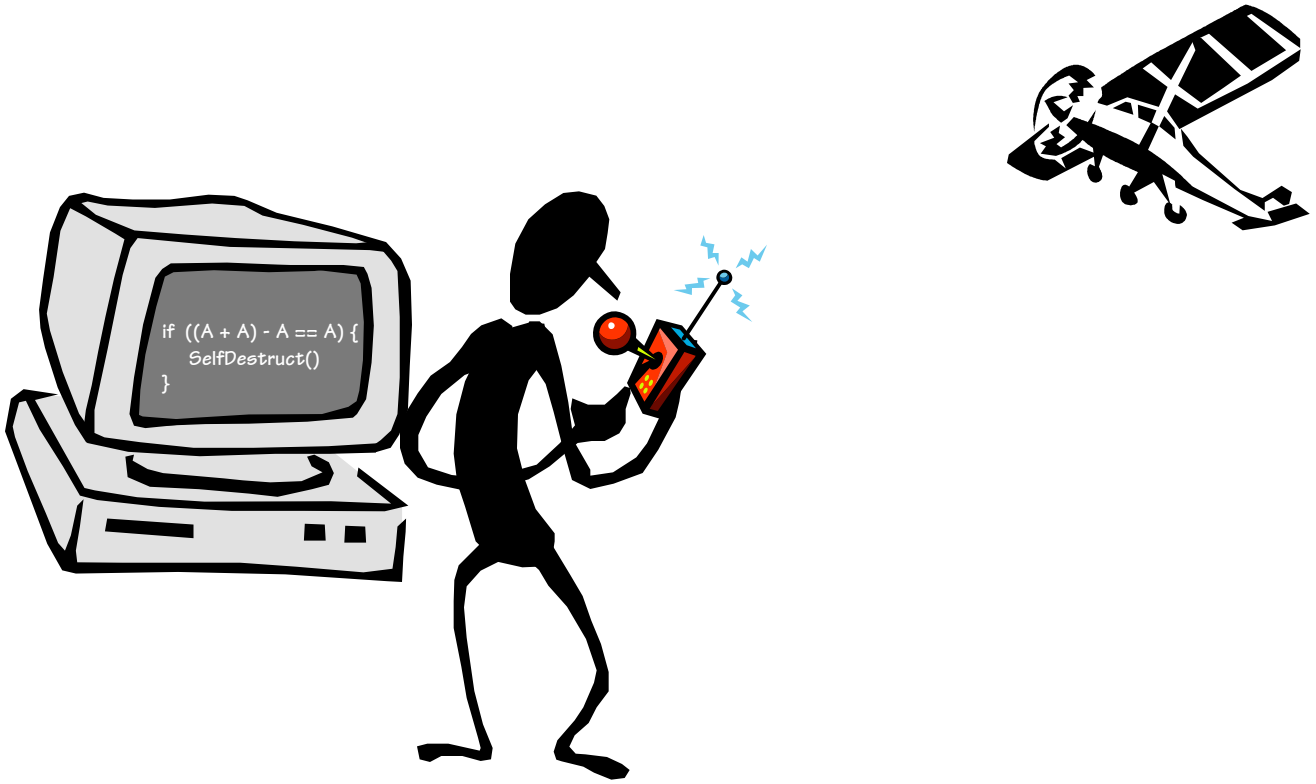
Floating-Point Arithmetic

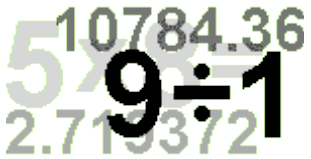


Reading: Study Chapter 3.

Why Floating Point?

Aren't Integers enough?

- Many applications require numbers with a VERY large range.
  (e.g. nanoseconds to centuries)



- Most scientific applications need real numbers (e.g. π)

But so far we have only used integers.

- We *COULD* use integers with a fixed "binary" point
- We *COULD* implement rational fractions using two integers (e.g. ½, 1023/102934)
- Floating point is a better answer for most applications.

