University of San Carlos | Department of
COMPUTER ENGINEERING

CpE 3202
Computer Organization & Architecture

Computer Arithmetic

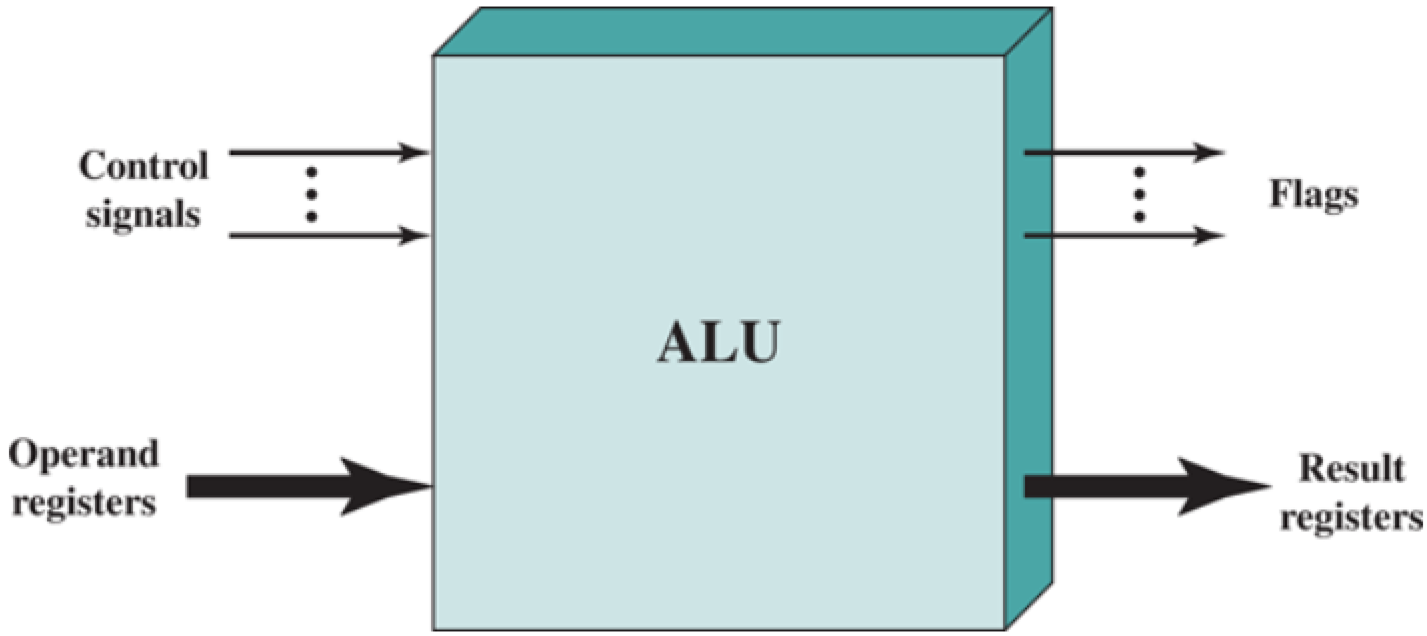
Computer Arithmetic

- Most complex aspect of the Arithmetic and Logic Unit (ALU)
- It is performed on two very different numbers: *integer* and *floating point*

Arithmetic and Logic Unit (ALU)

- Does the calculations
- Everything else in the computer is there to service this unit
- Handles integers
- May handle floating point (real) numbers
- May include a separate floating-point unit (FPU), as a math co-processor

ALU Inputs and Outputs



Integer Representation

- Only have 0 & 1 to represent everything
- Positive numbers stored in binary
  - e.g. 41 = 0010 1001
- No minus sign
- No radix point *(e.g. decimal point for fractional component)*
- Sign-Magnitude

32 8 1

54.5

1010.1110

Sign-Magnitude

- Used to represent positive and negative integers
- Left most bit is **sign bit**
  - 0 means positive
  - 1 means negative
  - Examples: +18 = 0001 0010 (16, 2)
    -18 = 1001 0010

    0000 = +0
    1000 = -0
- **Problems**
  - Need to consider both sign and magnitude in arithmetic
  - Two representations of zero (+0 and -0)

Twos Complement Notation

- +3 = 0000 0011
- +2 = 0000 0010
- +1 = 0000 0001
- +0 = 0000 0000
- -1 = 1111 1111
- -2 = 1111 1110
- -3 = 1111 1101

