COMP201
Computer Systems & Programming

Lecture #05 – Floating Point

Aykut Erdem // Koç University // Fall 2020
Recap: Bitwise Operators

• You’re already familiar with many operators in C:
  – Arithmetic operators: +, -, *, /, %
  – Comparison operators: ==, !=, <, >, <=, >=
  – Logical Operators: &&, ||, !

• Bitwise operators:
  • Logical operators: &, |, ~, ^,
  • Bit shift operators: <<, >>
GitHub Classroom – What happened?

Important announcement about assignment 0 and this week’s lab sessions

Posted on Monday, October 12, 2020 5:54:04 PM TRT

Dear COMP301 students,

We are experiencing technical issues with Github Classroom and many of you may have faced restrictions to your assignment repositories on Github Classroom today. We are in touch with the IT department and striving to come up with a workaround as soon as possible. In the meantime, to minimize delay and enable you to make progress towards the planned course agenda, please pay attention to the points below.

- Because of restrictions put on the Classroom, you won't be able to work on your private assignment repositories. Thus, please work on your starter code locally and test the correctness of your implementation using the test cases provided in the starter code.
- We are expecting the problem to be resolved soon so that you will be able to push your local repository to the Github assignment repository (just like it was originally expected). However, if the restrictions were not lifted before the deadline, you will need to upload your final version to Blackboard instead.
- If you do not have access to the Linux command line on your local machine, you can visit Repl.it and upload your code on Repl.it to be able to work on it from your browser.
- If you do not have access to the starter code, you can clone it from the link below:
  
  [https://github.com/Mandana88/M Assignment1](https://github.com/Mandana88/M Assignment1)

  The changes needed for your lab sessions will be announced individually by the corresponding TAs.

Best regards,
Aykut

Lecture 4 slides posted
GitHub Classroom – What happened?

- Someone accessed GitHub Classroom repository from abroad.
- Please use VPN if you’re accessing GitHub from outside.
- We will inform you about the new submission procedure soon.
New Lab Sections Added to KUSIS!

- Based on the lab section assignments, we have now officially
  - 2 new lab sections on Wednesday (LAB E-F)
  - 1 new lab section on Thursday (LAB D)

- Please change your section accordingly!
  - This is important for setting up the participants for the Zoom meetings.
New Lab Sections Added to KUSIS!

- Based on the final lab section assignments, we have now
- 2 new (official) lab sections on Wednesday (LAB E-F)
- 1 new (official) lab section on Thursday (LAB E-F)

Please change your section accordingly!

This is important for Zoom meeting settings.

Note that the Section IDs are not in alphabetical order!
About Quiz 1 & 2

• Due to a series of unfortunate events, Quiz 1 hasn’t been released yet.

• On Friday, you will take Quiz 1 and Quiz 2 together in one sitting.
  • 20 mins, between 14:35-14:55

• Topics covered: B&O 2.1-2.4

• There will be a mock-up quiz at the end of today’s lecture (will not be graded)
COMP201 Topic 2: How can a computer represent real numbers in addition to integer numbers?
Learning Goals

Understand the design and compromises of the floating point representation, including:

• Fixed point vs. floating point
• How a floating point number is represented in binary
• Issues with floating point imprecision
• Other potential pitfalls using floating point numbers in programs
Plan For Today

• Representing real numbers
• Fixed Point
• Floating Point

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
Lecture Plan

• Representing real numbers
  • Fixed Point
  • Floating Point
Real Numbers

• We previously discussed representing integer numbers using two’s complement.
• However, this system does not represent real numbers such as 3/5 or 0.25.
• How can we design a representation for real numbers?
Real Numbers

**Problem**: unlike with the integer number line, where there are a finite number of values between two numbers, there are an *infinite* number of real number values between two numbers!

Integers between 0 and 2: 1
Real Numbers Between 0 and 2: 0.1, 0.01, 0.001, 0.0001, 0.00001, ...

We need a fixed-width representation for real numbers. Therefore, by definition, *we will not be able to represent all numbers.*
Real Numbers

Problem: every number base has un-representable real numbers.

Base 10: \( \frac{1}{6}_{10} = 0.16666666\ldots_{10} \)

Base 2: \( \frac{1}{10}_{10} = 0.000110011001100110011\ldots_2 \)

Therefore, by representing in base 2, we will not be able to represent all numbers, even those we can exactly represent in base 10.
Fixed Point

• **Idea:** Like in base 10, let’s add binary decimal places to our existing number representation.

\[ \begin{align*}
\text{5} & \quad \text{9} & \quad \text{3} & \quad \text{4} & \quad . & \quad \text{2} & \quad \text{1} & \quad \text{6} \\
10^3 & & 10^2 & & 10^1 & & 10^0 & & 10^{-1} & & 10^{-2} & & 10^{-3} \\
\text{1} & \quad \text{0} & \quad \text{1} & \quad \text{1} & \quad . & \quad \text{0} & \quad \text{1} & \quad \text{1} \\
2^3 & & 2^2 & & 2^1 & & 2^0 & & 2^{-1} & & 2^{-2} & & 2^{-3}
\end{align*} \]
Lecture Plan

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

• **Idea:** Like in base 10, let’s add binary decimal places to our existing number representation.