Recall Scientific Notation

- Let's start our discussion of floating point by recalling scientific notation from high school
- Numbers represented in parts:

Significant Digits
Exponent

$$42 = 4.200 \times 10^{1}$$

$$1024 = 1.024 \times 10^{3}$$

$$-0.0625 = -6.250 \times 10^{-2}$$

- Arithmetic is done in pieces

$$\begin{array}{r} 1024 \\ -\ \ 42 \\ \hline 982 \end{array} \qquad \begin{array}{r} 1.024 \times 10^{3} \\ -\ 0.042 \times 10^{3} \\ \hline 0.982 \times 10^{3} \end{array}$$

Before adding, we must match the exponents, effectively "denormalizing" the smaller magnitude number

$$9.820 \times 10^{2}$$

We then "normalize" the final result so there is one digit to the left of the decimal point and adjust the exponent accordingly.

Multiplication in Scientific Notation

- Is straightforward:
  - Multiply together the significant parts
  - Add the exponents
  - Normalize if required
- Examples:

1024	1.024	x	10^{3}
x 0.0625	6.250	x	10^{-2}
64	6.400	x	10^{1}
42	4.200	x	10^{1}
x 0.0625	6.250	x	10^{-2}
2.625	26.250	x	10^{-1}

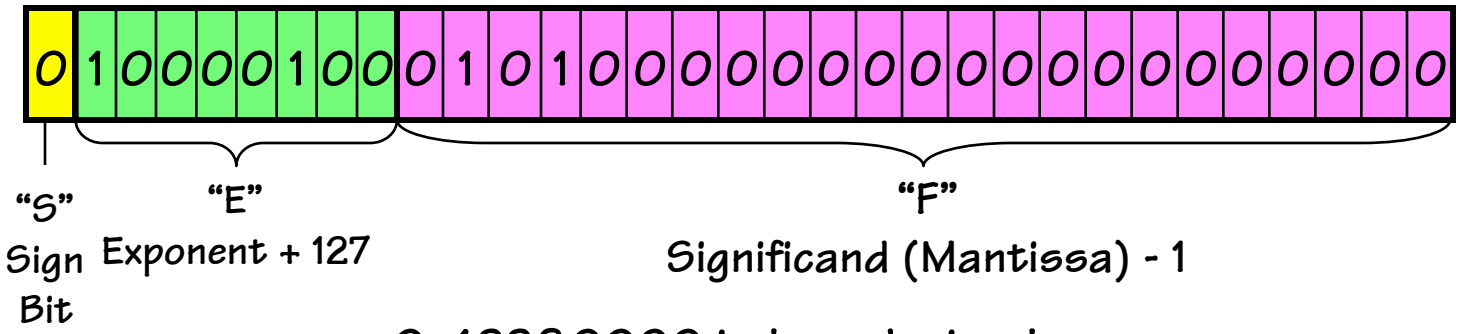
2.625 x 10^{0} (Normalized)

In multiplication, how far is the most you will ever normalize?

In addition?

FP == "Binary" Scientific Notation

- IEEE single precision floating-point format



0x42280000 in hexadecimal

- Exponent: Unsigned "Bias 127" 8-bit integer

  E = Exponent + 127

  Exponent = 10000100 (132) – 127 = 5

- Significand: Unsigned fixed binary point with "hidden-one"

  Significand = "1"+ 0.01010000000000000000000 = 1.3125

- Putting it all together

  $N = -1^S (1 + F) \times 2^{E-127} = -1^0 (1.3125) \times 2^5 = 42$

Example Numbers

- 1 (One) Sign = +, Exponent = 0, Significand = 1.0

  $-1^0 (1.0) \times 2^0 = 1$

  S = 0, E = 0 + 127, F = 1.0 – '1'

  0 01111111 00000000000000000000000 0x3f800000

- ½ (One-half) Sign = +, Exponent = -1, Significand = 1.0

  $-1^0 (1.0) \times 2^{-1} = ½$

  S = 0, E = -1 + 127, F = 1.0 – '1'