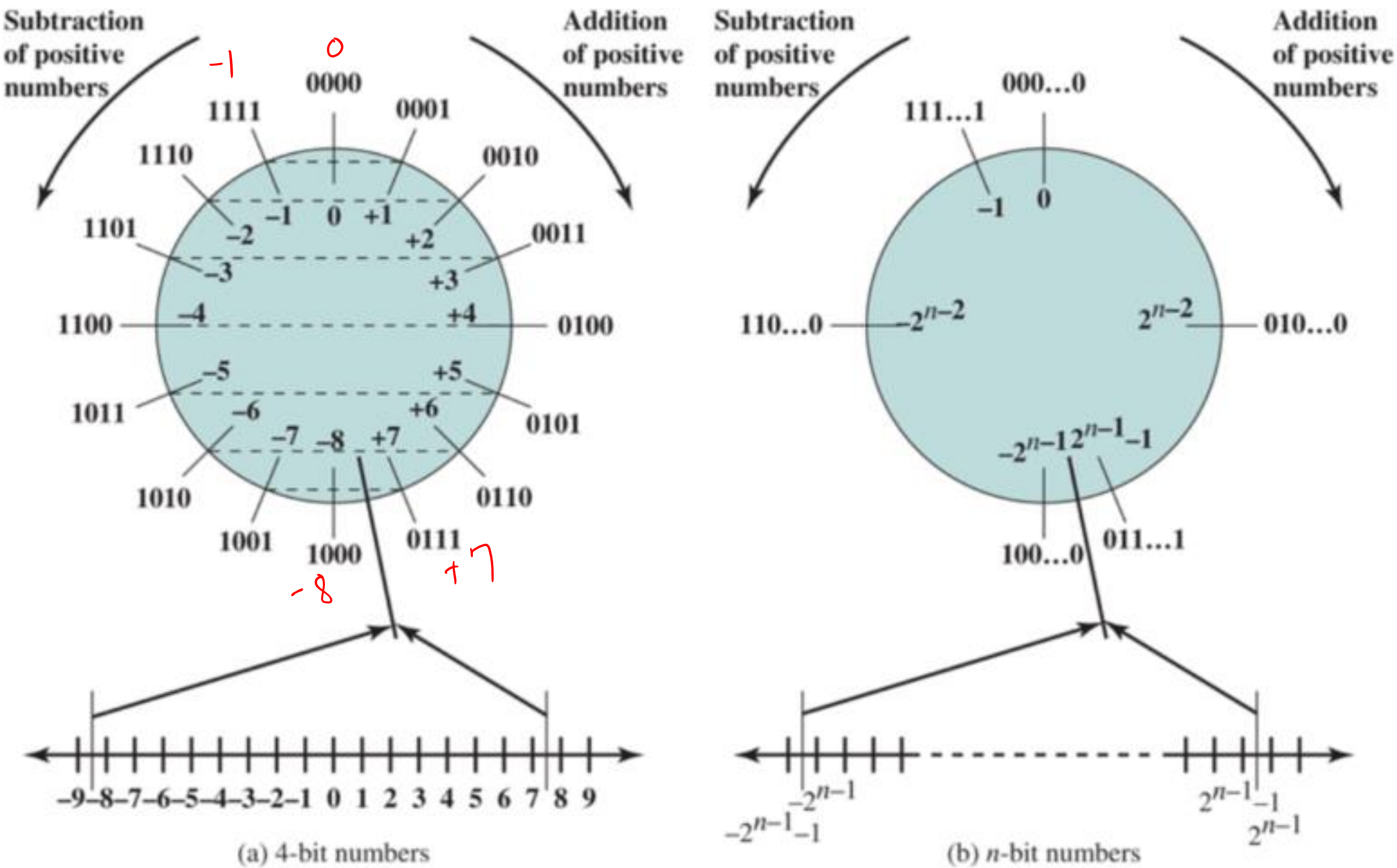
$$
\begin{array}{r}
0000\ 0001 \\
1111\ 1110 \\
+\ \ \ \ \ \ \ \ \ 1 \\
\hline
1111\ 1111
\end{array}
\qquad
\begin{array}{r}
0000\ 0010 \\
1111\ 1101 \\
+\ \ \ \ \ \ \ \ \ 1 \\
\hline
1111\ 1110
\end{array}
$$

Benefits

- Only one representation of zero
- Arithmetic works easily *(see later)*
- Negating is fairly easy
  - 3 = 0000 0011 ✓
  - Boolean complement gives 1111 1100
  - Add 1 to LSB 1111 1101 = -3

```
  0000 0010
+         1
-----------
  0000 0011 = 3
```

Geometric Depiction of Twos Complement Integers



Negation Special Case 1

- 0 = 0000 0000
  - Bitwise not 1111 1111
  - Add 1 to LSB +1
  - Result 1 0000 0000 -0

OF = 1

- Overflow is ignored, so:
  - - 0 = 0 ✓

Negation Special Case 2

- -128 = 1000 0000 ✓
  - Bitwise not 0111 1111
  - Add 1 to LSB +1
  - Result 1000 0000
- So:
  - -(-128) = -128 ✗

- Monitor MSB (sign bit)
  - It should change during negation

Range of Numbers

- 8 bit 2s complement
  - +127 = 0111 1111 = 2^7-1
  - -128 = 1000 0001 = -2^7
- 16 bit 2s complement
  - +32,767 = 0111 1111 1111 1111 = 2^{15} - 1
  - -32,768 = 1000 0000 0000 0001 = -2^{15}

n=8

max $2^{n-1} - 1$

min -2^{n-1}

n

4 bits n=4 -8 to +7

0000 0
0001 1
⋮
0111 7 = $2^3 - 1$ = 7
1000 -8 = -2^3 = -8
1001
⋮
1111 -1

1000 = -8

$$\begin{array}{r} 1\,1\,1 \\ 0111 \\ +\quad 1 \\ \hline 1000 = 8 \\ 8\;4\;2\;1 \end{array}$$

Conversion Between Lengths

- Positive numbers: pad with **leading zeros**
  - +18 = 0001 0010 = 8
  - +18 = 0000 0000 0001 0010 = 32
- Negative numbers: pad with **leading ones**
  - -18 = 1001 0010 -
  - -18 = 1111 1111 1001 0010
- Pad using the MSB (sign bit)

Addition

7 0111 → 1000 + 1 = 1001 −7

1110: 0001 + 1 = 0010 = 2

4 0100 = 4 → 1011 + 1 = 1100 = −4

0001 = 1 → 1110 + 1 = 1111 = −1

1011 = −? → 0100 + 1 = 0101 = 5

- Proceeds as if the two numbers are unsigned integers
  - Normal binary addition
- **Overflow -** result exceeds the bit-width size *(extra bit is ignored)*
  - *If two numbers are added, and **they are both positive or both negative**, the overflow occurs **if and only if** the result has the opposite sign.*

5 bits
00101 = 5
00100 = 4
01001 = 9
not overflow

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5) = −2	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1) = −5
0101 = 5 + +0100 = 4 + 1001 = Overflow (e) (+5) + (+4) = 9	1001 = −7 − +1010 = −6 − 10011 = Overflow (f) (−7) + (−6)

Subtraction

M = 0101 = 5 ✓
S = 0010 = 2
-S = 1101 + 1 = 1110 = -2 ✓

M = -5   0101 + 1010 + 1 = 1011 = -5
S = 2   0010

-S = 1101 + 1 = 1110 = -2

- Take **twos complement of subtrahend (S)** and **add to minuend (M)**
  - M - S = M + (-S)
- **When overflow occurs**, the ALU send a signal so that no attempt is made to use the result.

