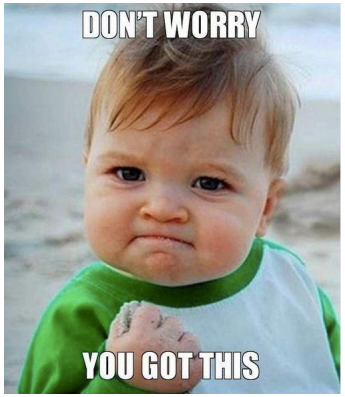
Up next

Midterm

- Next Friday in class
- Exams page on web site has info + practice problems
- Excited for you to rock the exams like you have been the assignments!



Today's lecture

- Back to numbers, bits, data representation
- How to represent real values
- The wonders of IEEE floating point

To follow

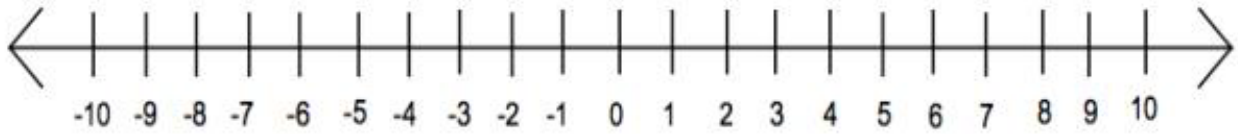
- Communing with our system in its native language (assembly) yeah!

Limits of finite representation

- **Computers work with finite storage**
  - **Typically 1/2/4/8 bytes at a time**
  - **Each type has fixed-size width, this limits representable range**
    - Some values make the cut and others don't…
  - **Budget only goes so far — how do we spend it?**
    - Which values can be represented as **char**? as **int**? as **long**? What about **signed** versus **unsigned**?
  - **Treatment of values outside range?**
    - Approximate? truncate? saturate? wrap? raise error?
- **What is "native" vs what can be synthesized**
  - **C gives transparent view of system primitives**
  - **Can create new abstractions that extend/layer**
    - Saturating addition
    - Python's "infinite"integer
    - C++ complex numbers

Representation => choices

- **Representing the infinite in the finite**
  - Real number line extends forever and has infinite resolution
  - Only fixed number of "breadcrumbs" to place on number line
    - Where do you put them?
- **What are the goals/priorities?**
  - Rational vs irrational
  - Widest possible range of magnitude?
  - Highest precision?
  - Negative vs positive, signed/unsigned?
  - Evenly spaced or …?



Thought experiment : fixed point

A possible representation for real values

Start with int representation (binary polynomial)

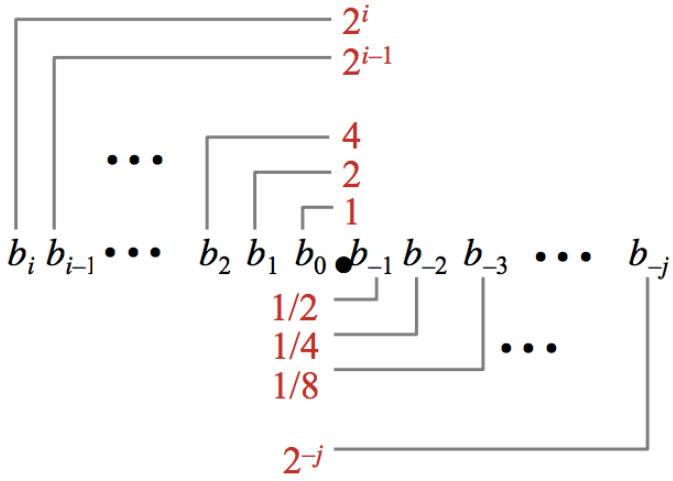
Remember: each bit represents a power of 2 from 2^{31} to 2^0

Decide on fixed granularity — e.g 1/128

Assign bits to instead represent powers from 2^{24} to 2^{-7}

```
0000 0000 0000 0000 0000 0100 1.010 0000   =  9.25
```

This strategy is known as "fixed point"



Evaluate as strategy

What values can be represented?

Largest magnitude? Smallest? To what precision?

How hard to implement?

How to convert int into fixed point? fixed point to int?

Can existing integer ops (add, multiply, shift) be repurposed?

How well does this meet our needs?

Scientific notation to the rescue!