  0 01111110 00000000000000000000000 0x3f000000

- -2 (Minus Two) Sign = -, Exponent = 1, Significand = 1.0

  1 10000000 00000000000000000000000 0xc0000000

Zeros

- How do you represent 0?
- Zero Sign = ?, Exponent = ?, Significand = ?
  - Here's where the hidden "1" comes back to bite you
  - Hint: Zero is small. What's the smallest number you can generate?
  - E = Exponent + 127, Exponent = -127, Signficand = 1.0

$$1^0 \ (1.0) \times 2^{-127} = 5.87747 \times 10^{-39}$$

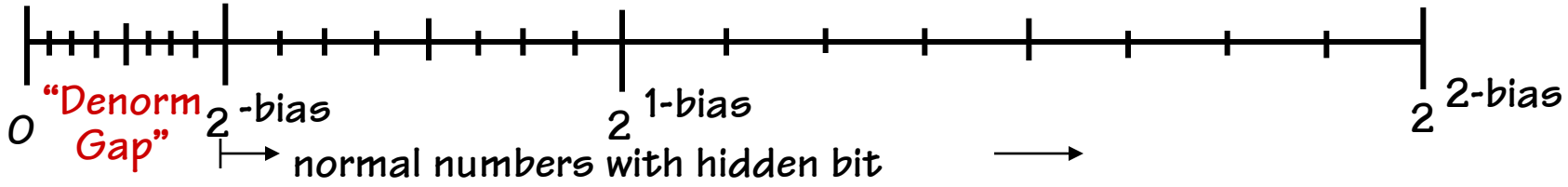
- <u>IEEE Convention</u>
  - When E = 0 (Exponent = -127), we'll interpret numbers differently…

0 00000000 00000000000000000000000 = 0 not 1.0×2^{-127}

1 00000000 00000000000000000000000 = -0 not -1.0×2^{-127}

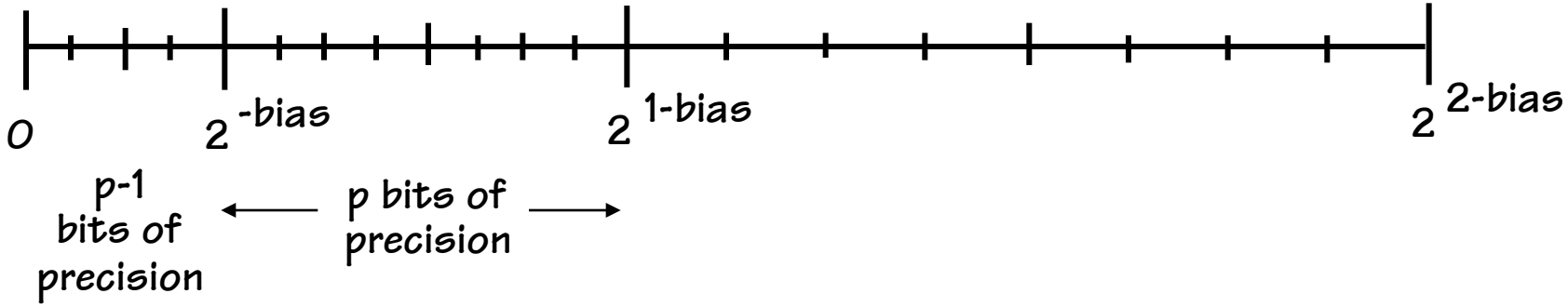
Yes, there are "2" zeros. Setting E=0 is also used to represent a few other small numbers besides 0. In all of these numbers there is no "hidden" one assumed in F, and they are called the "unnormalized numbers". WARNING: If you rely these values you are skating on thin ice!

Low-End of the IEEE Spectrum



The gap between 0 and the next representable normalized number is much larger than the gaps between nearby representable numbers.

IEEE standard uses denormalized numbers to fill in the gap, making the distances between numbers near 0 more alike.



Denormalized numbers have a hidden "0" and a fixed exponent of -126

$$X = (-1)^S \; 2^{-126} \; (0.F)$$

NOTE: Zero is represented using 0 for the exponent and 0 for the mantissa. Either, +0 or -0 can be represented, based on the sign bit.

