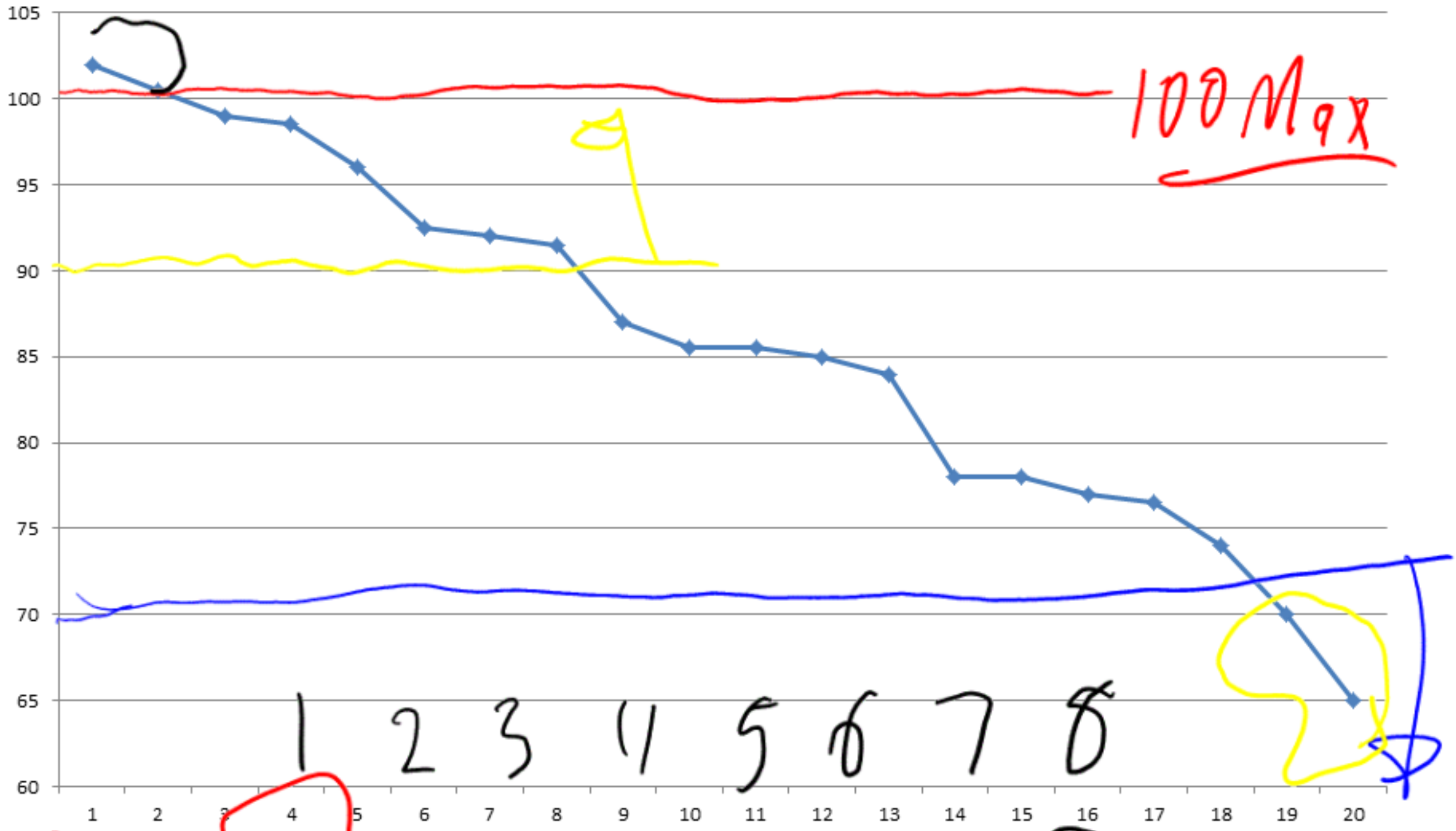


86%	Average	85%	71%	95%	95%	70%	75%	85%	60%	78%	81%	93%
86%	Median	85%	94%	100%	100%	75%	100%	100%	73%	88%	100%	100%
11%	STD	9%	40%	15%	8%	33%	44%	25%	30%	25%	37%	13%
102%	Max	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
65%	Min	75%	0%	50%	75%	0%	0%	0%	0%	25%	0%	67%

