(-f)</code></td>
<td>True</td>
</tr>
<tr>
<td><code>2/3 == 2/3.0</code></td>
<td>False</td>
</tr>
<tr>
<td><code>d &lt; 0.0 ⇒ ((d*2) &lt; 0.0)</code></td>
<td>True (OF?)</td>
</tr>
<tr>
<td><code>d &gt; f ⇒ -f &gt; -d</code></td>
<td>True</td>
</tr>
<tr>
<td><code>d * d &gt;= 0.0</code></td>
<td>True (OF?)</td>
</tr>
<tr>
<td><code>(d+f)-d == f</code></td>
<td>False</td>
</tr>
</tbody>
</table>
Floors Summary

• IEEE Floating Point is a carefully-thought-out standard. It’s complicated, but engineered for their goals.

• Floats have an extremely wide range, but cannot represent every number in that range.

• Some approximation and rounding may occur! This means you definitely don’t want to use floats e.g. for currency.

• Associativity does not hold for numbers far apart in the range

• Equality comparison operations are often unwise.
Recap

• Representing real numbers
• Fixed Point
• Floating Point
• Floating Point Arithmetic

Next time: How can a computer represent and manipulate more complex data like text?