photo by unsplash user @swimstaralex

COMP201

Computer Systems & Programming

Lecture #04 – Floating Point

KOÇ UNIVERSITY

Aykut Erdem // Koç University // Spring 2021

Image: Professor Farnsworth (Futurama)

Good news, everyone!

- Labs start this week.
- Assg1 will be out on Feb 25 (due Mar 11)

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:** `<<`, `>>`

Recap: 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*.

Plan For Today

- Floating Point
- 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

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

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. 1.2×10^6
- Still be able to compare quickly
- Have more predictable over/under-flow behavior

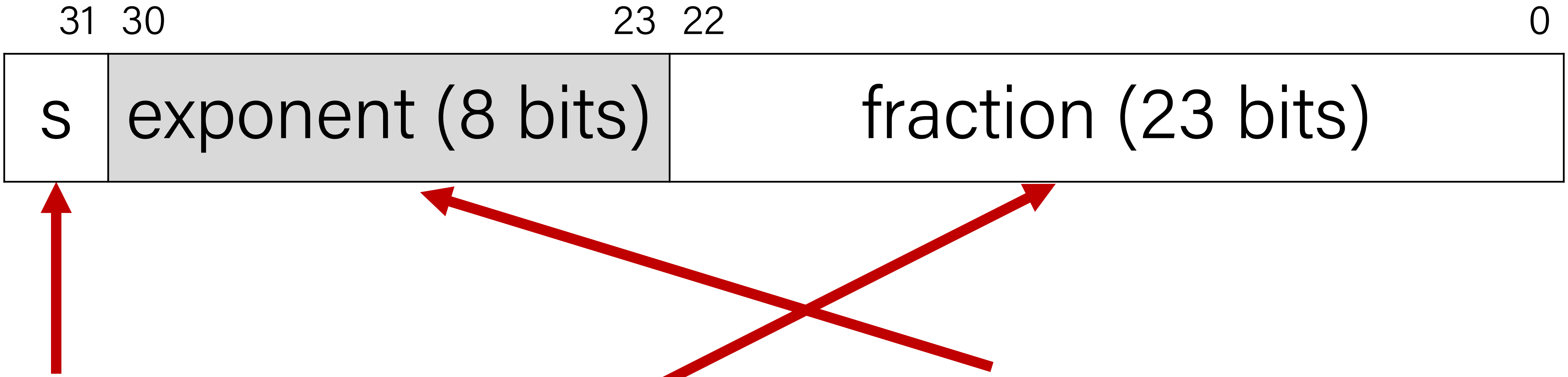
IEEE Floating Point

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

$$x \ * 2^{y}$$

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



Sign bit
(0 = positive)

$$x \; * \; 2^{y}$$

Exponent

s	exponent (8 bits)	fraction (23 bits)

Exponent (Binary)	Exponent (Base 10)
11111111	?
11111110	?
11111101	?
11111100	?
…	?
00000011	?
00000010	?
00000001	?
00000000	?

Exponent

s	exponent (8 bits)	fraction (23 bits)

Exponent (Binary)	Exponent (Base 10)
11111111	RESERVED
11111110	?
11111101	?
11111100	?
…	?
00000011	?
00000010	?
00000001	?
00000000	RESERVED

Exponent

s	exponent (8 bits)	fraction (23 bits)

Exponent (Binary)	Exponent (Base 10)
11111111	RESERVED
11111110	127
11111101	126
11111100	125
…	…
00000011	-124
00000010	-125
00000001	-126
00000000	RESERVED

Exponent

s	exponent (8 bits)	fraction (23 bits)

- 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")**

`11111110`	`254 - 127 = 127`
`11111101`	`253 - 127 = 126`
…	…
`00000010`	`2 - 127 = -125`
`00000001`	`1 - 127 = -126`

Fraction

s	exponent (8 bits)	fraction (23 bits)

$$x * 2^{y}$$

- 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:

$42.4 \times 10^5 = 4.24 \times 10^6$

$324.5 \times 10^5 = 3.245 \times 10^7$

$0.624 \times 10^5 = 6.24 \times 10^4$

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

In Base 2:

$10.1 \times 2^5 = 1.01 \times 2^6$

$1011.1 \times 2^5 = 1.0111 \times 2^8$

$0.110 \times 2^5 = 1.10 \times 2^4$

Observation: in base 2, this means there is always a 1 to the left of the decimal point!

