CS107 Lecture 17
Floating Point

reading:

B&O 2.4

Slides by Jerry Cain and Lisa Yan, who leveraged prior work by Nick Troccoli, Julie Zelenski, Marty Stepp, Cynthia Lee, Chris Gregg, and others.

How can a computer represent real 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 and (thought experiment) fixed point
- Floating Point: Normalized values
- **Joke Break**
- Floating Point: Special/denormalized values
- Floating Point Arithmetic

```
cp -r /afs/ir/class/cs107/samples/lectures/lect10 .
```

Plan For Today

- Representing real numbers and (thought experiment) fixed point
- Floating Point: Normalized values
- **Joke Break**
- Floating Point: Special/denormalized values
- Number representations in C
- Floating point arithmetic

Base 2 conversion

Convert the following number/fractions to base 10 (decimal) and base 2.

	Number/fraction	Base 10	Base 2
1.	`1/2`	`0.5`	`0.1`
2.	`5`		
3.	`9/8`		
4.	`1/3`		
5.	`1/10` (bonus)		

Base 2 conversion

Convert the following number/fractions to base 10 (decimal) and base 2.

	Number/fraction	Base 10	Base 2
1.	`1/2`	`0.5`	`0.1`
2.	`5`	`5.0`	`101.0`
3.	`9/8`	`1.125`	`1.001`
4.	`1/3`		
5.	`1/10` (bonus)		

Base 2 conversion

Convert the following number/fractions to base 10 (decimal) and base 2.

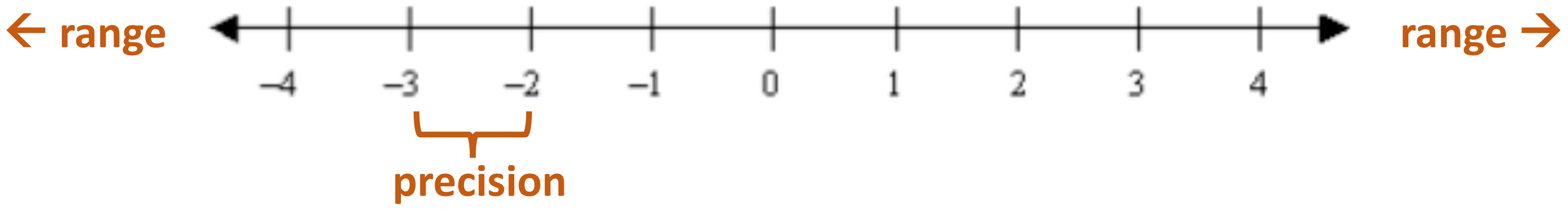
	Number/fraction	Base 10	Base 2
1.	`1/2`	`0.5`	`0.1`
2.	`5`	`5.0`	`101.0`
3.	`9/8`	`1.125`	`1.001`
4.	`1/3`	`0.3333…`	`0.01010101…`
5.	`1/10` (bonus)	`0.1`	`0.00011001100…`

Conceptual goal: How can we represent real numbers with a ***fixed*** number of bits?

Learning goal: Appreciate the IEEE floating point format!

Approximating real numbers

How can we represent real numbers with a ***fixed*** number of bits?



In the world of real numbers:

- The real number line extends forever (infinite **range**).
- Real numbers have infinite resolution (infinite **precision**).

In the base-2 world of computers, we must **approximate**:

- Each variable type is fixed width (`float`: 32 bits, `double`: 64 bits).
- Compromises are inevitable (**range** and **precision**). Like with `int`, we need to make choices about which numbers make the cut and which don't.

Thought experiment: Fixed point

Base 10, decimal case:

$5934.216123121\ldots_{10}$

$\Downarrow$

5 9 3 4 . 2 1 6 7

10^3 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3} 10^{-4}

Base 2, binary case:

$1011.0101010101\ldots_2$

$\Downarrow$

1 0 1 1 . 0 1 0 1

2^3 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3} 2^{-4}

- Decide on fixed granularity, e.g., 1/16
- Assign bits to represent powers from 2^3 to 2^{-4}

Thought experiment: Fixed point

Strategy evaluation

What values can be represented?

- Largest magnitude? Smallest? To what precision?

How hard to implement?

- How to convert `int` into 32-bit fixed point? 32-bit fixed point to `int`?
- Can existing integer ops (add, multiply, shift) be repurposed?

How well does this meet our needs?

