University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

# ECE 120: Introduction to Computing

## Fixed- and Floating-Point Representations

## In Binary, We Have A Binary Point

Let's talk about representations.

In decimal, we have a **decimal point**.

tenths
hundredths
3.1415...

In binary, we have a **binary point**.

$2^{-1}$'s place
$2^{-2}$'s place
11.001001...
$2^{-3}$'s place

## Fixed-Point Representations Support Fractions

If we need fractions,
- we can use a **fixed-point representation**
- in which some number of bits
- come after the binary point.

For example, with 32 bits:

| integer part (16 bits) | fractional part (16 bits) |
|---|---|

Some signal processing and embedded processors use fixed-point representations.

## What about Real Numbers?

A question for you:

**Do we need anything else to support real numbers?**

Note: Saying "yes" on the basis that there are uncountably many* real numbers is not a good answer. Integers are also infinite, and 2's complement is sufficient for practical use.

* An infinite number for each integer.

## Isn't Fixed-Point Good Enough?

Let's do a calculation.

**32-bit 2's complement** has what range?

That's right: **[-2,147,483,648, 2,147,483,647]**.

You DID all know that, right?

I didn't. I usually write **[$-2^{31}$, $2^{31} - 1$]**.

Let's write banking software to count pennies.

2,147,483,647 pennies is **$21,474,836.47**.

Anyone here have more? If not, we're done.

If so, use **64-bit**. You don't have that much!





## Anyone Here Taking Chemistry?

But maybe you want to do your Chemistry homework?

You may need **Avogadro's number**.

Anyone remember it? **$6.022 \times 10^{23}$ / mol**

Sure. No problem.

$10^3$ is around $2^{10}$, so 80 bits should work.

Who can tell me Avogadro's number to 80 bits (the first 24 decimal digits will do)?





## Wikipedia May Not Help as Much as You Think!

Last I checked (July 2016!), the best known experimental value was

**$6.022140858 \times 10^{23}$ / mol**

That's only 10 digits.

So you have some serious Chemistry research to get done for your next homework!

Good luck!

Maybe we can just be close?





## What about Physics?

Some have Quantum Mechanics homework?

Your computer will need **Planck's constant**.

What is it again? **$6.626 \times 10^{-27}$ erg-sec***

Ok. Another 90 bits after the binary point.

170 bits total.

Don't forget to find another 90 bits (27 more decimal digits) for Avogadro.

*Use ergs, not Joules; we'll need fewer bits!

# We Need More Dynamic Range, Not More Precision

Do we really need **170 bits** of precision?

Do we really need to specify the first **51 significant figures** for Avogadro's number?

Of course not!

But we do need 170 bits of range.

**We need to be able to express both tiny numbers and huge numbers.**

# Develop a Representation Based on Scientific Notation

Let's borrow another representation from humans: scientific notation.

sign exponent

$$\mathbf{+\underbrace{6.022}_{\text{mantissa/significant figures (precision)}} \times 10^{23}}$$

The human representation has three parts.

# Modern Computers Use Standard Floating-Point

**Modern digital systems implement the IEEE 754 standard for floating-point.**

A single-precision floating-point number consists of 32 bits:

| sign (1 bit) | exponent (8 bits) | mantissa (23 bits) |
|---|---|---|

What value does a bit pattern represent?

First, let me ask you a question…

# What Values Can a Leading Digit Take?

A question for you:

**In the canonical form of scientific notation, what are the possible values of the leading digit?**

This one

$$\mathbf{-4.123 \times 10^{45}}$$

Any digit? 0-4? 1-7?

**1-9 (not 0)**. Change exponent as needed.

## What Values Can a Leading Digit Take?

Another question for you:

**Same question, but now in binary.**

**1 (not 0)**. Change exponent as needed.

And one more:

**How many bits do we need to store one possible answer?**

The leading 1 is implicit in binary (0 bits)!





## How to Calculate the Value of a Floating-Point Bit Pattern

The value represented by an IEEE single-precision floating-point bit pattern is…

| sign (1 bit) | exponent (8 bits) | mantissa (23 bits) |
|---|---|---|

$$(-1)^{\textbf{sign}}\ 1.\underline{\textbf{mantissa}} \times 2^{(\textbf{exponent} - 127)}$$

Convert the exponent to decimal as if it were unsigned before subtracting 127.





## Except that Exponents 0 and 255 have Special Meanings

That's almost correct. But **exponents 0 and 255 have special meanings**:

- 255 can mean **infinity** or **not-a-number** (**NaN**).
- 0 is a **denormalized** number: the leading implicit "1" is replaced with "0" (with power $2^{-126}$), allowing the representation to capture numbers closer to 0.