Infinities

- IEEE floating point also reserves the largest possible exponent to represent "unrepresentable" large numbers
- Positive Infinity – S = 0, E = 255, F = 0
  0 11111111 00000000000000000000000 = +∞
  0x7f800000
- Negative Infinity -- S = 1, E = 255, F = 0
  1 11111111 00000000000000000000000 = -∞
  0xff800000
- Other numbers with E = 255 (F ≠ 0) are used to represent exceptions or Not-A-Number (NAN)
  $\sqrt{-1}$, -∞ x 42, 0/0, ∞/ ∞, log(-5)
- It does, however, attempt to handle a few special cases:
  1/0 = + ∞, -1/0 = - ∞, log(0) = - ∞

Floating Point Anomalies

It is CRUCIAL for computer scientists to know that Floating Point arithmetic is NOT the same arithmetic you learned in grade school!

1.0 is NOT EQUAL to 10*0.1 (Why?)

ex. 1.0 * 10.0 == 10.0 but, 0.1 * 10.0 != 1.0

0.1 decimal == 1/16 + 1/32 + 1/256 + 1/512 + 1/4096 + … ==

0.0 0011 0011 0011 0011 0011 …

Consider, in decimal 1/3 is a repeating fraction 0.333333…

If you quit at some fixed number of digits, then 3 * 1/3 != 1

F.P. ≠ Real Numbers

- Floating point is an approximation of "Real Numbers"; it even breaks basic LAWS of math

  Floating Point arithmetic IS NOT, in general, associative

  x + (y + z) is not necessarily equal to (x + y) + z

  Thus, the order of calculations matters, and *should* be considered

- Arithmetic is *approximate* whereas integer arithmetic is *exact*. Operations may not result in any visible change