Base 2, binary case:

1011.0101010101…$_2$

$$\Downarrow$$

$$\underset{2^3}{1}\quad \underset{2^2}{0}\quad \underset{2^1}{1}\quad \underset{2^0}{1}\quad .\quad \underset{2^{-1}}{0}\quad \underset{2^{-2}}{1}\quad \underset{2^{-3}}{0}\quad \underset{2^{-4}}{1}$$

- Decide on fixed granularity, e.g., 1/16
- Assign bits to represent powers from 2^3 to 2^{-4}

The problem with fixed point

Problem: We must fix where the decimal point is in our representation. This fixes our **precision**.

$$6.022e23 = \underbrace{11\ldots\ldots0}_{\text{79 bits}}.0$$

(base 10) (base 2)

$$6.626e\text{-}34 = 0.\underbrace{0\ldots\ldots01}_{\text{111 bits}}$$

To store both these numbers in the same fixed-point representation, the bit width of the type would need to be at least 190 bits wide!

Scientific notation to the rescue (1/2)

We have a need for **relative** rather than absolute precision.

- How much error/approximation is tolerable? Radius of atom, bowling ball, planet?

Consider for decimal values:

$$3{,}650{,}123 \rightarrow 3.65 \times 10^{6}$$

$$0.0000072491 \rightarrow \underbrace{7.25}_{\text{3 digits for mantissa (and round)}} \times 10^{-6} \leftarrow \text{1 digit for exponent}$$

- As a datatype, store mantissa and exponent separately
- Allocations of bits to exponent and mantissa (respectively) determines range and precision

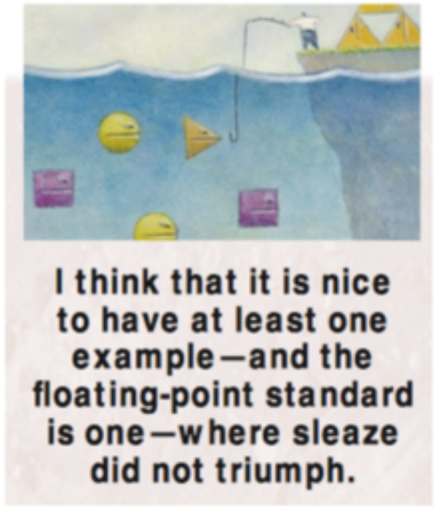
IEEE floating point

IEEE Standard 754

- Established in 1985 as a uniform standard for floating point arithmetic
- Supported by all major systems today
  Hardware: specialized co-processor vs. integrated into main chip

Driven by numerical concerns

- Behavior defined in mathematical terms
- Clear standards for rounding, overflow, underflow
- Support for transcendental functions (roots, trig, exponentials, logs)
- Hard to make fast in hardware
  Numerical analysts predominated over hardware designers in defining standard


I think that it is nice to have at least one example—and the floating-point standard is one—where sleaze did not triumph.

— Will Kahan (chief architect of standard)

Plan For Today

- Representing real numbers and fixed point
- Floating Point: Normalized values
- **Joke Break**
- Floating Point: Special/denormalized values
- Number representations in C
- Floating point arithmetic

IEEE Floating Point

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

$$V = (-1)^{s} \times M \times 2^{E}$$

Sign bit, s:
negative (s == 1)
positive (s == 0)

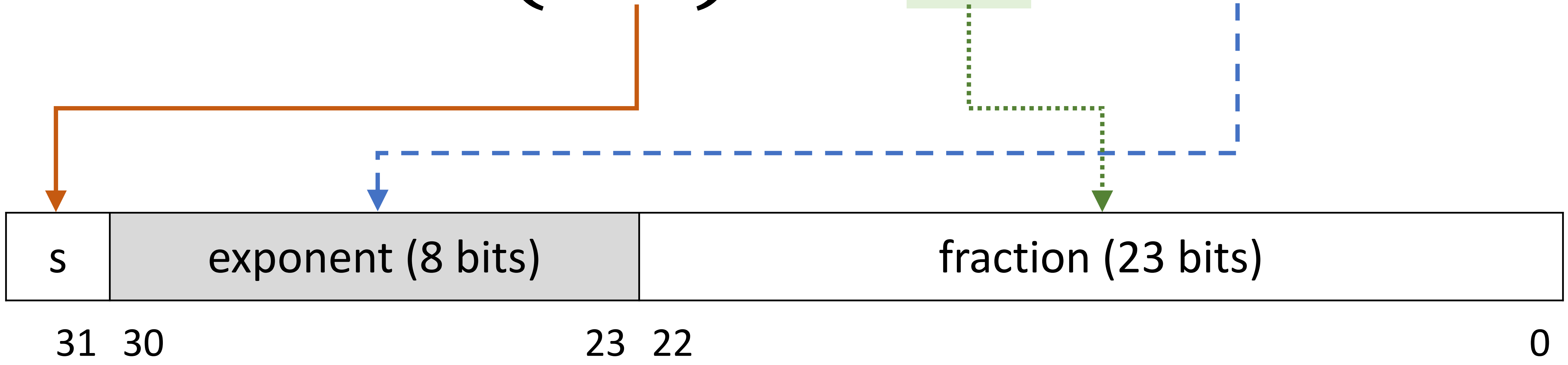
Mantissa, M :
Significant digits,
also called
significand