Floating Point Numbers

Lecture Objectives:

1) Define floating point number.
2) Define the terms fraction and exponent when dealing with floating point numbers.
3) Define overflow and underflow in relation to floating point numbers.
4) Convert a floating point number from binary to decimal format.
5) Convert a floating point number from decimal to floating point format.
6) Draw the algorithm for adding two floating point numbers.
7) Calculate the result of adding two floating point numbers together.

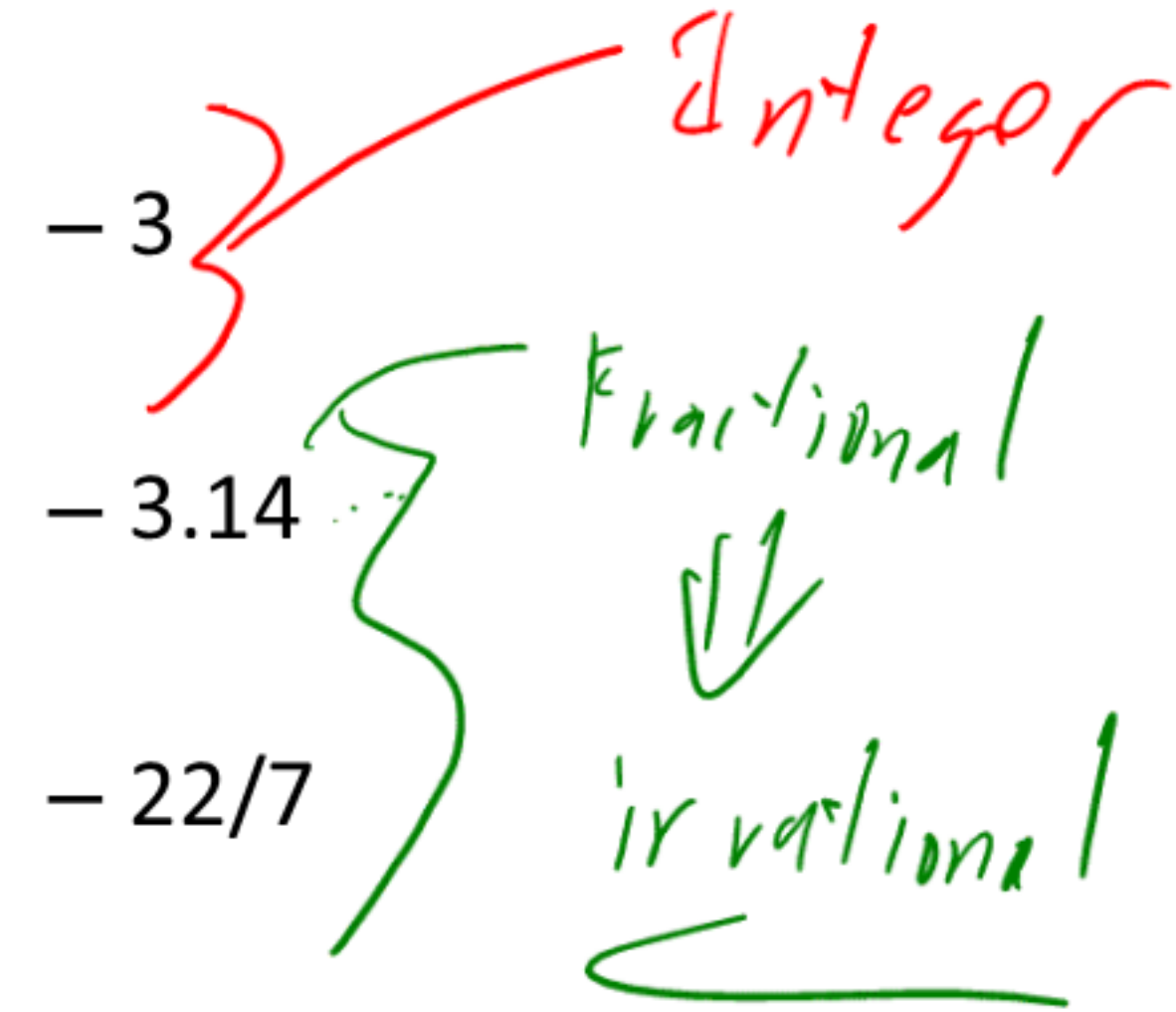
- What is the difference between the following numbers