Fraction

s	exponent (8 bits)	fraction (23 bits)

$$x * 2^{y}$$

- We can adjust this value to fit the format described previously. Then, x will always be in the format 1.XXXXXXXXX…
- 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._[FRACTION BINARY DIGITS]_

Practice

Sign	Exponent						Fraction			
0	0	…	0	0	0	1	0	1	0	…

Is this number:

A) Greater than 0?

B) Less than 0?

Practice

Sign	Exponent						Fraction			
0	0	…	0	0	0	1	0	1	0	…

Is this number:

A) Greater than 0?

B) Less than 0?

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. 1.2×10^6 ❓
- 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)

Sign	Exponent	Fraction
`any`	`All zeros`	`All zeros`

- This means there are two representations for zero! ☹

Representing Small Numbers

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

Sign	Exponent	Fraction
`any`	`All zeros`	`Any`

- 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.

Representing Exceptional Values

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

Sign	Exponent	Fraction
`any`	`All ones`	`All zeros`

- 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.

Representing Exceptional Values

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

Sign	Exponent						Fraction
any	1	…	…	…	…	1	Any nonzero

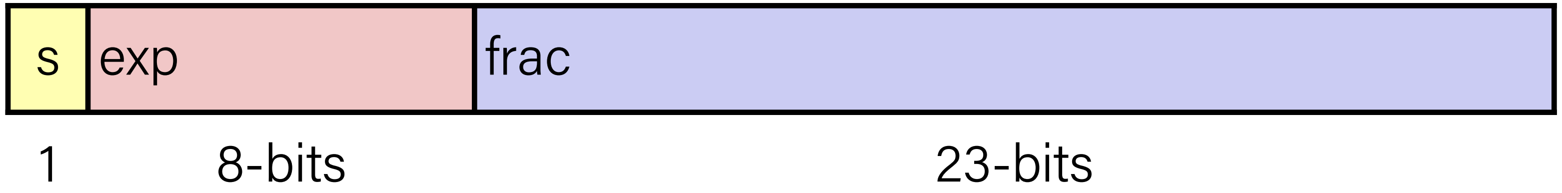
- NaN results from computations that produce an invalid mathematical result.
  - Sqrt(negative)
  - Infinity / infinity
  - Infinity + -infinity
  - Etc.

Number Ranges

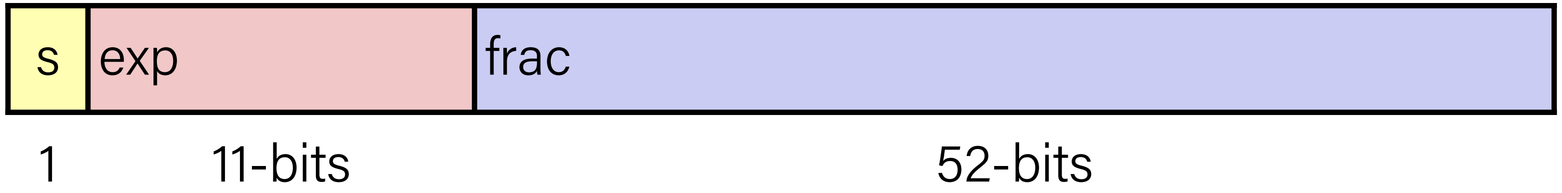
- 32-bit integer (type **int**):
  - -2,147,483,648 to 2147483647
  - Every integer in that range can be represented
- 64-bit integer (type **long**):
  - –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- 32-bit floating point (type **float**):
  - ~1.2 $\times 10^{-38}$ to ~3.4 $\times 10^{38}$
  - 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 $\times 10^{-308}$ to ~1.8 $\times 10^{308}$