Exponent, E:
Scales value by
(possibly negative)
power of 2

IEEE Floating Point

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

$$V = (-1)^s \times M \times 2^E$$



⚠️Quirky! Exponent and fraction are not encoded as M and E in 2’s complement!

Example for the next few slides

What is the number represented by the following 32-bit `float`?

s	exponent (8 bits)	fraction (23 bits)
0	1000 0000	0100 0000 0000 0000 0000 000

Exponent

s	exponent (8 bits)	fraction (23 bits)

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

special

normalized

denormalized

Exponent: Normalized values

s	exponent (8 bits)	fraction (23 bits)

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

- Based on this table, how do we compute an exponent from a binary value?
- Why would this be a good idea? (hint: what if we wanted to compare two floats with >, <, =?)

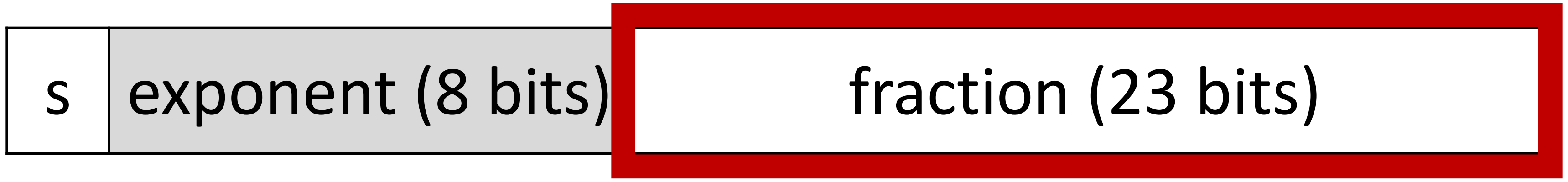
Exponent: Normalized values

s	exponent (8 bits)	fraction (23 bits)

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

- Not 2’s complement
- E = exponent – **bias**, where `float` **bias** = $2^{8-1} - 1 = 127$
- Exponents are sequentially represented starting from 000…1 (most negative) to 111…10 (most positive).
- Bit-level comparison is fast!

Fraction: Normalized values



We could just encode whatever M is in the fraction field (f). But there's a trick we can use to make the most out of the bits we have…

$$M = 1.[\text{fraction bits}]$$

What???

Scientific notation to the rescue (2/2)

Correct scientific notation:
In the mantissa, always keep one non-zero digit to the left of the decimal point.

For base 10:

```
42.4     x 10^5 → 4.24  x 10^6
324.5    x 10^5 → 3.245 x 10^7
  0.624  x 10^5 → 6.24  x 10^4
```

$42.4 \times 10^5 \rightarrow \mathbf{4}.24 \times 10^6$

$324.5 \times 10^5 \rightarrow \mathbf{3}.245 \times 10^7$

$0.624 \times 10^5 \rightarrow \mathbf{6}.24 \times 10^4$

For base 2:

$10.1 \times 2^5 \rightarrow \mathbf{1}.01 \times 2^6$

$1011.1 \times 2^5 \rightarrow \mathbf{1}.0111 \times 2^8$

$0.110 \times 2^5 \rightarrow \mathbf{1}.10 \times 2^4$

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

Fraction: Normalized values

s	exponent (8 bits)	fraction (23 bits)

We could just encode whatever M is in the fraction field (f). But there's a trick we can use to make the most out of the bits we have…

$$M = 1.[\text{fraction bits}]$$

- An "implied leading 1" representation
- Means: we get one additional bit of precision for free!

Thanks, Will!