Numbers

– 3

– 3.14

– 22/7



Definitions

- Scientific Notation
  - A notation which renders numbers with a single digit to the left of the decimal place
- Normalized
  - A number in floating point notation that has no leading zeros
    - -2.34×10^{56}
    - $+0.002 \times 10^{-4}$
    - $+987.02 \times 10^{9}$
- Floating Point
  - Computer arithmetic that represents numbers in which the binary point is not fixed

Definitions

- Fraction
  - The value, between 0 and 1, placed in the fraction field
- Exponent
  - The value that is placed in the exponent field

negative and big #'s

Left part ranging between 0 and 1

Floating Point

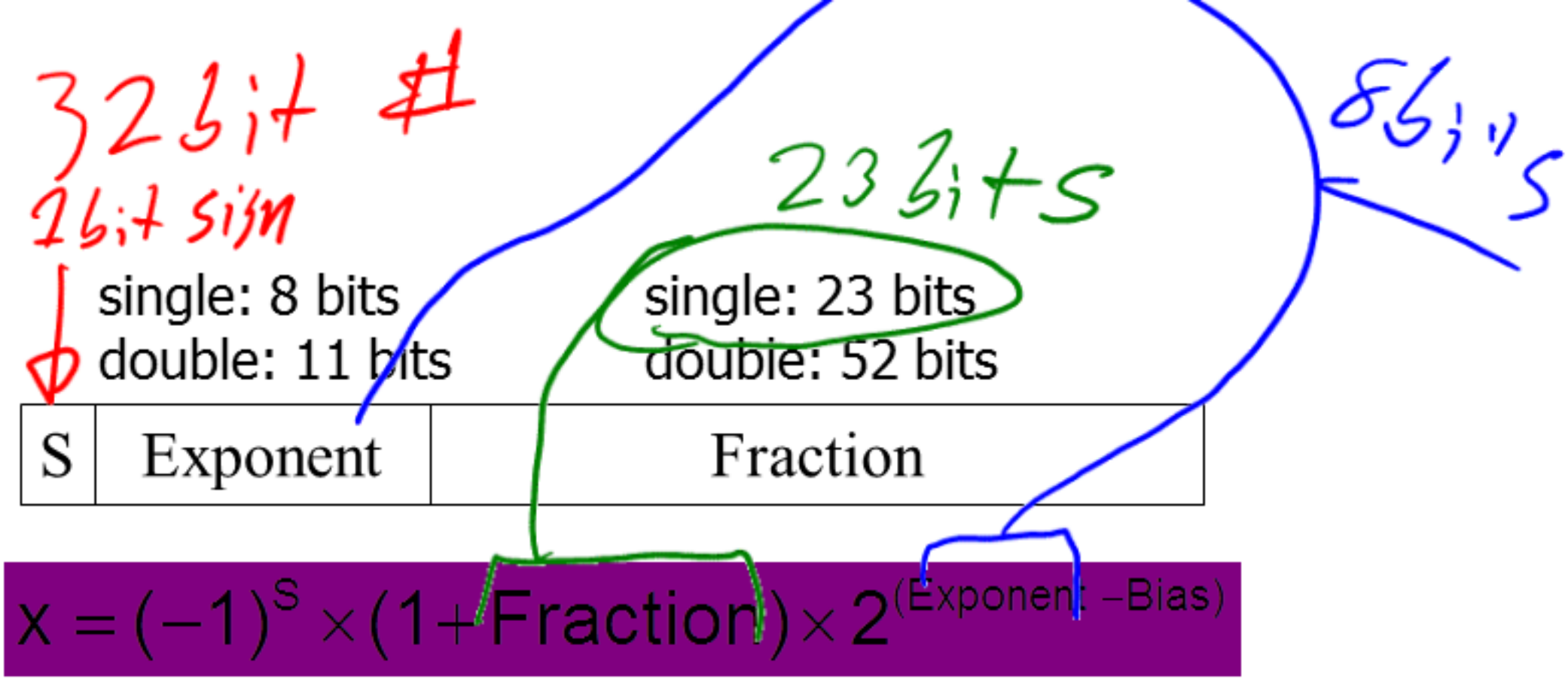
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

& Power. showing the scale.