Except for the fact that **the bit pattern of all 0s means 0**, these aspects are beyond the scope of our class.





* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

## Exponent 255 is Used for Infinity and NaN

Exponent 255

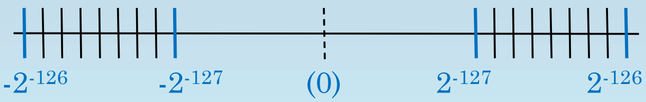
- Mantissa 0
  - Sign 0: **Positive infinity**
  - Sign 1: **Negative infinity**
- Non-zero mantissa: **NaN** (Not a Number)

These special values allow the representation to have 'correct' answers to some problems (such as 42.0 / 0.0) and to silently track the impact of missing values and incorrect computation (such as Infinity * 0).

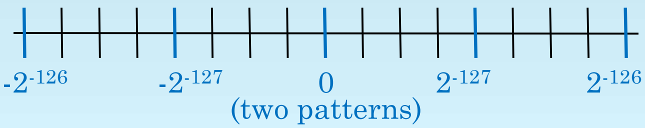
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

## Denormalization Allows Numbers Closer to 0 (and 0)

Without denormalized numbers, we have (shown with a 3-bit mantissa)…




Denormalization puts these patterns closer to 0 and gives two patterns for 0:








## Converting to a Floating-Point Bit Pattern

Conversion from decimal to IEEE floating-point is not too hard:

1. Convert to binary.
2. Change to scientific notation (in binary).
3. Encode each of the three parts.





## Use a Polynomial to Convert a Fraction to Binary

To convert a fraction **F** to binary, remember that **a fraction also corresponds to a polynomial**:

$$\mathbf{F = a_{-1}2^{-1} + a_{-2}2^{-2} + a_{-3}2^{-3} + a_{-4}2^{-4} + \ldots}$$

If we multiply both sides by 2

- the left side can only be $\geq 1$
- if $\mathbf{a_{-1} = 1}$

We can then subtract $\mathbf{a_{-1}}$ from both sides

and repeat to get $\mathbf{a_{-2}}$, $\mathbf{a_{-3}}$, $\mathbf{a_{-4}}$, and so forth.





## Example of Finding a Floating-Point Bit Pattern

For example, let's say that we want to find the bit pattern for **5.046875**.

We first write **5** in binary: **101**.

Now we need to convert the fraction

**F = 0.046875**.

$0.046875 \times 2 = 0.09375$ (< 1, so $\mathbf{a_{-1} = 0}$)

$0.09375 - 0 = 0.09375$

## Example of Finding a Floating-Point Bit Pattern

Start with 0.09375.

$0.09375 \times 2 = 0.1875$ (< 1, so $\mathbf{a_{-2} = 0}$)

$0.1875 - 0 = 0.1875$

$0.1875 \times 2 = 0.375$ (< 1, so $\mathbf{a_{-3} = 0}$)

$0.375 - 0 = 0.375$

$0.375 \times 2 = 0.75$ (< 1, so $\mathbf{a_{-4} = 0}$)

$0.75 - 0 = 0.75$





## Example of Finding a Floating-Point Bit Pattern

Start with 0.75.

$0.75 \times 2 = 1.5$ (so $\mathbf{a_{-5} = 1}$)

$1.5 - 1 = 0.5$

$0.5 \times 2 = 1$ (so $\mathbf{a_{-6} = 1}$)

$1 - 1 = 0$ (done)

Putting the bits together, we find

$$\mathbf{F = 0.046875_{10} = 0.000011_2}$$





## Example of Finding a Floating-Point Bit Pattern

Now we have converted to binary:

$$\mathbf{5.046875_{10} = 101.000011_2}$$

In binary scientific notation, we have

$$\mathbf{+\ 1.01000011 \times 2^2}$$

And, in single-precision floating point,

- the sign bit is **0**,
- the exponent is 2+127 = 129 = **10000001**,
- and the mantissa is **01000011…** (no leading 1, and 15 more 0s afterward).





## Tricky Questions about Floating-Point

A question for you:

**What is $2^{-30} + (1 - 1)$?**

Quite tricky, I know. But yes, it's $\mathbf{2^{-30}}$.

Another question for you:

**What is $(2^{-30} + 1) - 1$?**

That's right. It's **0**.

At least it is with floating-point.

# Floating-Point is Not Associative

Why?

Our first sum was $(2^{-30} + 1)$.

To hold the integer 1, the bit pattern's exponent must be $2^0$.

But, the mantissa for single-precision floating point has only **23 bits**.

And thus represents powers down to $2^{-23}$.

The $2^{-30}$ term is lost, giving $(2^{-30} + 1) = 1$.

**So** $2^{-30} + (1 - 1) \neq (2^{-30} + 1) - 1$.