Example from the past few slides

What is the number represented by the following 32-bit `float`?

s	exponent (8 bits)	fraction (23 bits)
0	1000 0000	0100 0000 0000 0000 0000 000

Subtract bias ($2^{8-1} - 1 = 127$)

Add implicit 1

$$E = 128 - 127 = 1$$

$$M = (1.01)_2 \quad \text{(base 2)}$$

$$= 1 + 0\times2^{-1} + 1\times2^{-2} = 1.25 \quad \text{(base 10)}$$

$$\boxed{V = (-1)^0 \times 1.25 \times 2^1 = 2.5}$$

Practice #1

$V = (-1)^s \times M \times 2^E$

s	exponent (8 bits)	fraction (23 bits)
0	0111 1110	0000 0000 0000 0000 0000 000

1. Is this number:
   - A. Greater than 0?
   - B. Less than 0?
2. Is this number:
   - A. Less than -1?
   - B. Between -1 and 1?
   - C. Greater than 1?
3. Bonus: What is the number?

Practice #1

$V = (-1)^s \times M \times 2^E$

s	exponent (8 bits)	fraction (23 bits)
0	0111 1110	0000 0000 0000 0000 0000 000

1. Is this number:
   A. Greater than 0?
   B. Less than 0?

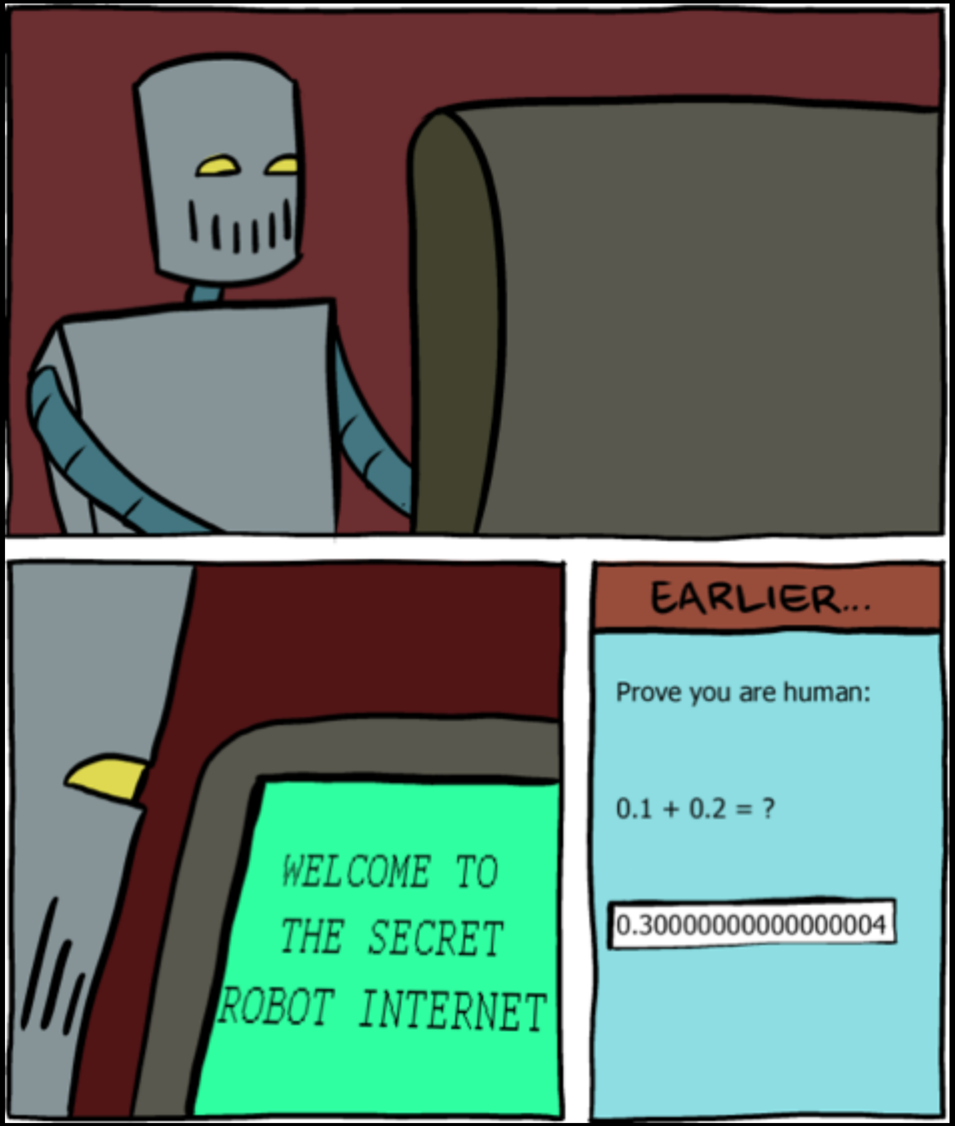
2. Is this number:
   A. Less than -1?
   B. Between -1 and 1?
   C. Greater than 1?