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating Point Standard



- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: $1.0 \le |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

power

.5 ⇒ Sign bit ⇒ 0
positive 4+

Fraction ⇒ All zeros

Power/Exponent ⇒ -1

$X = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exp} - \text{bias})}$

$1 \times (1+0) \times 2^{(126-127)} = 0.5$

IEEE Floating Point Encodings

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Converting a number from Binary to Decimal Floating Point

Binary

S	Exponent								Fraction																			
7	6	5	4	3	2	1	0																					
0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$$(-1)^{S} \times (1 + Fraction) \times 2^{(Exponent - Bias)}$$

$(-1)^0 \times (1.5) \times 2^{(130 - 127)}$

$1 \times 1.5 \times 2^3$

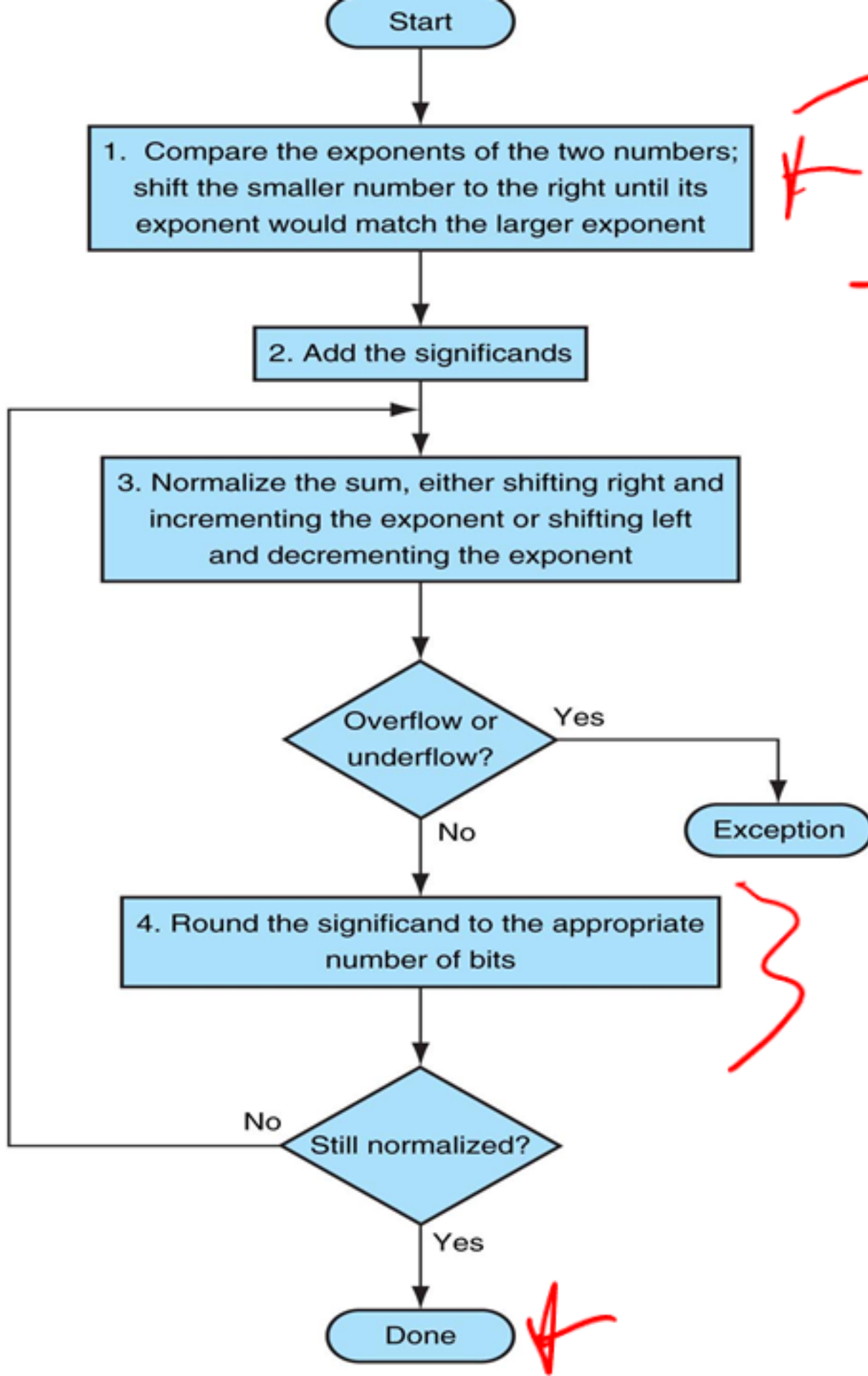
$1 \times 1.5 \times 8$

12

Ranges between $<= 0$ and < 1

MSOE

Scientific Notation Addition Algorithm



