Computer Architecture

Computer Arithmetic

Lynn Choi

Korea University

高麗大學校

1905

Computer System Laboratory

Arithmetic & Logic Unit

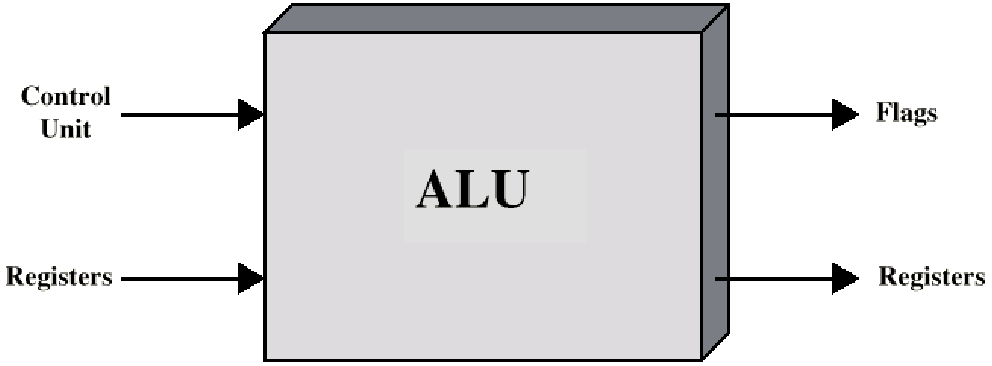
Roles of ALU

- Does the computations
- Everything else in the computer is there to service this unit
- Handles integers
- **FPU** (floating point unit) – arithmetic unit that handles floating point (real) numbers

Implementation

- All microprocessors has integer ALUs
- On-chip or off-chip FPU (co-processor)

ALU inputs and outputs



Integer Representation

- **Only have 0 & 1 to represent everything**
- **Two representative representations**
  - Sign-magnitude
  - Two′s compliment
- **Sign-magnitude**
  - Left most bit is sign bit
    - 0 means positive
    - 1 means negative
  - Example
    - +18 = 00010010
    - -18 = 10010010
  - Problems
    - Need to consider both sign and magnitude in arithmetic
    - Two representations of zero (+0 and -0)

2's Complement

- **Given N, 2's complement of N with n bits**
  - $2^n - N = \underline{(2^n - 1) - N} + 1 = \underline{\text{bit complement of N}} + 1$
  - 32 bit number
    - Positive numbers : 0 (x00000000) to $2^{31} - 1$ (x7FFFFFFF)
    - Negative numbers : -1 (xFFFFFFFF) to -2^{31} (x8000000)
  - Like sign-magnitude, MSB represents the sign bit
- **Examples**
  - +3 = 011
  - +2 = 010
  - +1 = 001
  - +0 = 000
  - -1 = 111
  - -2 = 110
  - -3 = 101
  - -4 = 100

Characteristics of 2's Complement

- **A single representation of zero**
- **Negation is fairly easy (bit complement of N + 1)**
  - 3 = 00000011
  - Boolean complement gives 11111100
  - Add 1 to LSB 11111101
- **Overflow occurs only**
  - When the sign bit of two numbers are the same but the result has the opposite sign ($V = C_n \oplus C_n$-1)

Operation	*A*	*B*	*Overflow Condition*
A + B	+	+	-
A + B	-	-	+
A - B	+	-	-
A - B	-	+	+

- **Arithmetic works easily (see later)**
  - To perform A – B, take the 2′s complement of B and add it to A
  - $A + (2^n - B) = A - B + 2^n$ (if A >= B, ignore the carry)
    $= 2^n - (B - A)$ (if B > A, 2′s complement of B – A)

Range of Numbers

- **8 bit 2's complement**
  - $+127 = 01111111 = 2^7 - 1$
  - $-128 = 10000000 = -2^7$
- **16 bit 2's complement**
  - $+32767 = 011111111\ 11111111 = 2^{15} - 1$
  - $-32768 = 100000000\ 00000000 = -2^{15}$
- **N bit 2's complement**
  - $-2^{n-1} \sim 2^{n-1} - 1$