3. Bonus: What is the number? $(-1)^0 \times 1.0 \times 2^{-1} = 0.5$

Plan For Today

- Representing real numbers and fixed point
- Floating Point: Normalized values
- **Joke Break**
- Floating Point: Special/denormalized values
- Number representations in C
- Floating point arithmetic

Joke break



https://www.smbc-comics.com/comic/2013-06-05

Slightly off from the real float 0.3 ☺

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

Plan For Today

- Representing real numbers and fixed point
- Floating Point: Normalized values
- **Joke Break**
- Floating Point: Special/denormalized values
- Number representations in C
- Floating point arithmetic

Reserved exponent values

s	exponent (8 bits)	fraction (23 bits)

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

special

normalized

denormalized

All zeros: Zero + denormalized floats

Zero (`+0`, `-0`)

s	exponent (8 bits)	fraction (23 bits)
any	`0000 0000`	`all zeros`

Why would two zeros be okay?

Denormalized floats:

s	exponent (8 bits)	fraction (23 bits)
any	`0000 0000`	`any nonzero`

- Smallest normalized exponent: $E = 1 - \text{bias} = -126$
- Mantissa has **no leading zero**: $M = 0.[\text{fraction bits}]$

$$V = (-1)^s \times M \times 2^E$$

Why would we want so much precision for tiny numbers?

All zeros: Zero + denormalized floats

Zero (`+0`, `-0`)

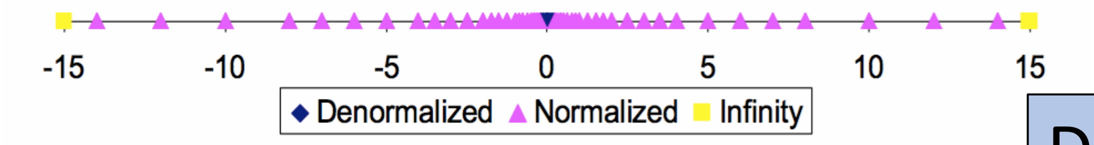
s	exponent (8 bits)	fraction (23 bits)
any	`0000 0000`	`all zeros`

Denormalized floats:

s	exponent (8 bits)	fraction (23 bits)
any	`0000 0000`	`any nonzero`

- Smallest normalized exponent: $E = 1 - \text{bias} = -126$
- Mantissa has **no leading zero**: $M = 0.[\text{fraction bits}]$

$$V = (-1)^s \times M \times 2^E$$



Denormalized values enable gradual **underflow** (too-small-to-represent floats).

All ones: Infinity and NaN

Infinity (`+inf`, `-inf`)

s	exponent (8 bits)	fraction (23 bits)
any	`1111 1111`	`all zeros`

Why would we want to represent infinity?

Not a number (**NaN**):

s	exponent (8 bits)	fraction (23 bits)
any	`1111 1111`	`any nonzero`

Computation result that is an invalid mathematical real number.

What kind of mathematical computation would result in a **non-real** number? (hint: square root)

All ones: Infinity and NaN

Infinity (`+inf`, `-inf`)

s	exponent (8 bits)	fraction (23 bits)
any	`1111 1111`	`all zeros`

Floats have built-in handling of overflow: infinity + anything = infinity.

Not a number (**NaN**):

s	exponent (8 bits)	fraction (23 bits)
any	`1111 1111`	`any nonzero`

Computation result that is an invalid mathematical real number.

Examples: `sqrt(x)` (i.e., $\sqrt[2]{x}$), where x is negative, $\frac{\infty}{\infty}$, $\infty + (-\infty)$, etc.

Questions?

Plan For Today

- Representing real numbers and fixed point
- Floating Point: Normalized values
- **Joke Break**
- Floating Point: Special/denormalized values
- Number representations in C
- Floating point arithmetic

Skipping Numbers

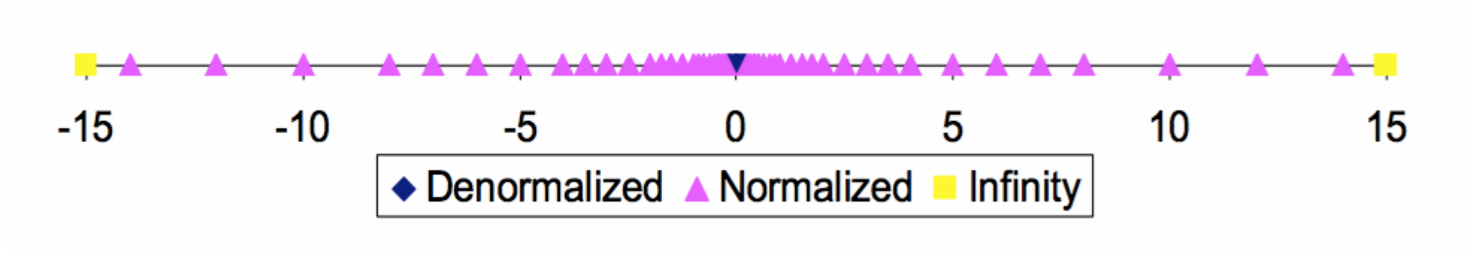
We said that it's not possible to represent *all* real numbers using a fixed-width representation. What does this look like?