OF=1

1001 = -? -7
+ 0110 + 1 = 0111 = 7

0010 = 2 +1001 = −7 1011 = −5 (a) M = 2 = 0010 S = 7 = 0111 −S = 1001	0101 = 5 +1110 = −2 ~~1~~0011 = 3 (b) M = 5 = 0101 S = 2 = 0010 −S = 1110 = 3
1011 = −5 ✓ +1110 = −2 ✓ ~~1~~1001 = −7 ✓ (c) M = −5 = 1011 S = 2 = 0010 −S = 1110 = -7	0101 = 5 +0010 = 2 0111 = 7 (d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 + +0111 = 7 + - 1110 = Overflow (e) M = 7 = 0111 S = −7 = 1001 −S = 0111	1010 = −6 - +1100 = −4 - + ~~1~~0110 = Overflow (f) M = −6 = 1010 S = 4 = 0100 −S = 1100

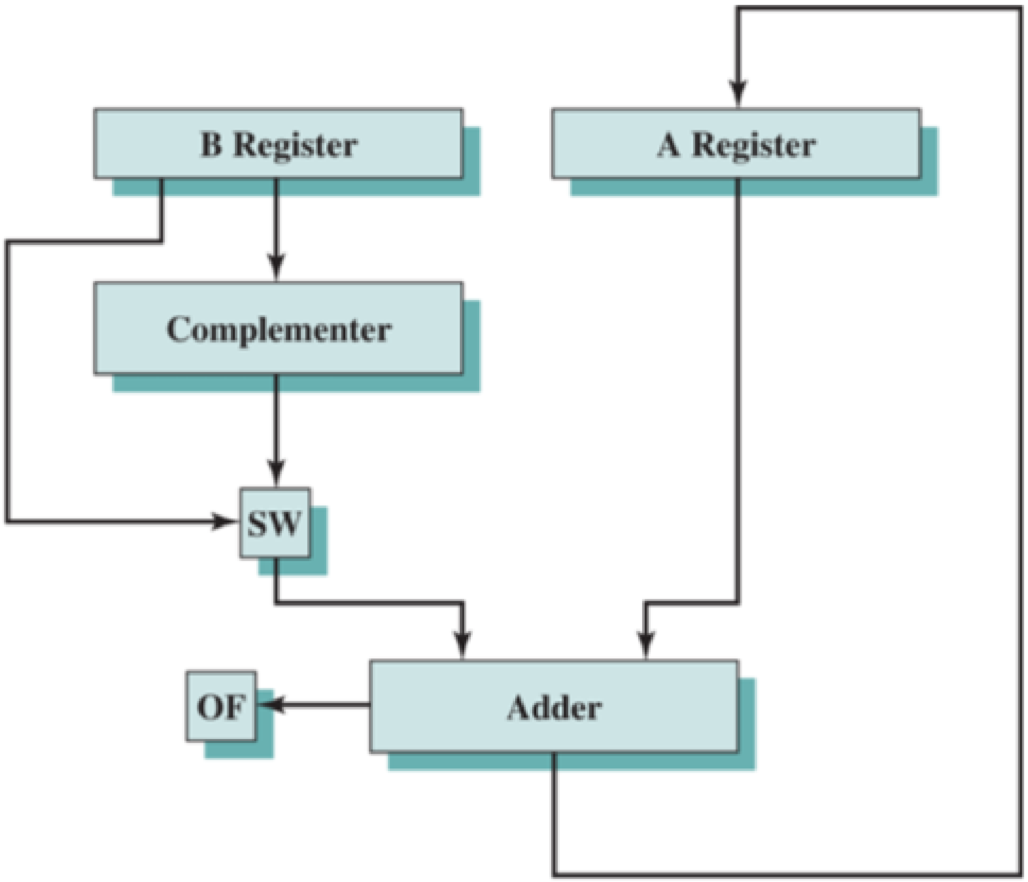
5 - (-8) = 13
S = -8 = 1000 + 0111 + 1 = 1000
-S =

0101 = 5
+ 1001 = -8
1111 -1

1111 + 0000 + 1 = 0001

Hardware for Addition and Subtraction

- For addition and subtraction, we only need addition and complement circuitry



OF = Overflow bit
SW = Switch (select addition or subtraction)

