13

Floating Point Arithmetic

Want: Something like the real numbers... in a computer

Have: Integers, made of bits

$$23 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

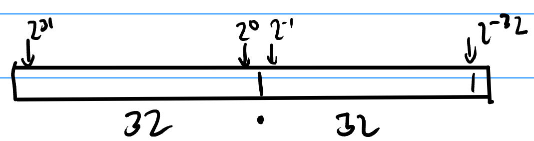
How should we even represent fractions?

Idea: Keep going down past exponent zero

$$23.625 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \underline{+ 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}}$$

So: Could store
- a fixed number of bits with exponents >= zero
- a fixed number of bits with exponents < zero

Suppose we use a 64-bit integer, with 32 bits >= 1 and 32 bits < 1.



This is called fixed-point arithmetic.

What is the smallest number we can represent?

$$2^{-32} \approx 10^{-10}$$

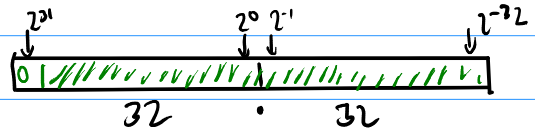
What is the biggest number we can represent?

$$2^{31} + 2^{30} + \cdots + 2^1 + 2^0 + 2^{-1} + \cdots + 2^{-32} \approx 10^9$$

What's our range then?

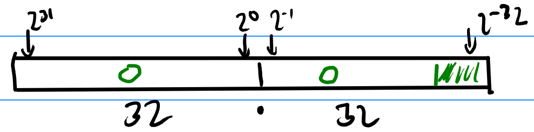
$\sim 10^{-10} \ldots \quad 10^9 \leftarrow$ not a lot

How many accurate decimal digits do we have in a number near 10^9 ?



about 19

How many accurate digits do we have in a number near 10^{-9} ?



$2^{-29} \approx 10^{-9}$ Rounding starts at $2^{-33} \approx 10^{-10}$

→ about one digit (!)

What is the problem here?

Fixed point offers much more relative accuracy for large numbers than small numbers.

Also, the overall magnitude of the representable numbers is limited.

Idea: Set a few bits aside to store the largest exponent. How?

$$2^3 + 2^2 + 2^0 = (1101)_2 = (1.101)_2 \cdot 2^3$$

Instead of "fixing" where the 'decimal' point goes, we make its location "floating".

This is called a "floating point" number.

It has two distinct parts that need to be stored:

$$\underbrace{(1.101)_2}_{\text{"significand"}} \cdot 2^{\overbrace{3}^{\text{"exponent"}}}$$

Write these here as a floating point number:

$$12.25 = (1100.01)_2 = (1.10001)_2 \cdot 2^3$$

$$0.125 = (0.001)_2 = (1.0)_2 \cdot 2^{-3}$$

$$2^{50} = (1.0)_2 \cdot 2^{50}$$

$$2^{-100} = (1.0) \cdot 2^{-50}$$

The first digit of the significand seems to always be a one. Do we need to store it?

No, we can just leave it out.

$$(1.\underbrace{10001}_{\text{just store this (not the leading 1)}})_2 \cdot 2^{\underbrace{3}_{\text{and this}}}$$

This is called the "implied one" in floating point representation.

How is zero represented?

$1.000 \cdot 2^{-???}$ ← The "implied one" means our FP number is $\neq 0$. (regardless of the exponent)

To fix this, we designate a "special" value of the exponent (one of the extreme values, probably) to mean "turn off the implied one."

Suppose this special exponent value is -1023.

Then 0 is stored as: $\underbrace{0000}_{\text{significand}} \qquad \underbrace{-1023}_{\text{exponent}}$

And because the (entire) significand (with the "implied one" turned off) is now legitimately zero, we've found a way to represent zero.

What happens if you run out of digits to store in your significand?

Suppose, for the sake of argument, that we store four bits of the significand. But the true number we would like to represent has seven binary digits.

$$(1110.011)_2 = (1.\underbrace{1100}_{}11)_2 \cdot 2^3$$

This is how much we can store.

Revolutionary idea (not really): Round the result.

$$(1110.011)_2 = (1.\underbrace{1100}_{}11)_2 \cdot 2^3 \qquad (1.\underbrace{1101}_{})_2 \cdot 2^3$$

What is a denormal number?

Suppose the smallest exponent you can represent is -1022.

How would you represent 2^{-1025}?

$$2^{-1025} = 1.\underbrace{\qquad ? \qquad}_{} \cdot 2^{-1022}$$

already as small as they get

Idea: Use our "special" exponent value of -1023 to turn off the "implied one."

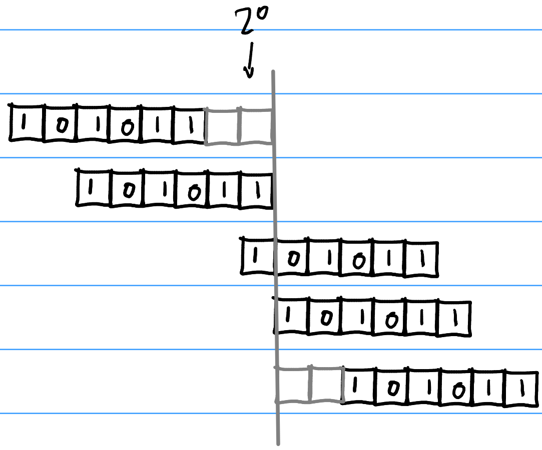
	Significand	Exponent
Intended meaning:	$(0.\underbrace{001}_{})_2$	-1022
Stored values:	$(001)_2$	-1023

means: use exponent -1022, but also turn off the "implied one."

Numbers that have the leading zero turned off are called "denormal".

Zero was our first example of a denormal number.

What is the...



	exponent?	significand?	value?
	7	$(1.01011)_2 = 1.34375$	$1.34375 \cdot 2^7$
	5	—"—	$1.34375 \cdot 2^5$
	0	—"—	$1.34375 \cdot 2^0$
	-1	—"—	$1.34375 \cdot 2^{-1}$
	-3	—"—	$1.34375 \cdot 2^{-3}$

In our 64-bit example:

- 1 bit for sign (+/-)
- 11 bits for largest exponent
- 52 bits for "bits"

This is called "double precision".

Exponent ranges from -1023 to 1024

but: extreme values are special So really:

-1022 to 1023

What is (very roughly) the smallest number we can represent?

$$2^{-1022} \approx 10^{-308}$$

What is (very roughly) the largest number we can represent?

$$2^{1023} \approx 10^{307}$$

How many accurate decimal digits do we have in the largest representable number?

Rounding starts at $2^{1023-52} \approx 10^{292}$

$307 - 292 \approx 15$ digits

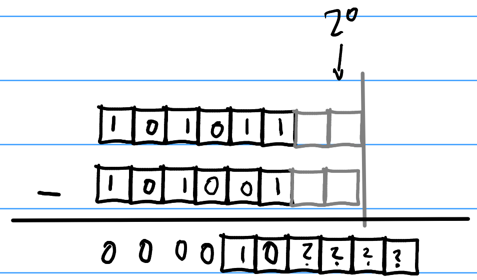
How many accurate decimal digits do we have in the smallest representable number?

Rounding starts at $2^{-1022-52} \approx 10^{-324}$

$-308-(-324) \approx 15$ digits

Same relative accuracy for numbers of every magnitude: Yay!

So what could possibly go wrong?



How many accurate (binary) digits are there in the above result?

2/6 ← not good

Called "catastrophic cancellation"

Demo: Catastrophic Cancellation

Demo: Picking apart a floating point number

Demo: Floating point vs. program logic