Float Converter

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

Floats and Graphics

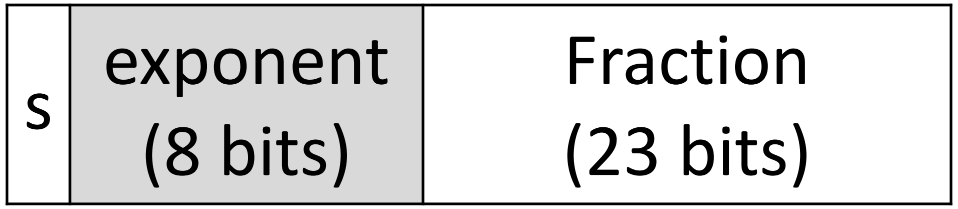
- https://www.shadertoy.com/view/4tVyDK



float and double

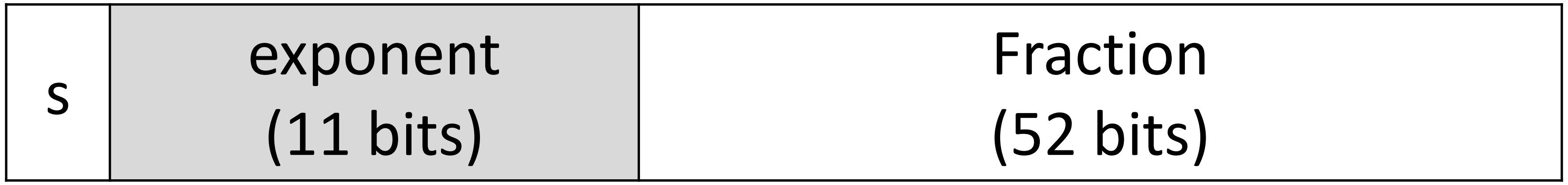
`float` (32 bits)

- 8-bit exponent ranges from -126 to +127, $2^{127} = 10^{37}$



`double` (64 bits)

- 11-bit exponent ranges from -1022 to +1023, $2^{1023} = 10^{308}$

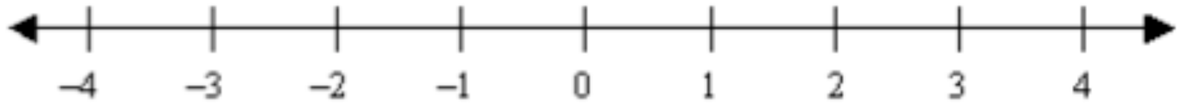


float vs int

32-bit integer (type **int**): :
–2,147,483,648 to 2147483647

64-bit integer (type **long**):
–9,223,372,036,854,775,80
to 9,223,372,036,854,775,807

All integers in these ranges can be represented.



32-bit floating point (type **float**):

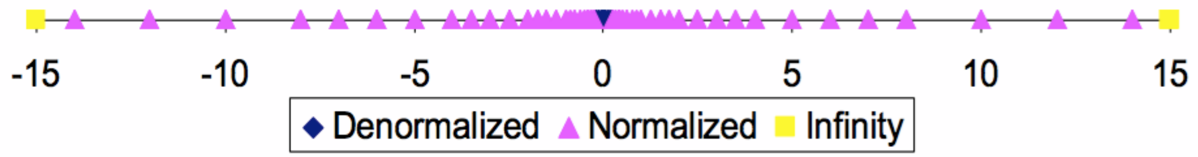
~1.7 x10^{-38} to ~3.4 x10^{38} (+ negative range)

64-bit floating point (type **double**):

~9 x10^{-307} to ~1.8 x10^{308} (+ negative range)

(normalized float/double ranges)

Not all numbers can be represented. **Gaps can get quite large**: larger the exponent, larger the gap between successive fraction values.



Plan For Today

- Representing real numbers and fixed point
- Floating Point: Normalized values
- **Joke Break**
- Floating Point: Special/denormalized values
- Number representations in C
- Floating point arithmetic

Key (floating) points

Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)

But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often unwise.