Multiplication

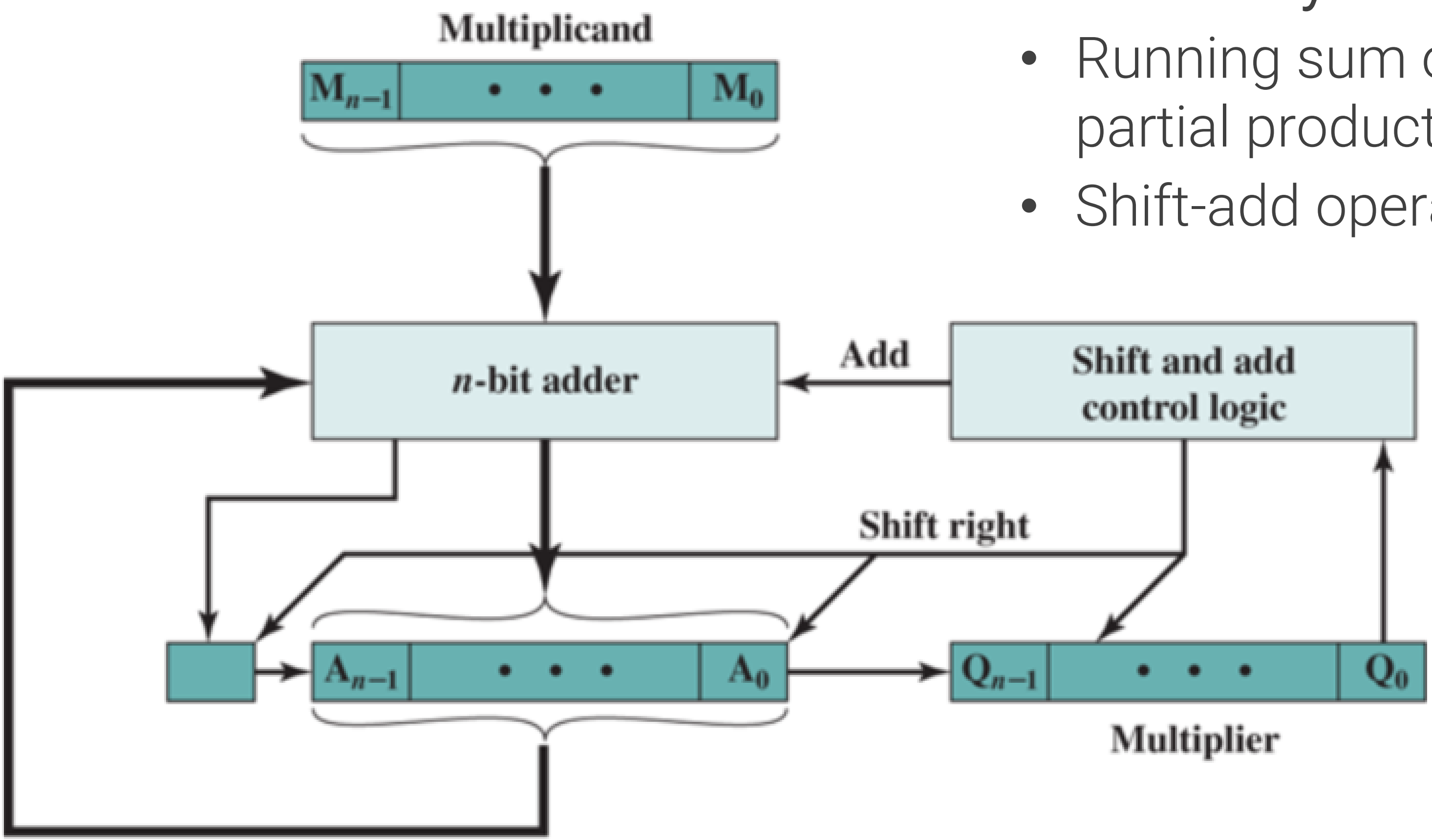
- Complex
- Work out a **partial product** for each digit
- Take care with placing values *(column)*
- Add partial products

Multiplication Example

1011	Multiplicand	(11_{10})
x 1101	Multiplier	(13_{10})
1011	<u>Partial products</u>	
0000	*Note:* IF multiplier bit is 1, copy multiplicand	
+ 1011	*(multiplicand is the partial product);*	
1011	otherwise, zero	
10001111	Product	(143_{10})

- **Note:** Need a **double-length width result** *(2n-bit result for n-bit binary integers)*

Unsigned Binary Multiplication



(a) Block diagram

- For efficiency:
  - Running sum on the partial products
  - Shift-add operation

Execution of Example

```
      1011 (11₁₀)  ✓ M
x     1101 (13₁₀)  ✓ Q
-------------------
      1011
     0000
    1011
   1011
-------------------
  10001111 (143₁₀)
```

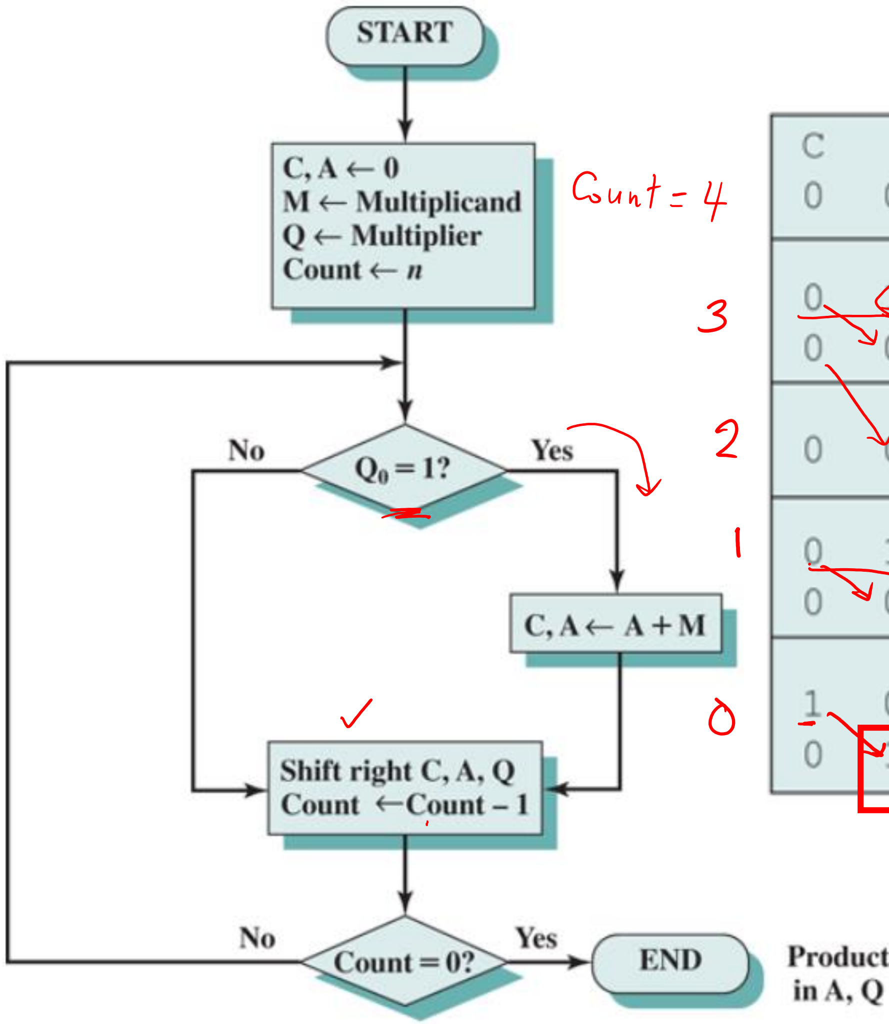
C	A	Q	M			
0	0000	1101	1011	**Initial values**		
0	1011	1101	1011	**Add**	}	**First cycle**
0	0101	1110	1011	**Shift**		
0	0010	1111	1011	**Shift**	}	**Second cycle**
0	1101	1111	1011	**Add**	}	**Third cycle**
0	0110	1111	1011	**Shift**		
1	0001	1111	1011	**Add**	}	**Fourth cycle**
0	$\boxed{1000}$	$\boxed{1111}$	1011	**Shift**		