Precision options

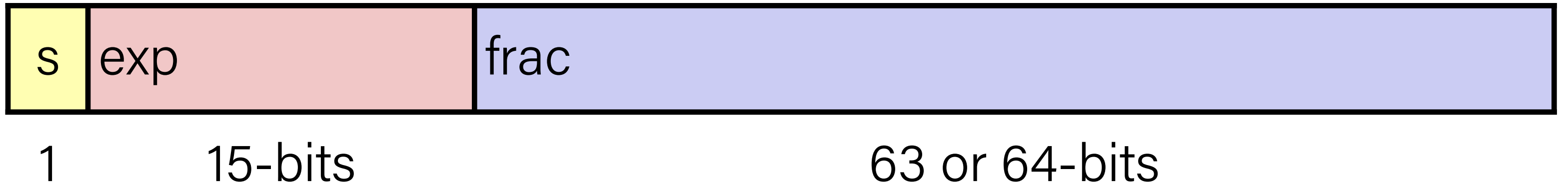
- Single precision: 32 bits



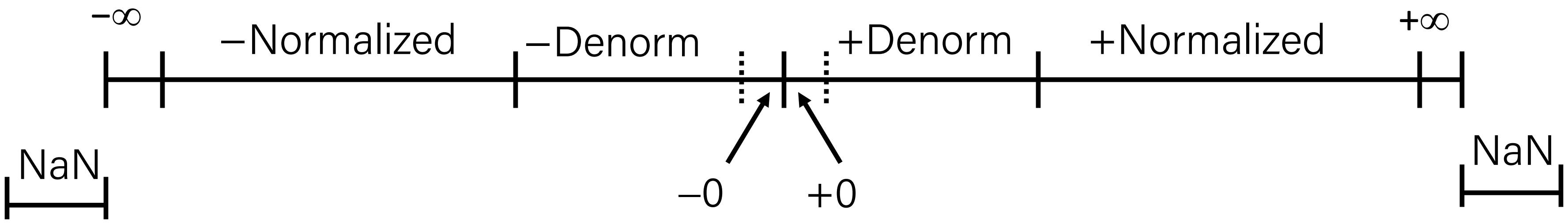
- Double precision: 64 bits



- Extended precision: 80 bits (Intel only)



Visualization: Floating Point Encodings



Lecture Plan

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

Tiny Floating Point Example

s	exp	frac
1	4-bits	3-bits

- 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 **`frac`**

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

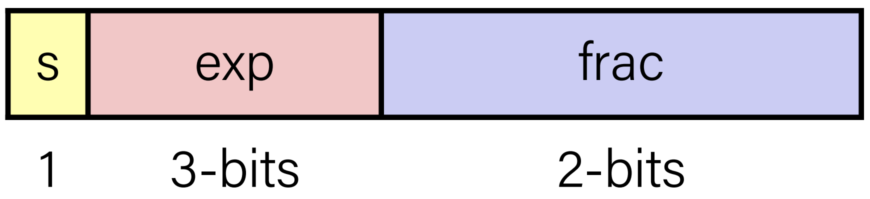
Dynamic Range (Positive Only)

$v = (-1)^s\, M\, 2^E$
n: $E = \text{Exp} - \text{Bias}$
d: $E = 1 - \text{Bias}$

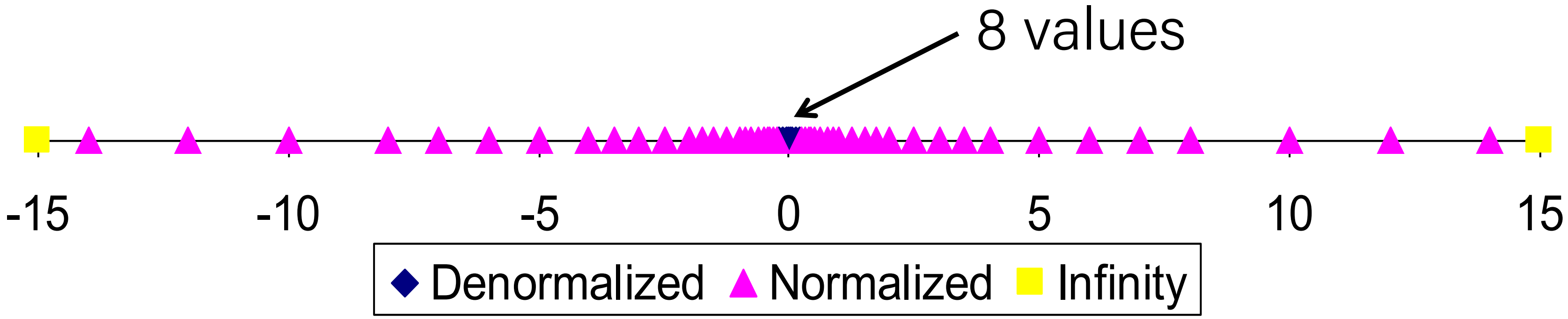
	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	closest to zero
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	largest denorm
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	smallest norm
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	closest to 1 below
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	closest to 1 above
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	largest norm
	0	1111	000	n/a	inf	