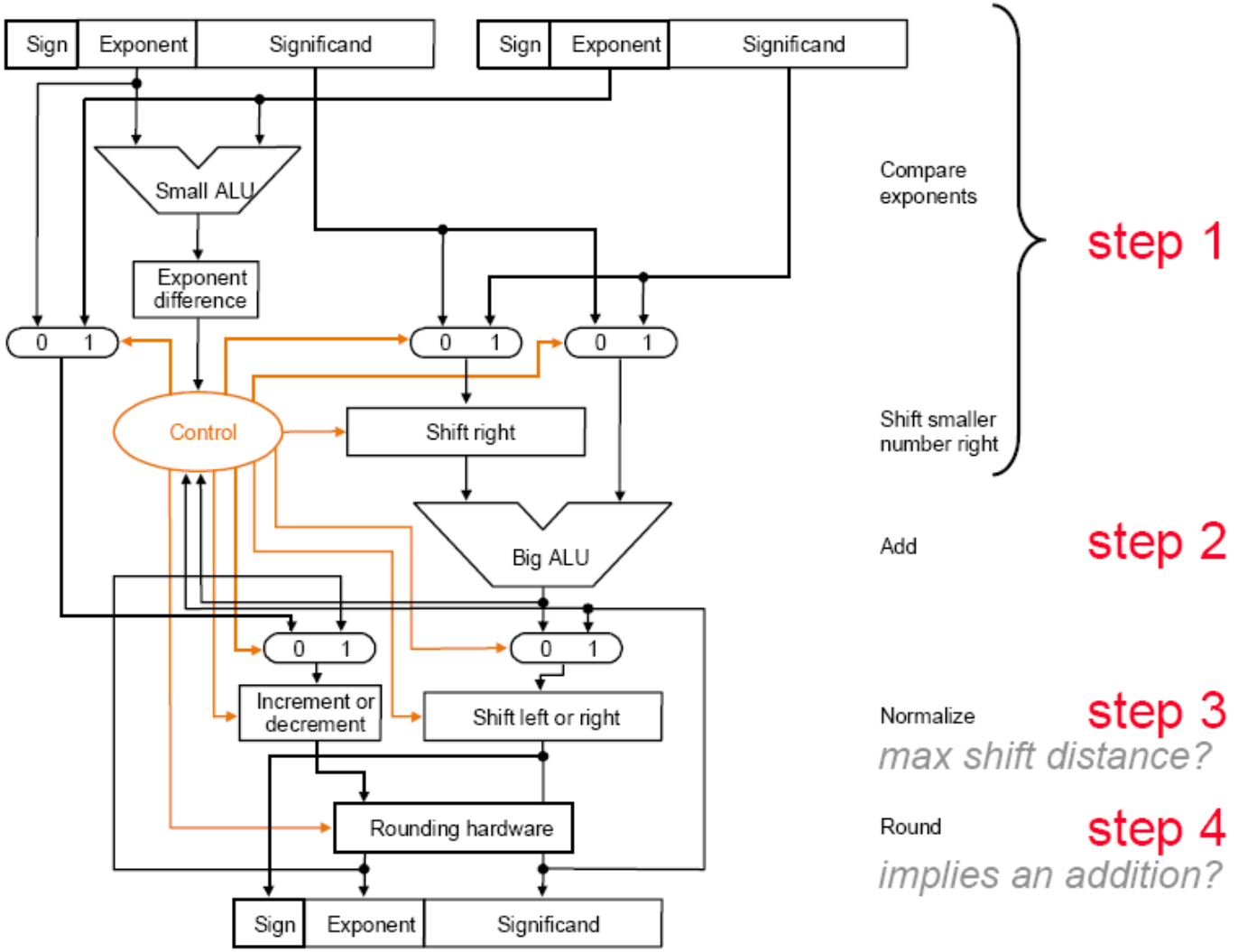
  (x + 1.0) MAY == x

- Floating-point is often *overused* by programmers, since the "finiteness" of the representation less apparent
- Programmers who assume that floating point numbers are real numbers, do so at their peril