Product in {A, Q}

Note:

- M = Multiplicand
- Q = Multiplier
- A = Accumulator
- C = Carry bit

Flowchart for Unsigned Binary Multiplication



Count = 4

C	A	Q	M		
0	0000	1101	1011	Initial values	
0	1011	1101	1011	Add	First cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second cycle
0	1101	1111	1011	Add	Third cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth cycle
0	1000	1111	1011	Shift	

3

2

1

0

$$\begin{array}{r} 0010 \\ +\ 1011 \\ \hline 1101 \end{array} \qquad \begin{array}{r} 0110 \\ +\ 1011 \\ \hline 10001 \end{array}$$

Multiplying Negative Numbers

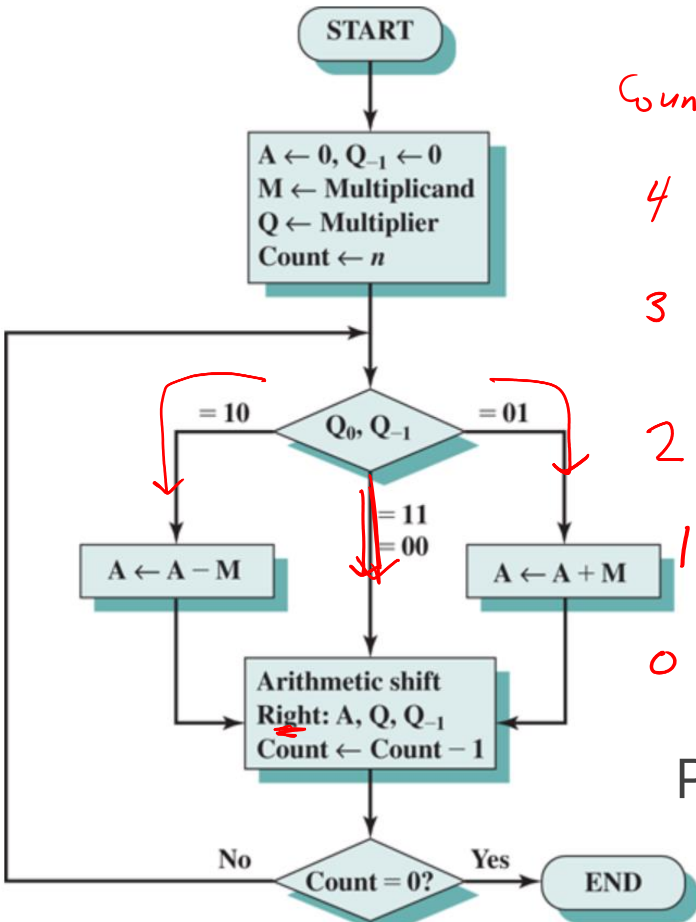
- This does not work!
- Solution 1
  - Convert to positive, if required
  - Multiply as above
  - If signs were different, negate answer
- Solution 2
  - Booth's algorithm

Booth's Algorithm

A 0000
M 0111
−M 1000
+ 1
1001

A + (−M)
0000
+ 1001
1001

1 1
1110
+ 0111
~~1~~0101



Perform: $3_{10} \times 7_{10} = 21_{10}$

Count	A	Q	Q_{-1}	M		
4	0000	0011	0	0111	Initial values	
3	1001	0011	0	0111	A←A − M	First cycle
	1100	1001	1	0111	Shift	
2	1110	0100	1	0111	Shift	Second cycle
1	0101	0100	1	0111	A←A + M	Third cycle
	0010	1010	0	0111	Shift	
0	$\boxed{0001}$	$\boxed{0101}$	0	0111	Shift	Fourth cycle