1 0 1 1 . 0 1 1

8s  4s  2s  1s  1/2s  1/4s  1/8s

• **Pros:** arithmetic is easy! And we know exactly how much precision we have.
Fixed Point

• **Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

\[
.0110011
\]

\[
1/2s \quad 1/4s \quad 1/8s \quad ... \]

\[
10110.11
\]

16s 8s 4s 2s 1s 1/2s 1/4s
Fixed Point

• **Problem:** we have to fix where the decimal point is in our representation. What should we pick? This also fixes us to 1 place per bit.

To be able to store both these numbers using the same fixed point representation, the bitwidth of the type would need to be at least 207 bits wide!

- Base 10
  - $5.07 \times 10^{30} = 1.0 \ldots 0.1$
  - 100 zeros

- Base 2
  - $9.86 \times 10^{-32} = 0.0 \ldots 01$
  - 100 zeros
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. $1.2 \times 10^6$
• Still be able to compare quickly
• Have more predictable over/under-flow behavior
Lecture Plan

- Representing real numbers
- Fixed Point
- Floating Point
IEEE Floating Point

Let’s aim to represent numbers of the following scientific-notation-like format:

\[ x \times 2^y \]

With this format, 32-bit floats represent numbers in the range \(~1.2 \times 10^{-38}\) to \(~3.4 \times 10^{38}\). Is every number between those representable? **No.**
IEEE Single Precision Floating Point

\[ x \times 2^y \]

- **Sign bit** (0 = positive)
- **exponent** (8 bits)
- **fraction** (23 bits)
### Exponent

<table>
<thead>
<tr>
<th>Exponent (Binary)</th>
<th>Exponent (Base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111111111</td>
<td>?</td>
</tr>
<tr>
<td>1111111110</td>
<td>?</td>
</tr>
<tr>
<td>111111101</td>
<td>?</td>
</tr>
<tr>
<td>111111100</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td>?</td>
</tr>
<tr>
<td>000000011</td>
<td>?</td>
</tr>
<tr>
<td>000000010</td>
<td>?</td>
</tr>
<tr>
<td>000000001</td>
<td>?</td>
</tr>
<tr>
<td>000000000</td>
<td>?</td>
</tr>
</tbody>
</table>
## Exponent

The exponent is divided into two parts:
- **exponent (8 bits)**
- **fraction (23 bits)**

<table>
<thead>
<tr>
<th>Exponent (Binary)</th>
<th>Exponent (Base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1111111100</td>
<td>?</td>
</tr>
<tr>
<td>1111111001</td>
<td>?</td>
</tr>
<tr>
<td>1111111000</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td>?</td>
</tr>
<tr>
<td>0000000111</td>
<td>?</td>
</tr>
<tr>
<td>0000000100</td>
<td>?</td>
</tr>
<tr>
<td>0000000011</td>
<td>?</td>
</tr>
<tr>
<td>0000000000</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
### Exponent

<table>
<thead>
<tr>
<th>Exponent (Binary)</th>
<th>Exponent (Base 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111111111</td>
<td>RESERVED</td>
</tr>
<tr>
<td>1111111100</td>
<td>127</td>
</tr>
<tr>
<td>11111101</td>
<td>126</td>
</tr>
<tr>
<td>11111100</td>
<td>125</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>000000011</td>
<td>-124</td>
</tr>
<tr>
<td>00000010</td>
<td>-125</td>
</tr>
<tr>
<td>00000001</td>
<td>-126</td>
</tr>
<tr>
<td>0000000000</td>
<td>RESERVED</td>
</tr>
</tbody>
</table>
Exponent

- The exponent is **not** represented in two’s complement.
- Instead, exponents are sequentially represented starting from 000...1 (most negative) to 111...10 (most positive). This makes bit-level comparison fast.

**Actual value = binary value – 127 (“bias”)**

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>11111110</td>
<td>254 - 127 = 127</td>
</tr>
<tr>
<td>111111101</td>
<td>253 - 127 = 126</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>00000010</td>
<td>2 - 127 = -125</td>
</tr>
<tr>
<td>00000001</td>
<td>1 - 127 = -126</td>
</tr>
</tbody>
</table>
We could just encode whatever $x$ is in the fraction field. But there's a trick we can use to make the most out of the bits we have.
An Interesting Observation

In Base 10:
42.4 \times 10^5 = 4.24 \times 10^6
324.5 \times 10^5 = 3.245 \times 10^7
0.624 \times 10^5 = 6.24 \times 10^4

In Base 2:
10.1 \times 2^5 = 1.01 \times 2^6
1011.1 \times 2^5 = 1.0111 \times 2^8
0.110 \times 2^5 = 1.10 \times 2^4

We tend to adjust the exponent until we get down to one place to the left of the decimal point.