Key (floating) points

Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)

But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often unwise.

Lisa's Official Guide To Making Money

It's easy!

FAST!

You can lose money, too!

Demo: Float Arithmetic

Try it yourself:

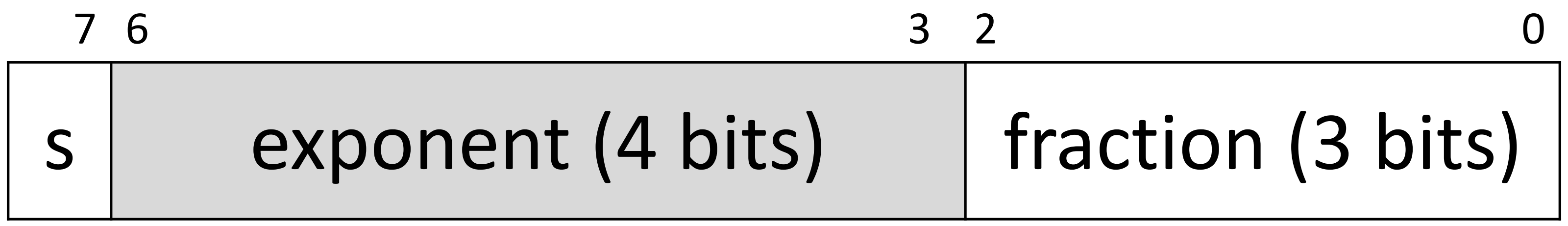
```
./bank 100 1         # deposit
./bank 100 -1        # withdraw
./bank 100000000 -1 # make bank
./bank 16777216 1   # lose bank
```

`bank.c`

Why is 2^{24} special?

Introducing "Minifloat"

For a more compact example representation, we will use an 8 bit "minifloat" with a 4 bit exponent, 3 bit fraction and bias of 7 (note: minifloat is just for example purposes, and is not a real datatype).



Floating Point Arithmetic

In minifloat, with a balance of $128, a deposit of $4 **would not be recorded** at Lisa's Bank. Why not?

$$
\begin{array}{r}
128 \\
+\quad 4 \\
\hline
128?
\end{array}
$$

Let's step through the calculations to add these two numbers (note: this is just for understanding; real float calculations are more efficient).

Floating Point Arithmetic

128	0	1110	000
+ 4	0	1001	000

```
    128.00            1.00 x 2^7
+     4.00        + 1.00 x 2^2
    132.00
```

aligned

not aligned

Float arithmetic (at a high level):

x floatop y = Round(x op y)

1. Manipulate significand and exponents independently to **align** **(FPU)**
2. Compute **exact** result (x op y) **132**
3. **Round** and put in floating point **(next slide)**

Practice #2

7	6 3	2 0
s	exponent (4 bits)	fraction (3 bits)
?	????	???

What is 132 as a minifloat?