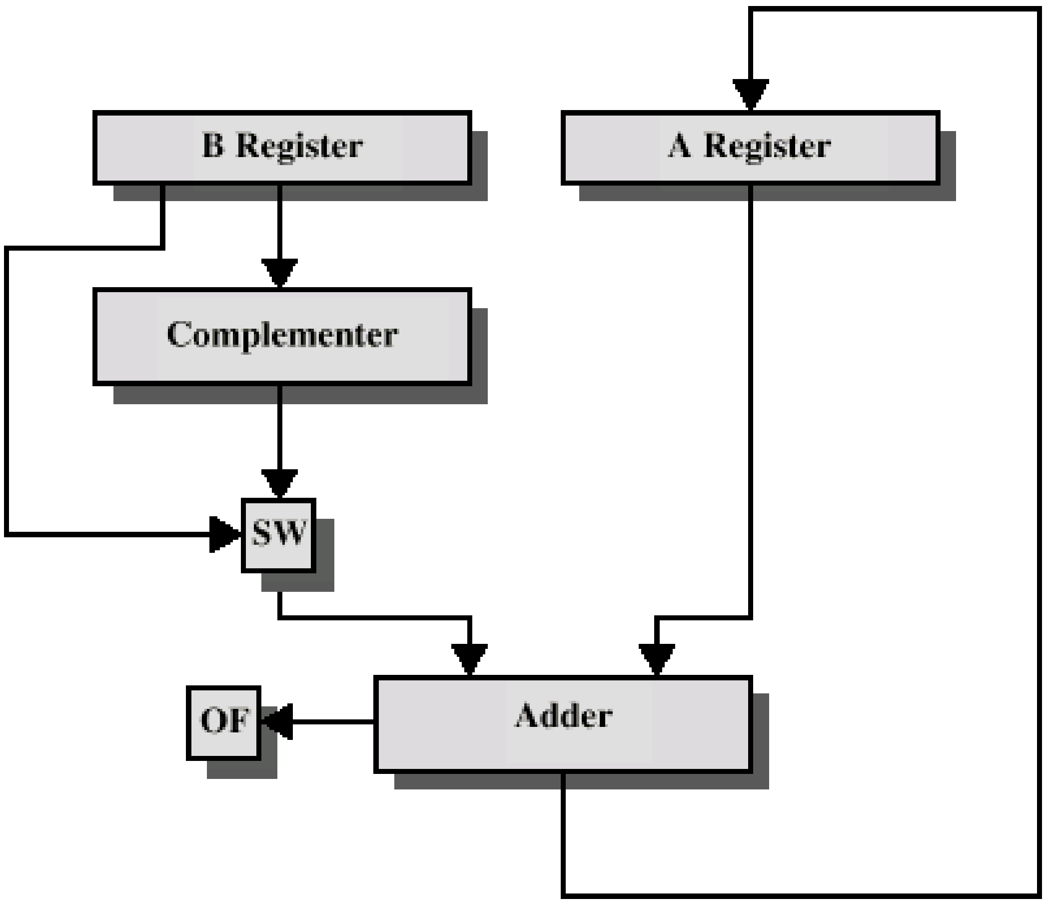
Conversion Between Lengths

- **Positive number – pack with leading zeros**
  - +18 = 00010010
  - +18 = 00000000 00010010
- **Negative numbers – pack with leading ones**
  - -18 = 10010010
  - -18 = 11111111 10010010
- **Sign-extension**
  - i.e. pack with MSB (sign bit)

Addition and Subtraction

- **Addition**
  - Normal binary addition
  - Monitor sign bit for overflow
- **Subtraction**
  - Take the two's complement of subtrahend and add to minuend
    - i.e. a - b = a + (-b)
  - So we only need adder and complement circuits