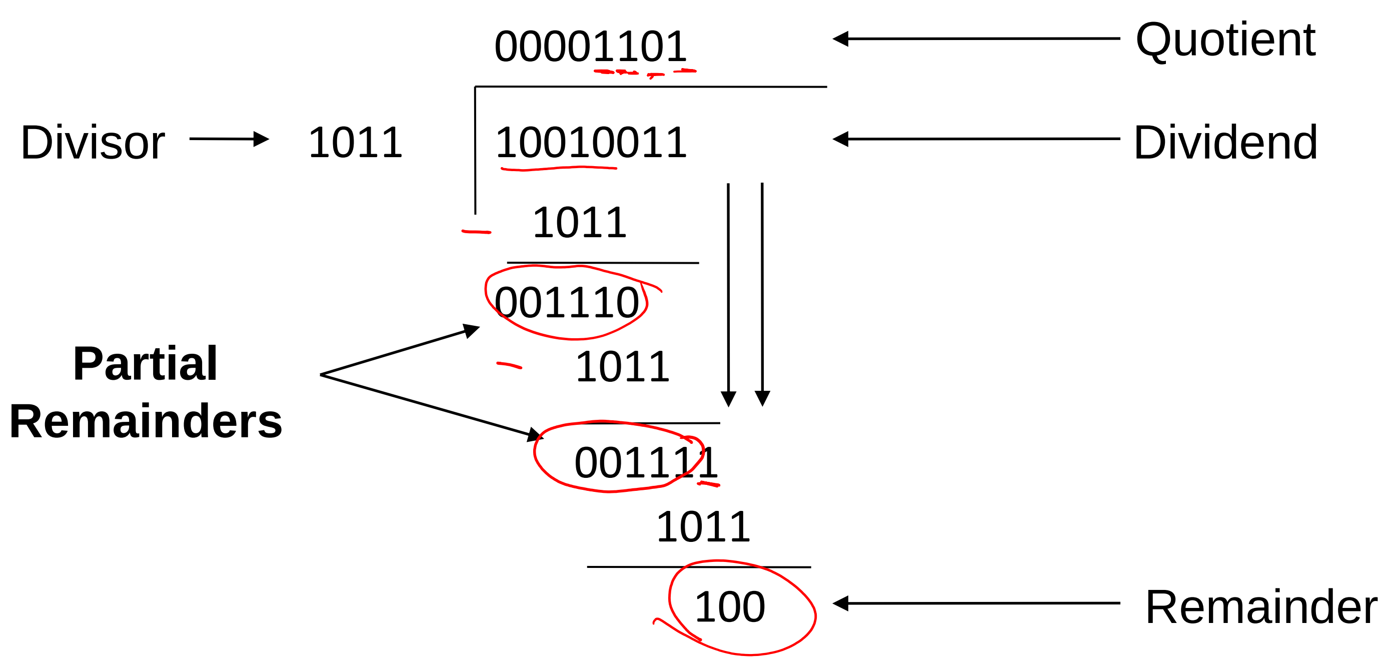
16 4 1

Product in {A, Q}

Division

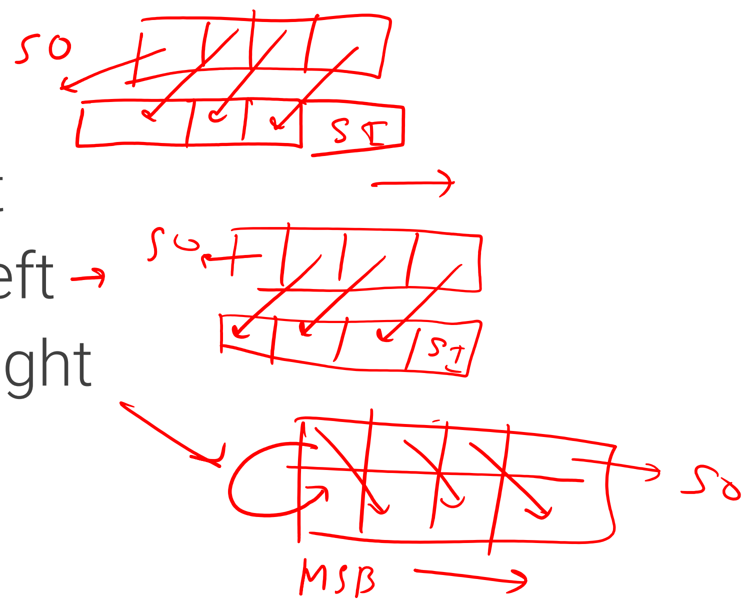
- More complex than multiplication
- Negative numbers are really bad!
- Based on long division

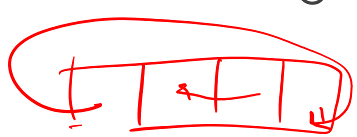
Division of Unsigned Binary Integers



Logical Operations

- NOT
- AND
- OR
- XOR
- Logical Shift Left
- Logical Shift Right
- Arithmetic Shift Left
- Arithmetic Shift Right
- Rotate Left/Right





Real Numbers

Unsigned integer		Integer	
Signed integer	Sign	Integer	
Unsigned fixed point		Integer	Fraction
Signed fixed point	Sign	Integer	Fraction

- Numbers with fractions
- Could be done in pure binary
  - $1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9.625$ 15.
- Where is the binary point?
  - Fixed?
    - Very limited
  - Moving?
    - How do you show where it is?

- **Another limitation:** Very large and very small numbers may not be represented
  - Solution in decimal is to use scientific notation
    - $976{,}000{,}000{,}000{,}000 = 9.76 \times 10^{14}$
    - $0.0000000000000976 = 9.76 \times 10^{-14}$

Floating-Point (IEEE 754 Standard)

n		0
Sign bit	Biased Exponent	Significand or Mantissa

- The same approach used with scientific notation:

  $$+/-\ \text{Significand} \times 2^{\text{Exponent}}$$

- **Binary point is fixed** between the sign bit and body of the mantissa
- **Exponent** indicates the point's place value *(point position)*

- **IEEE 754 Standards** for floating point storage:
  - Single Precision (32 bit): 1 sign bit, 8 bit exponent, and 23 bit mantissa
  - Double Precision (64 bit): 1 sign bit, 11 bit exponent, and 52 bit mantissa

Floating-Point (IEEE 754 Standard)

- **Bias** is a fixed value that is subtracted from the field to get the true exponent value
  - **Biased exponent** means exponent is **in excess** *(biased notation)*
  - **Bias** = $2^{k-1} - 1$
    - k = number of bits for the binary exponent

- **For single precision** (32 bits): $k = 8$
  - **Bias** = $2^7 - 1 = 127$ *(8 bit represents 0 to 255)*
  - **True exponent** values range from -127 to +128