1) Making the exponents the same.

2) How accurate is the number.

Example

- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$

$.01610 \times 10^{1}$

9.999×10^{1}

$+ \quad .016 \times 10^{1}$

10.015×10^{1}

1.0015×10^{2} ← Increased #1 of Sig Figs.

1.002×10^{2} ← Rounded

Binary Example

- $0.5 + -0.4375$ ∈ Not binary

$-1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ is binary

$1.000_2 \times 2^{-1}$

$0.1110_2 \times 2^{-1}$

$1.110_2 \times 2^{-1}$

Binary Example

- $0.5 + -0.4375$

$$-1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$$

4 Sig Figs

0 1 1 2

-1.0000×2^{-1}

-0.1110×2^{-1}

0.0010×2^{-1}

1.0×2^{-4}

1. Align binary points.
2. Add Significands.
3. Normalize.

2 Sig Figs

$1.0_2 \times 2^{-4} \Rightarrow -.0625$

0.4375

.0111

$2^{-1} \Rightarrow .5$

$2^{-2} \Rightarrow .25$

$2^{-3} \Rightarrow .125$

$2^{-4} \Rightarrow .0625$

$2^{-5} \Rightarrow$

.4375
− .25
.1875
.1250
.0625
.0625
0

$.0111_b$

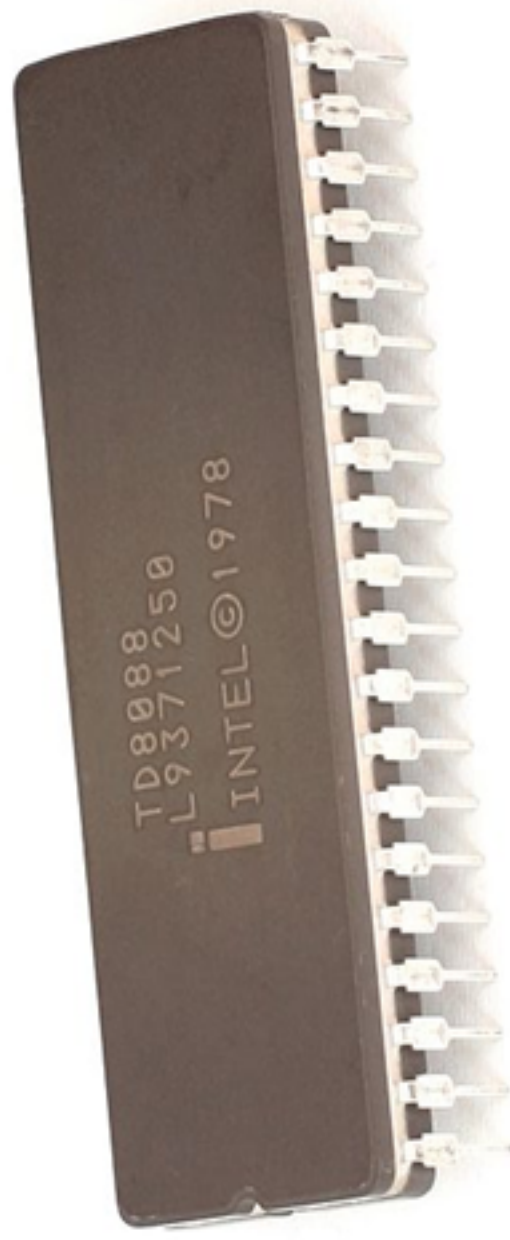
Normalize:

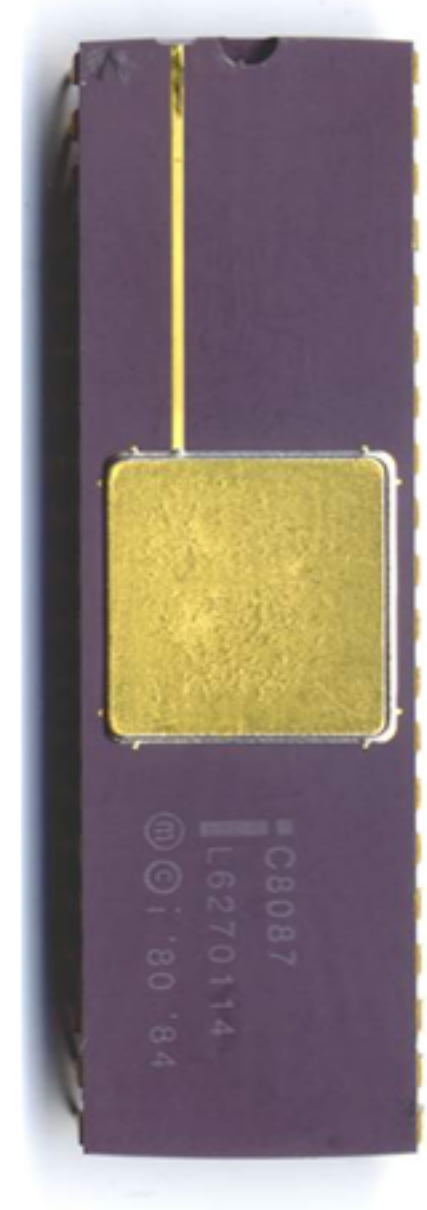
1.11×2^{-2}

Floating-Point Addition

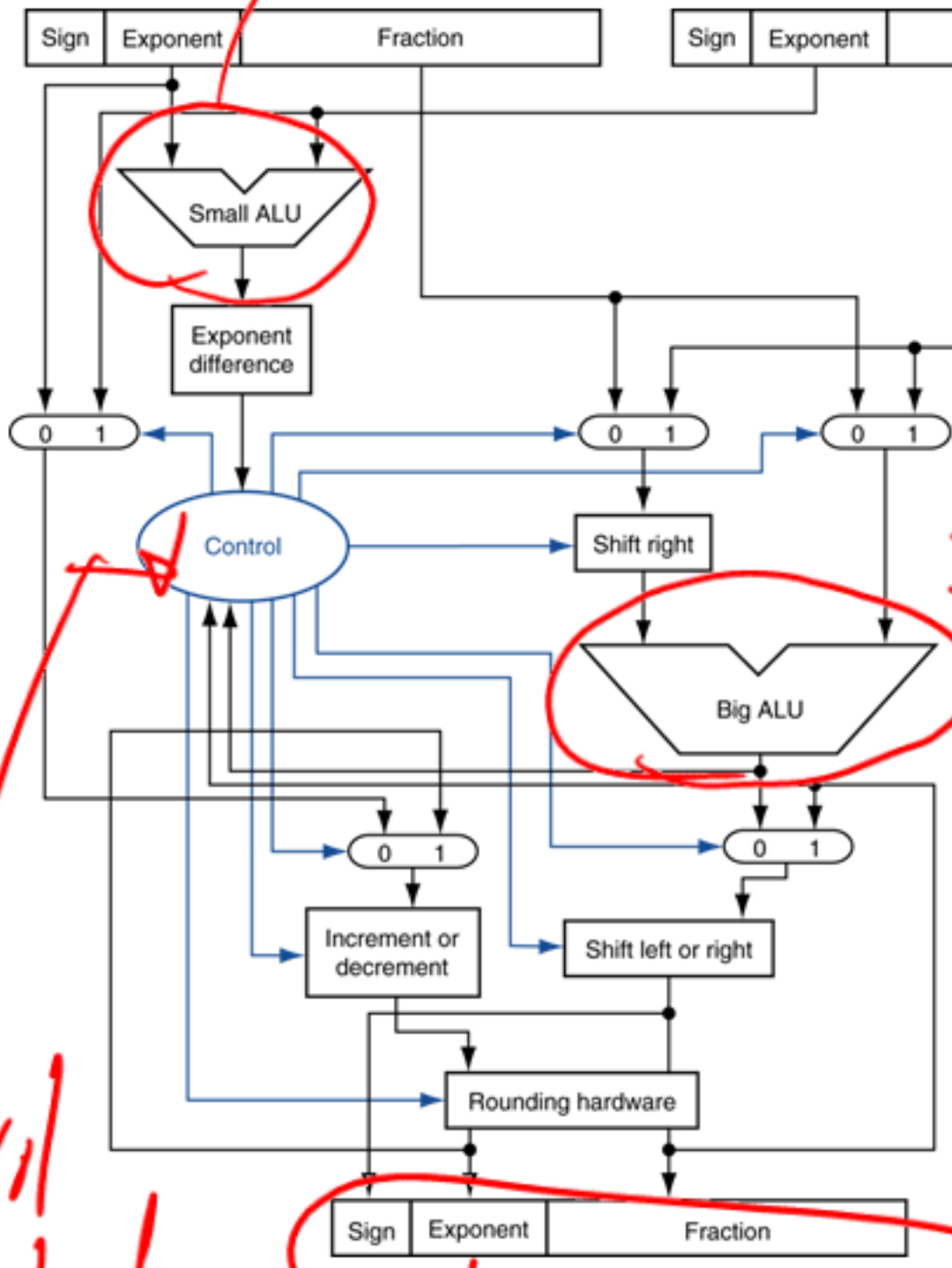
- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

Floating Point Hardware





FP Adder Hardware



Compare exponents

Shift smaller number right

Step 1

Add

Step 2

Normalize

Step 3

Round

Step 4

Floating Point Numbers Part 2

Lecture Objectives:

1) Draw the algorithm for multiplying two floating point numbers together.
2) Calculate the result of the multiplication of two floating point numbers.
3) Explain the ramifications of the pentium division bug.

Quiz Minute

- Take out a sheet of paper and complete the following:
  - Convert the following number to decimal from IEEE floating point

S	Exponent								Fraction																			
0	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Add the following two binary numbers from floating point

  1.001×2^{-1}

  1.100×2^{-2}

Converting a number from Binary to Decimal Floating Point

S	Exponent								Fraction																			
0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