- **Need for relative rather than absolute precision**
  - Radius of atom vs bowling ball vs planet: how much error/approximation is tolerable?
- **Consider this example for decimal values**
  - 3,650,123 .0000072491
  - $3.65 * 10^{7}$ $7.25 * 10^{-6}$
  - Store mantissa and exponent separately
  - 4 decimal digits, apportion as 1 for exponent, 3 for mantissa
    - Round the mantissa to nearest representable
  - Division of bits into exponent/mantissa determines range/precision
    - More bits to exponent = larger range
    - More bits to mantissa = higher precision (lower relative error)
- **Let's do that again, but this time in binary**

IEEE floating point

IEEE Standard 754

Established in 1985 as uniform standard for floating point arithmetic

Before that, many idiosyncratic formats

Supported by all major systems today

Software: emulation

Hardware: specialized co-processor vs. integrated into main chip

Driven by numerical concerns

Behavior defined in mathematical terms

Avoid anomalies, establish portability across systems

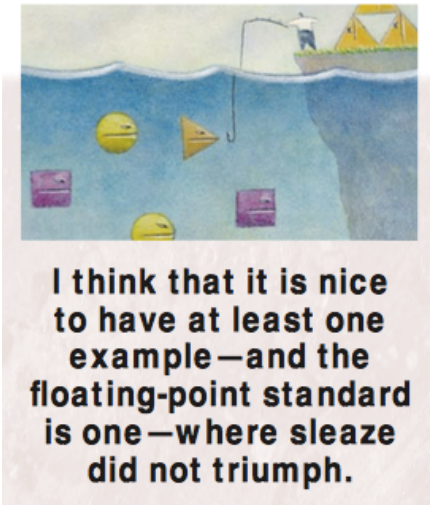
Clear standards for rounding, overflow, underflow

Support for transcendental functions

(square root, trig, exponentials, logarithms)

Hard to make fast in hardware

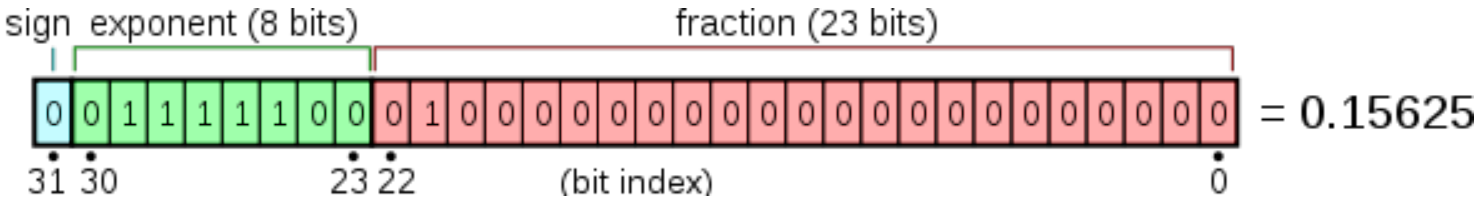
Numerical analysts predominated over hardware designers in defining standard



— Will Kahan
(chief architect of standard)

Floating point encoding

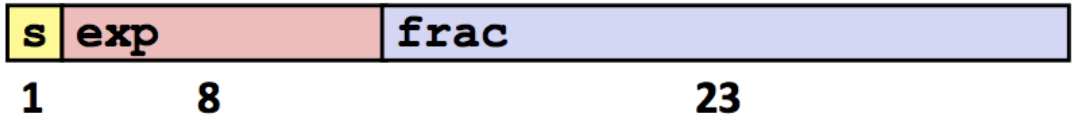
- **Numerical Form:** $(-1)^S\ M\ 2^E$
  - Sign bit `S` determines whether number is negative or positive
  - Significand `M` fractional value in range `[1.0,2.0)`
  - Exponent `E` scales value by power of two
- **Encoded into 32-bit float**
  - 1 sign bit, 8 exponent bits, 23 fractions
  - M and E have quirky encoding details
    - M has hidden leading, E is encoded relative to bias



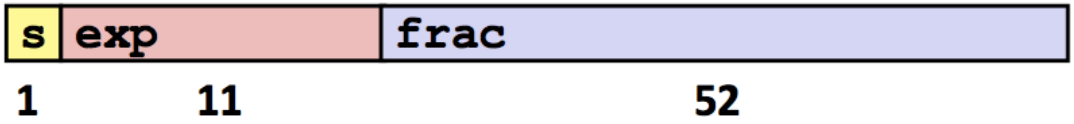
Float, double

- **Float, 32 bits**
  - **8-bit exponent**
    - Ranges -126 to +127, $2^{127} = 10^{37}$
  - **23-bit fraction**
    - Roughly 7 decimal digits of precision
- **Double, 64 bits**
  - **11-bit exponent**
    - Ranges -1022 to + 1023, $2^{1023} = 10^{308}$
  - **52-bit fraction**
    - Roughly 16 decimal digits of precision

Single precision: 32 bits



Double precision: 64 bits



Binary fractions

- **Value** **As binary fraction**

```
1/2       0.1
1/4       0.01
3/8       0.011

1/3       0.0101[01]...
1/5       0.00110011[0011]...
1/10      0.000110011[0011] ...
```

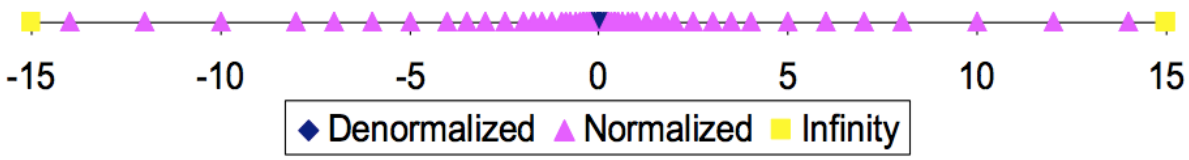
- **Which values are exactly representable as float?**
  - Magnitude of exponent must be between -126 127
  - Binary fraction terminates and fits into 23 (24) binary digits
- **All other values rounded to nearest representable**

Let's look at some code!