- **Leading 1** of the significand is not stored for **normalized FP numbers**
  - Considered a *"hidden bit"*

Floating Point Examples (Single Precision)

(a) Format

32 bits

8 23

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}$

$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}$

$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}$

$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$

(b) Examples

Conversion Example 1

- Convert 100_{10} to **single precision** floating-point
- Convert to binary
  - $100_{10} = 0110\ 0100_2$

128	64	32	16	8	4	2	1
0	1	1	0	0	1	0	0

- In a binary representation form of 1.xxx:
  - $\cancel{0}110\ 0100 = 1.1001\cancel{00} \times 2^6 = 1.1001 \times 2^6$
  - **Exponent** = 6
  - **Biased Exponent** = 6 + 127 = 133 = 1000 0101 *(8 bits)*
  - **Sign** = 0 (positive)
  - **Significand/mantissa** = 100100 *(23 bits: pad with trailing 0s)*
    - **Normalized** - *leading 1 of the significand is not stored*

- **Final answer:**
  - 100_{10} = 0100 0010 1100 1000 …. *(32 bits in total)*
  - 100_{10} = **0x42C80000** *(in hex)*

Conversion Example 2

- Convert **-175_{10}** to **single precision** floating-point
  - -175_{10} = 128 32 8 4 2 1 $1010\ 1111_2$
  - 1010 1111 = 1.0101111 x 2^7
  - **Exponent** = 7
  - **Biased Exponent** = 7 + 127 = 134 = 128 4 2 1000 0110 *(8 bits)*
  - **Sign** = 1 (negative)
  - **Significand/mantissa** = 0101111 *(23 bits: pad with trailing 0s)*
    - **Normalized** - *leading 1 of the significand is not stored*

- **Final answer:**
  - -175_{10} = 1100 0011 0010 1111 0000 …. *(32 bits in total)*
  - -175_{10} = **0xC32F0000**

Example 3: Converting Back

- Convert **0xC32F0000** to **decimal**

- Extract components from **1100 0011 0010 1111**
  - **Sign** = 1 (negative)
  - **Biased Exponent** = 1000 0110 *(8 bits)* = 134_{10}
  - Unbias: 134 – 127 = 7
  - **Exponent** = 7
  - **Significand/mantissa** = 0101111 *(23 bits: remove trailing 0s)*
  - **Denormalized significand** = 1.0101111

- Move binary point by 7 *(exponent)* places = 1010 1111
- Convert to decimal
  - 128 32 8 4 2 1
  - $1010\ 1111_2 = 175_{10}$
  - Sign is negative, so **-175_{10}**

Conversion Example 4

- Convert **0x41C8000** to **decimal**
  - 0100 0001 1100 1~~000 0000~~…
  - **Sign** = 0 (positive)
  - **Biased Exponent** = 1000 0011 *(8 bits)* = 131_{10}
  - Unbias: 131 – 127 = 4
  - **Exponent** = 4
  - **Significand/mantissa** = 1001 *(23 bits: remove trailing 0s)*
  - **Denormalized significand** = 1.1001

16 8 1

- Move binary point by 4 *(exponent)* places = 11001
- 11001_2 = $\mathbf{25}_{10}$