$$(-1)^{S} \times (1 + Fraction) \times 2^{(Exponent - Bias)}$$

Floating Point Multiplication

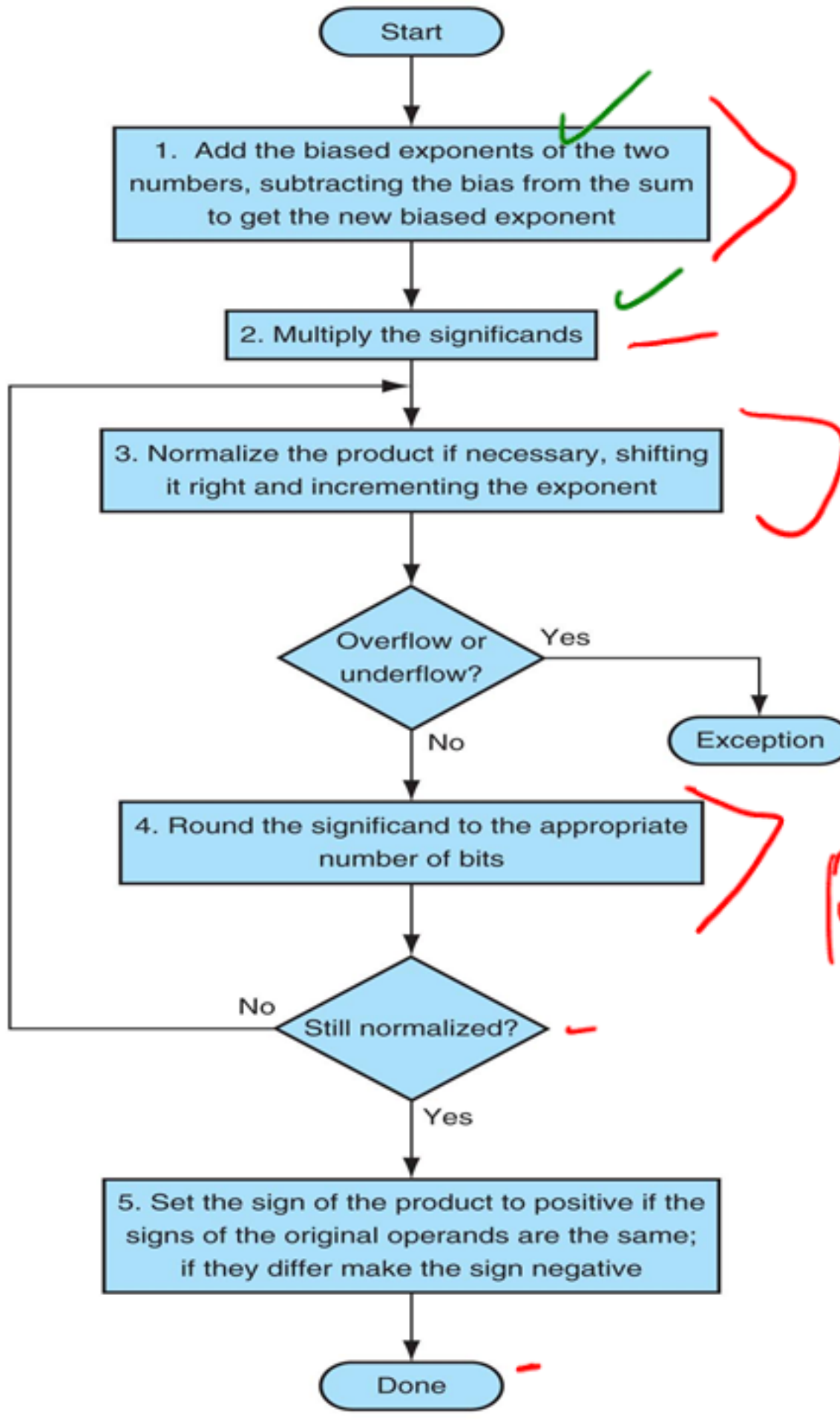


Figure out how close to the answer we

Floating Point Multiplication Example

$$1.100_{two} \times 2^{-1}$$

$$\times 1.010_{two} \times 2^{2}$$

What are these =1's?

1.100
x1.010 X2^1
0000
11000
00000
1100000
1.111000 X2^1

11.11_b x 2^0

3.75

$1,100 \times 2^{-0} = 7.5$

If

.5 .25 =

$1,010 \times 2^{2}$

101.0×2^{0}

4 1 = 5

$7\frac{2}{5}$

5

3.75

FP Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

"slow"

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPs ISA supports 32 × 64-bit FP reg's
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, `div.s`
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*xx*`.s`, `c.`*xx*`.d` (*xx* is `eq`, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

Lets work an example

- We want to write a program which will convert from degrees into Radians

$$radians = degrees \times \frac{\pi}{180}$$

3.14.....

formula

floating point program

⇒ MIPS coprocessor

⇒ double precision.

Program Flowchart

Who Cares About FP Accuracy?

- Important for scientific code
  - But for everyday consumer use?
    - "My bank balance is out by 0.0002¢!" ☹
- The Intel Pentium FDIV bug
  - The market expects accuracy
  - See Colwell, *The Pentium Chronicles*

Pentium Floating point bug