$$V = (-1)^S \times M \times 2^E$$

$$E = \text{exponent} - \text{bias}$$

$$M = 1.[\text{fraction bits}]$$

$$\text{bias} = 2^{4-1} - 1 = 7$$

Practice #2

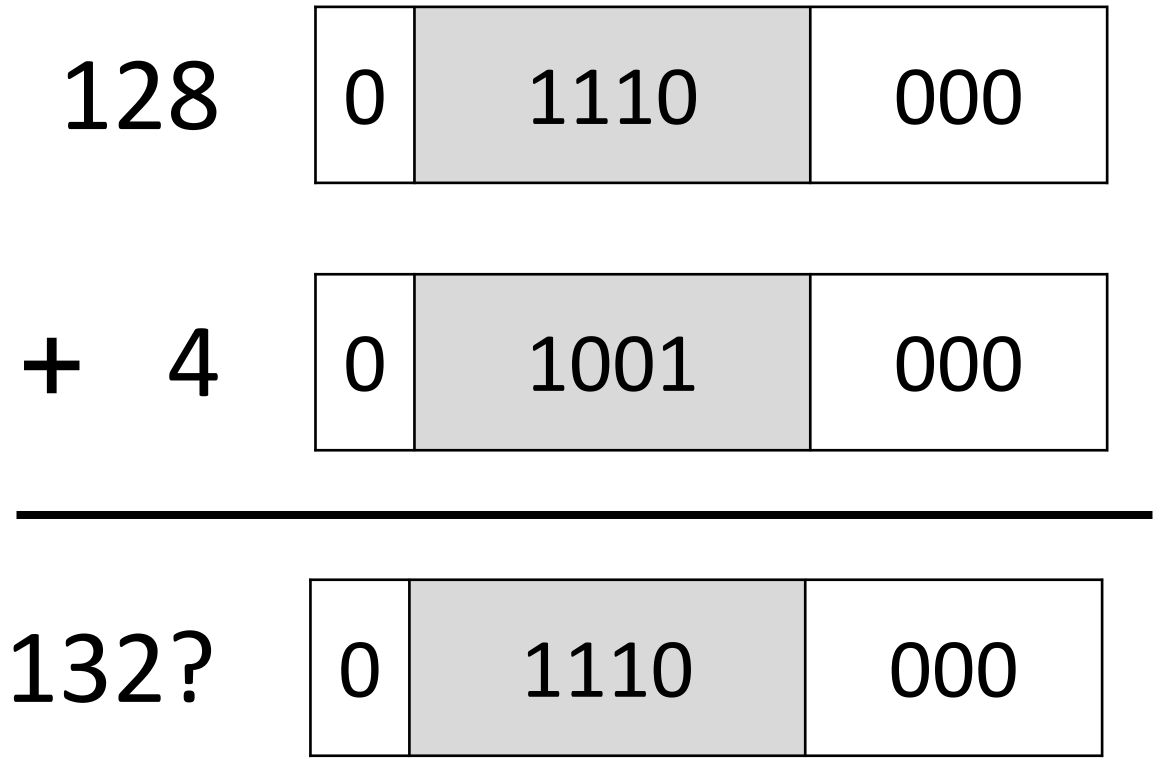
7 s	6 exponent (4 bits) 3	2 fraction (3 bits) 0
0	1110	000

$$V = (-1)^s \times M \times 2^E$$
$$E = \text{exponent} - \text{bias}$$
$$M = 1.[\text{fraction bits}]$$
$$\text{bias} = 2^{4-1} - 1 = 7$$

What is 132 as a minifloat?

1. Convert to binary. $(1000\ 0100)_2$
2. Convert to scientific notation $M \times 2^E$. $(1.0000100)_2 \times 2^7$
3. Determine exponent $= E +$ bias. $7 + 7 = 14 = (1110)_2$
4. Determine fraction, with rounding. $(1.\underline{000}0100)_2$
5. Determine sign s. 0 (positive)

Approximation error is inevitable



We didn’t have enough bits to differentiate between 128 and 132.

Approximation error is inevitable

128	0	1110	000
+ 4	0	1001	000
132	0	1110	000

We didn’t have enough bits to differentiate between 128 and 132.

Another way to corroborate this: the *next-largest minifloat* that can be represented after 128 is **144**. 132 isn’t representable!

144:	0	1110	001

Key Idea: the smallest float “hop” we can take is to adjust the fractional component by 1.

Key (floating) points

Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)

But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often inaccurate.

Floating Point Arithmetic

Is this just over/underflowing? It turns out it’s more subtle.

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

Key (floating) points

Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)

But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often ~~unwise~~ inaccurate.

Demo: Float Equality

`float_equality.c`

Floating point in other languages

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

- http://geocar.sdf1.org/numbers.html

Key (floating) points

Approximation and rounding is inevitable.

Single operations are commutative, but sequence is not associative.

(a + b) equals (b + a)

But (a + b) + c may not equal a + (b + c)

Equality comparison operations are often ~~unwise~~ inaccurate.

Let’s Get Real Representation

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

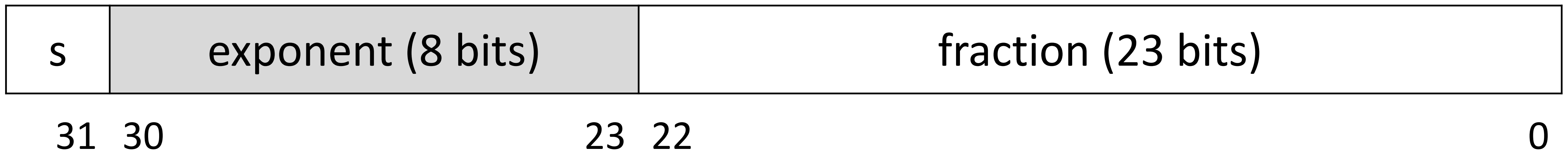
✓ Represent widest range of numbers possible

✓ Flexible “floating” decimal point

✓ Still be able to compare quickly

✓ Represent scientific notation numbers, e.g. 1.2 x 10^6

✓ Have more predictable overflow behavior


s
exponent (8 bits)
fraction (23 bits)
31
30
23
22
0