`/afs/ir/class/cs107/samples/lect10`

Distribution of values

- **Float number line is "holey"**
  - Value v is exactly representable. How far away is next neighbor on number line?
- **Binade**
  - All floats of same exponent form one binade
  - [1.0, 2.0) is a binade, [2.0, 4.0), [1024.0, 2048.0), and so on
  - 2^{23} values in each binade, spread evenly through range
- **Gap between representable values -> epsilon**
  - Epsilon is fixed for a given binade, doubles for each successive binade
- **Is epsilon always tiny?**
  - Relatively speaking yes, but not in absolute terms
  - As fraction of value = $1/2^{24}$



Floating point operations

- **Implementing floating point operations**
  - Separate significand and exponent bits, manipulate independently
  - Float ops historically more expensive than integer, but hardware has narrowed that gap
- **Floating point operations**
  - Basic idea: x op y = Round(x op y)
  - Compute exact result
  - Fix result
- **Sequence of operations: x op y op z = Round(Round(x op y) op z)**
  - Intermediate result is rounded
  - Approximation error in first result flows into rest of calculation
  - Single op is commutative, but sequence is not associative — wacky!
    - a + b equals b + a
    - BUT (a + b) + c may not equal a + (b + c)

Floating point multiplication

- **Multiplication** $(-1)^{S1}\ M1\ 2^{E1}\ \times\ (-1)^{S2}\ M2\ 2^{E2}$
  - **Exact result:** $(-1)^{S}\ M\ 2^{E}$
    - Sign s: s1 ^ s2
    - Significand M: M1 * M2
    - Exponent E: E1 + E2
  - **Fix result**
    - If M ≥ 2, shift M right, increment E
    - If E out of range, overflow (to infinity), underflow (to zero)
      Round M to fit frac precision
- **Implementation**
  - **biggest chore is in multiplying the significands**

Floating point addition

- $(-1)^{S1}$ M1 2^{E1} + $(-1)^{S2}$ M2 2^{E2}

  Exact Result: $(-1)^{S}$ M 2^{E}

  use larger of E1/E2 as exponent E

  shift right significand of smaller exponent to align the binary points

  signed add significands to compute S, M

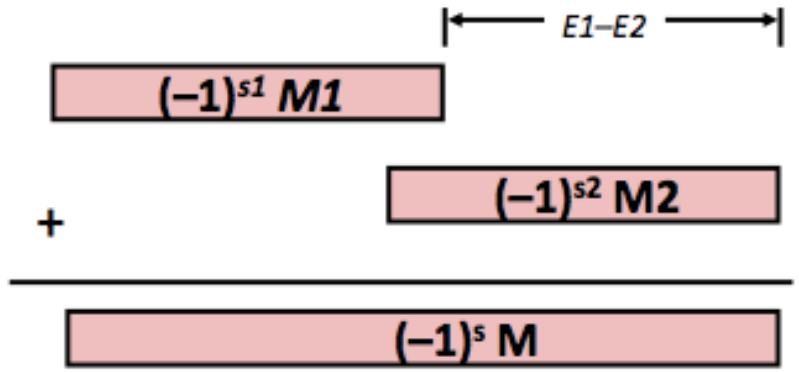
  Fix result

  If M ≥ 2, shift M right, increment E
  if M < 1, shift M left k positions, decrement E by k

  Overflow if E out of range
  Round M to fit frac precision

- **Rounding consequence**

  Small magnitude can be lost when added to large



Contrast float and int representations

32-bit int **-2,147,483,648 to 2147483647**

Every integer within range is representable

64-bit long **–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807**

32-bit float **1.7 x10^{-38} to ~3.4 x10^{38}**

4 billion values, but which 4 billion? Not even all integers in range can be represented

64-bit double **~9 x10^{-307} to ~2 x10^{308}**

Conversions/casting in C

Casting between int, float, and double changes bit representation

Double/float → int

Truncates fractional part

Not defined when out of range or NaN: Generally sets to INT_MIN

int → double

Exact conversion, as long as int has ≤ 53 bit word size

int → float

Will round according to rounding mode

Summary

- IEEE Floating Point has clear mathematical properties
- Represents numbers a form of binary scientific notation
- Can reason about operations independent of implementation
  - As if computed with perfect precision and then rounded
- Not the same as real arithmetic
  - Violates associativity/distributivity
  - Makes life difficult for compilers & serious numerical applications programmers