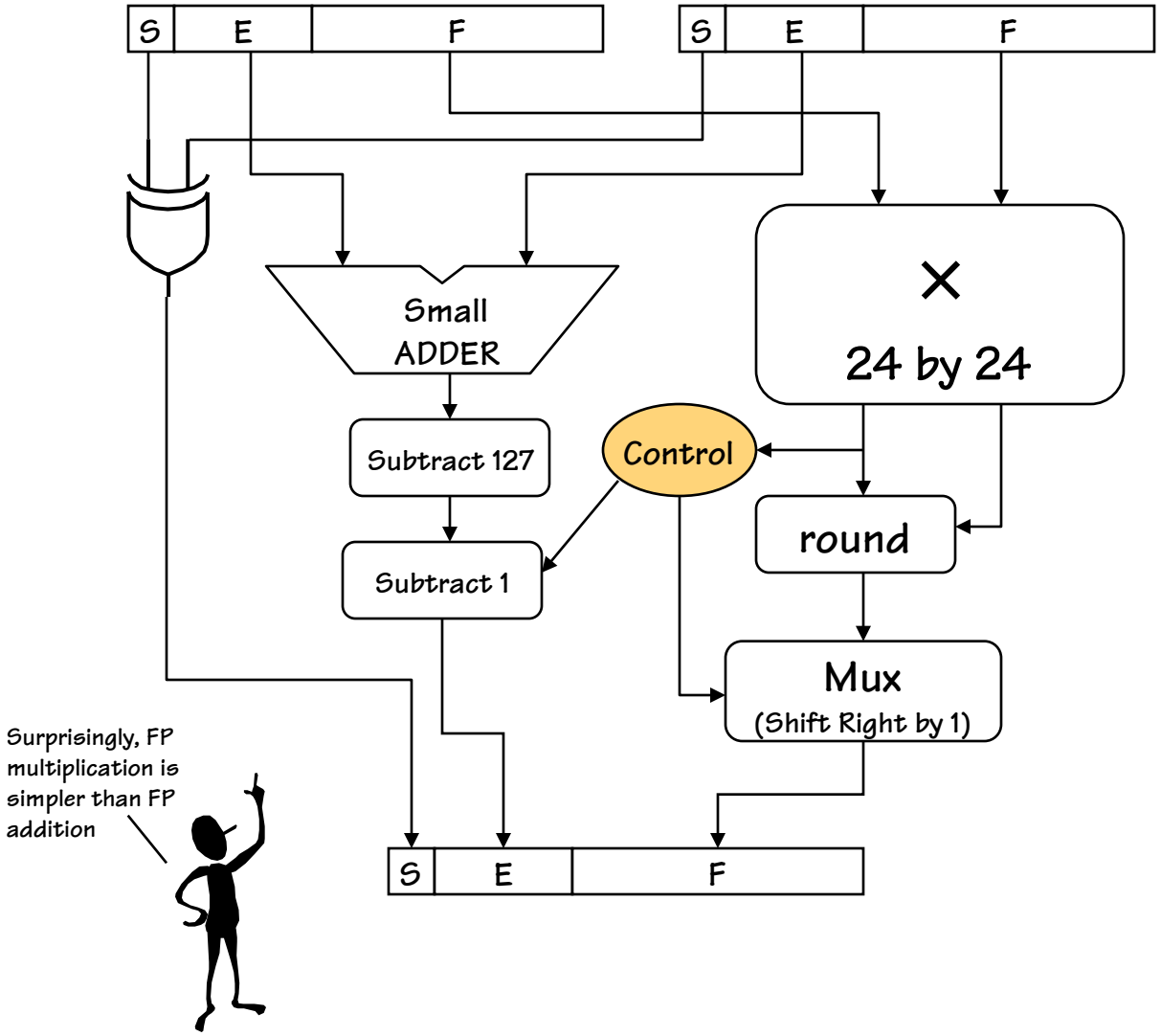
Floating Point Disasters

- Scud Missiles get through, 28 die
  - In 1991, during the 1st Gulf War, a Patriot missile defense system let a Scud get through, hit a barracks, and kill 28 people. The problem was due to a floating -point error when taking the difference of a converted & scaled integer. (Source: Robert Skeel, "Round-off error cripples Patriot Missile", SIAM News, July 1992.)
- $7B Rocket crashes (Ariane 5)
  - When the first ESA Ariane 5 was launched on June 4, 1996, it lasted only 39 seconds, then the rocket veered off course and self-destructed. An inertial system, produced an floating-point exception while trying to convert a 64-bit floating-point number to an integer. Ironically, the same code was used in the Ariane 4, but the larger values were never generated (http://www.around.com/ariane.html).
- Intel Ships and Denies Bugs
  - In 1994, Intel shipped its first Pentium processors with a floating-point divide bug. The bug was due to bad look-up tables used in to speed up quotient calculations. After months of denials, Intel adopted a no-questions replacement policy, costing them $300M. (http://www.intel.com/support/processors/pentium/fdiv/)

Floating-Point Addition H/W



Floating-Point Multiplication H/W



Step 1:

Multiply significands
Add exponents

$E_R = E_1 + E_2 - 127$

$Exponent_R + 127 = Exponent_1 + 127 + Exponent_2 + 127 - 127$

Step 2:

Normalize result
(Result of [1,2) *[1,2) = [1,4)
at most we shift right one bit, and fix exponent

MIPS Floating Point

Floating point "Co-processor"

32 Floating point registers

- separate from 32 general purpose registers, 32 bits wide each.
- uses an even-odd pair for double precision, R-type format

```
add.d fd, fs, ft      # fd = fs + ft in double precision
add.s fd, fs, ft      # fd = fs + ft in single precision

sub.d, sub.s, mul.d, mul.s, div.d, div.s, abs.d, abs.s

l.d  fd, address      # load a double from address
l.s, s.d, s.s
```

Conversion instructions, Compare instructions,
Branches (bc1t, bc1f)

Chapter Three Summary

Computer arithmetic is constrained by limited precision

Bit patterns have no inherent meaning, but standards do exist for representing numbers

- two's complement
- IEEE 754 floating point

Computer instructions determine "meaning" of the bit patterns

Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).

Numerical computing employs methods that differ from those of the math you learned in grade school.