Observation: in base 2, this means there is always a 1 to the left of the decimal point!
• We can adjust this value to fit the format described previously. Then, x will always be in the format $1.xxxxxxxxxx...$

• Therefore, in the fraction portion, we can encode just what is to the right of the decimal point! This means we get one more digit for precision.

Value encoded = $1._{[\text{FRACTION BINARY DIGITS]}}$
Is this number:
A) Greater than 0?
B) Less than 0?

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>...0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>010...</td>
</tr>
</tbody>
</table>

Is this number:
A) Less than -1?
B) Between -1 and 1?
C) Greater than 1?
Skipping Numbers

• We said that it’s not possible to represent all real numbers using a fixed-width representation. What does this look like?

Float Converter

• https://www.h-schmidt.net/FloatConverter/IEEE754.html

Floats and Graphics

• https://www.shadertoy.com/view/4tVyDK
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. $1.2 \times 10^6$  ❓
• Still be able to compare quickly ✔
• Have more predictable over/under-flow behavior  ❓
Representing Zero

The float representation of zero is all zeros (with any value for the sign bit)

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>All zeros</td>
<td>All zeros</td>
</tr>
</tbody>
</table>

• This means there are two representations for zero! 😞
Representing Small Numbers

If the exponent is all zeros, we switch into “denormalized” mode.

- We now treat the exponent as -126, and the fraction as without the leading 1.
- This allows us to represent the smallest numbers as precisely as possible.

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>All zeros</td>
<td>Any</td>
</tr>
</tbody>
</table>
Representing Exceptional Values

If the exponent is all ones, and the fraction is all zeros, we have +- infinity.

- The sign bit indicates whether it is positive or negative infinity.
- Floats have built-in handling of over/underflow!
  - Infinity + anything = infinity
  - Negative infinity + negative anything = negative infinity
  - Etc.

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>All ones</td>
<td>All zeros</td>
</tr>
</tbody>
</table>
Representing Exceptional Values

If the exponent is all ones, and the fraction is nonzero, we have Not a Number (NaN)

- NaN results from computations that produce an invalid mathematical result.
  - Sqrt(negative)
  - Infinity / infinity
  - Infinity + -infinity
  - Etc.

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>1</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any nonzero</td>
</tr>
</tbody>
</table>
Number Ranges

• 32-bit integer (type int):
  › -2,147,483,648 to 2,147,483,647
  › Every integer in that range can be represented

• 64-bit integer (type long):
  › -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

• 32-bit floating point (type float):
  – ~1.2 x10^-38 to ~3.4 x10^38
  – Not all numbers in the range can be represented (not even all integers in the range can be represented!)
  – Gaps can get quite large! (larger the exponent, larger the gap between successive fraction values)

• 64-bit floating point (type double):
  – ~2.2 x10^-308 to ~1.8 x10^308
Precision options

• Single precision: 32 bits
  
  ![Single Precision Diagram](image)
  
  s  exp  frac
  1  8-bits  23-bits

• Double precision: 64 bits
  
  ![Double Precision Diagram](image)
  
  s  exp  frac
  1  11-bits  52-bits

• Extended precision: 80 bits (Intel only)
  
  ![Extended Precision Diagram](image)
  
  s  exp  frac
  1  15-bits  63 or 64-bits
Visualization: Floating Point Encodings
Recap

• Representing real numbers
• Fixed Point
• Floating Point

Next time: More on floating points. How can a computer perform arithmetic operating on floating points?
What Every Computer Scientist Should Know About Floating-Point Arithmetic

DAVID GOLDBERG
Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304

Floating-point arithmetic is considered an esoteric subject by many people. This is rather surprising, because floating-point is ubiquitous in computer systems: Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. This paper presents a tutorial on the aspects of floating-point that have a direct impact on designers of computer systems. It begins with background on floating-point representation and rounding error, continues with a discussion of the IEEE floating-point standard, and concludes with examples of how computer system builders can better support floating point.

Categories and Subject Descriptors: (Primary) C.0 [Computer Systems Organization]: General—instruction set design; D.3.4 [Programming Languages]: Processors—compilers, optimization; G.1.0 [Numerical Analysis]: General—computer arithmetic, error analysis, numerical algorithms (Secondary) D.2.1 [Software Engineering]: Requirements/Specifications—languages; D.3.1 [Programming Languages]: Formal Definitions and Theory—semantics D.4.1 (Operating Systems): Process Management—synchronization

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: denormalized number, exception, floating-point, floating-point standard, gradual underflow, guard digit, NaN, overflow, relative error, rounding error, rounding mode, ulp, underflow