Distribution of Values

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

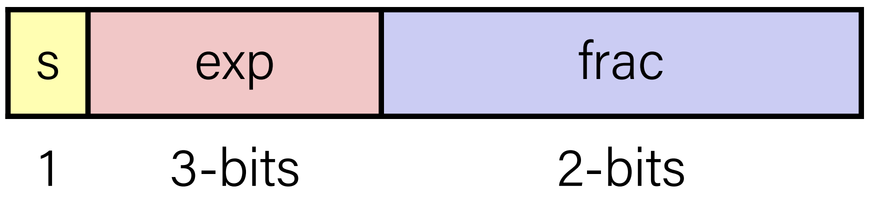


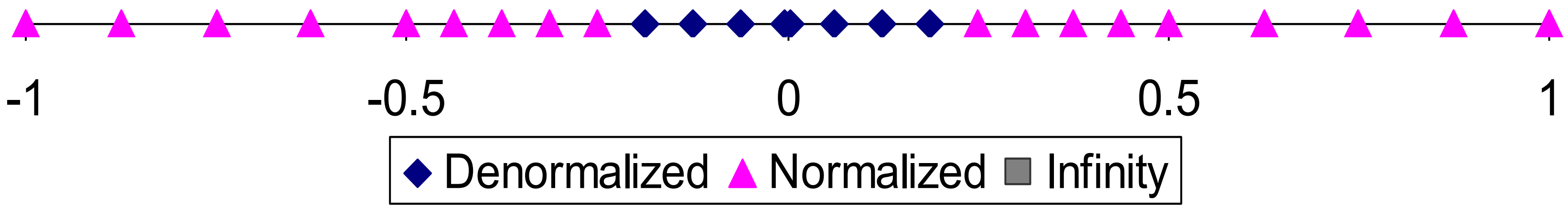
- Notice how the distribution gets denser toward zero.



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

- Floating Point
- 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.

```
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:

	31	30 … 23	22 … 0
3.14:	0	10000000	10010001111010111000011

	31	30 … 23	22 … 0
1e20:	0	11000001	01011010111100011101100

Floating Point Arithmetic

3.14:

31	30 – 23	22 – 0
0	10000000	10010001111010111000011

1e20:

31	30 – 23	22 – 0
0	11000001	01011010111100011101100

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

```
                       3.14
+   100000000000000000000.00
    100000000000000000003.14
```

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

Floating Point Arithmetic

Step 1: convert from base 10 to binary

What is 100000000000000000000003.14 in binary? Let's find out!

http://web.stanford.edu/class/archive/cs/cs107/cs107.1184/float/convert.html

1010110101111000111010111100010110101100011000100000000000000000011.0010001111010111000010100011…

Floating Point Arithmetic

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

10101101011110001110101111000101101011000110001000000000000000000011.0010001111010111000010100011…

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.

1010110101111000111010111100010110101100011000100000000000000000011.00100011110101110000010100011…

66 shifts -> 66 + 127 = 193

Floating Point Arithmetic

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

1010110101111000111010111100010110101100011000100000000000000000011.0010001111010111000010100011…

Sign bit is 0.

Floating Point Arithmetic

The binary representation for 1e20 + 3.14 thus equals the following:

31	30 … 23	22 … 0
0	11000001	01011010111100011101100

This is the **same** as the binary representation for 1e20 that we had before!

We didn’t have enough bits to differentiate between 1e20 and 1e20 + 3.14.

Floating Point Arithmetic

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

```
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. 1.2×10^6 ✔
- Still be able to compare quickly ✔
- Have more predictable over/under-flow behavior ✔

Lecture Plan

- Floating Point
- 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 ≤ 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

Floating Point Puzzles

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

```
int x = …;
float f = …;
double d = …;
```

Assume neither **d** nor **f** is NaN

`x == (int)(float) x`	False
`x == (int)(double) x`	True
`f == (float)(double) f`	True
`d == (float) d`	False
`f == -(-f);`	True
`2/3 == 2/3.0`	False
`d < 0.0  ⇒  ((d*2) < 0.0)`	True (OF?)
`d > f  ⇒  -f > -d`	True
`d * d >= 0.0`	True (OF?)
`(d+f)-d == f`	False

Floats 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

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

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

Additional Reading

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 esotoric 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

What Every Computer Scientist Should Know About Floating-Point Arithmetic,
David Goldberg, ACM Computing Surveys, 23(1), 1991