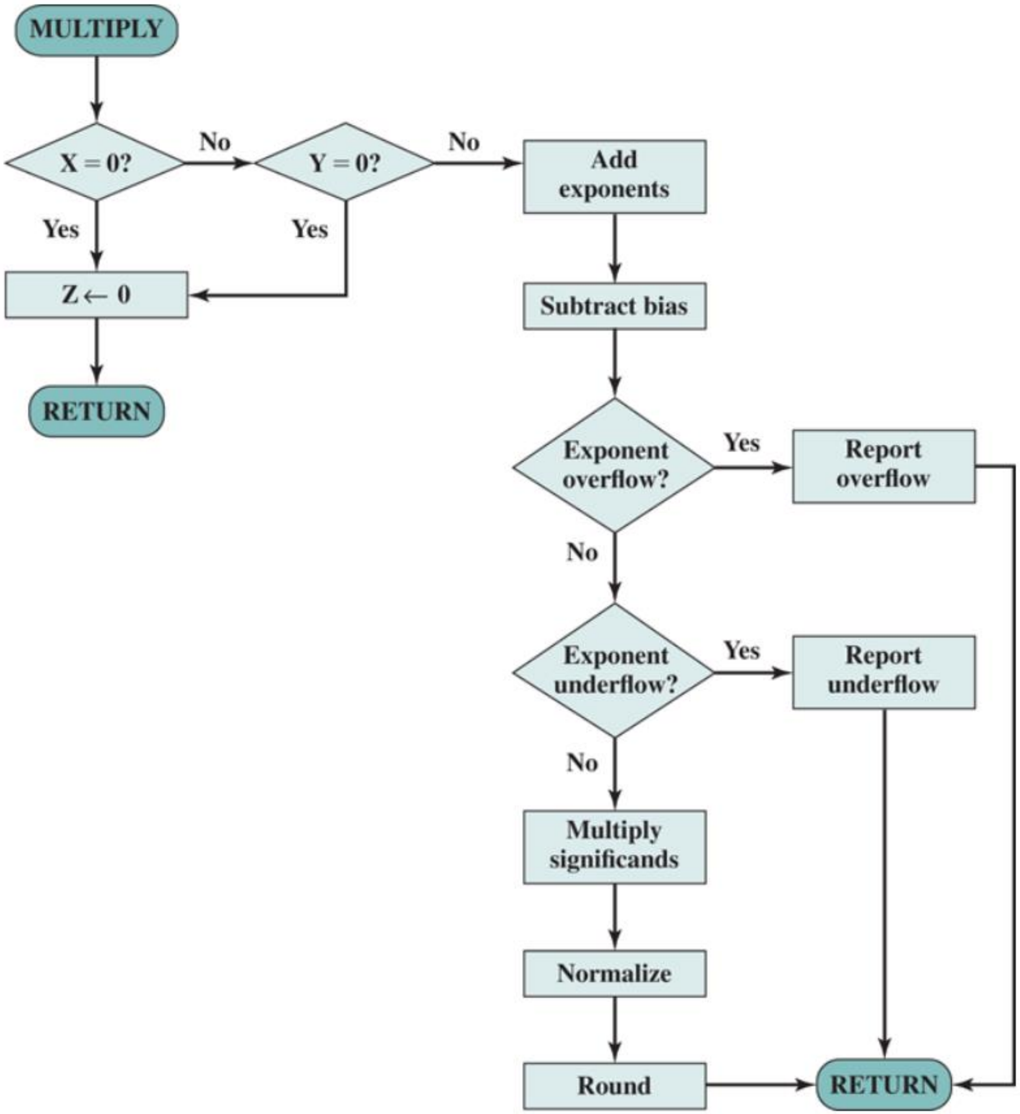
FP Arithmetic: ADD and SUB

- Check for zeros
- Align significands *(adjusting exponents)*
- Add or subtract significands
- Normalize result

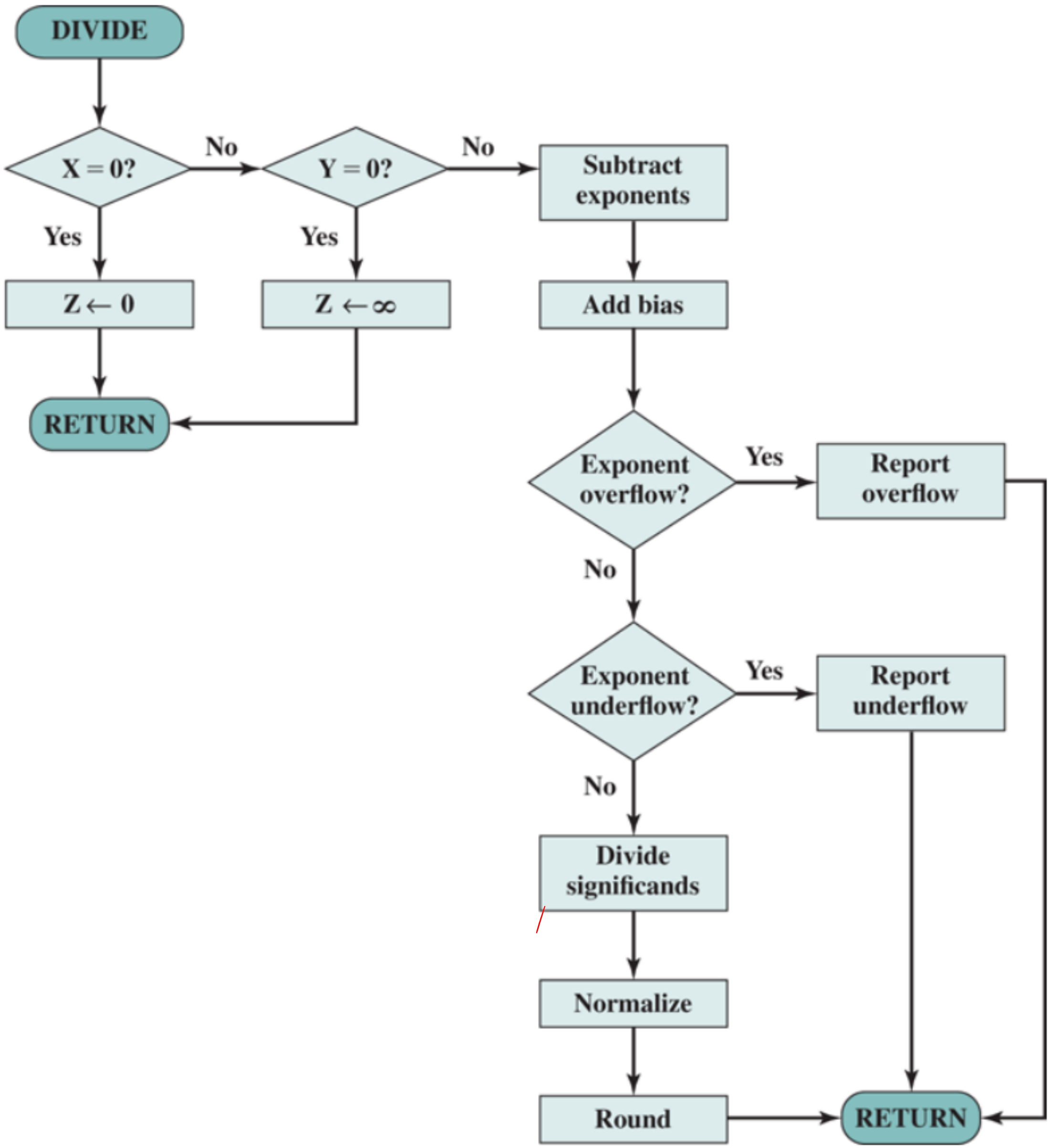
FP Arithmetic: MUL and DIV

- Check for zeros
- Add/subtract exponents
- Multiply/divide significands *(watch sign)*
- Normalize
- Round
- All intermediate results should be in double length storage

Floating-Point Multiplication



Floating-Point Division



University of San Carlos | Department of
COMPUTER ENGINEERING

CpE 3202
Computer Organization & Architecture

End of Lecture

References:

- Stallings, W. Computer Organization and Architecture, 6th edition, Pearson Education, Inc. (2003).
- Stallings, W. Computer Organization and Architecture, 11th edition, Pearson Education, Inc. (2019).
- https://www.tutorialspoint.com/fixed-point-and-floating-point-number-representations
- DeGroat, Joanne. "IEEE Floating Point“, Lecture Slides.