Hardware for Addition and Subtraction



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

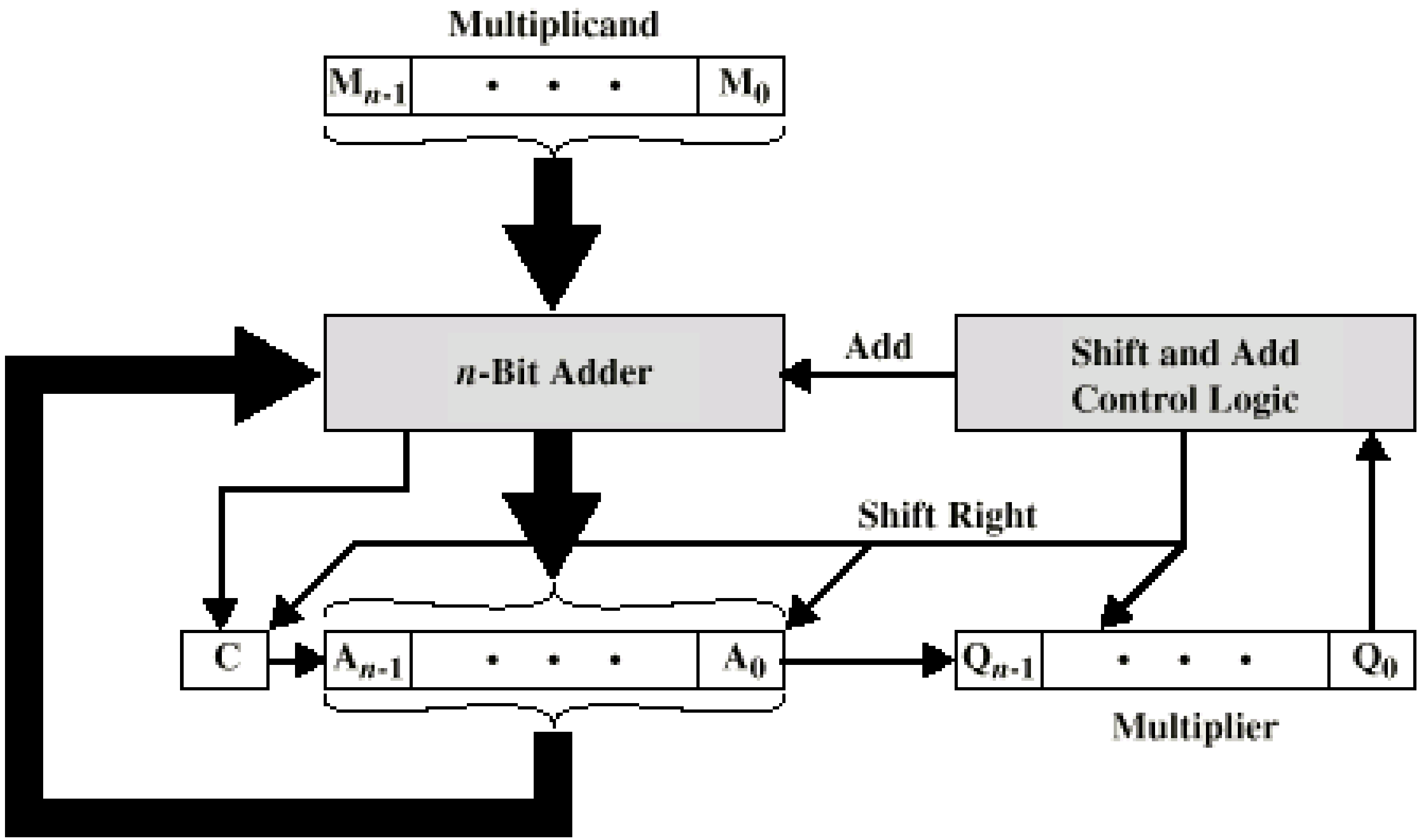
Multiplication

- **Example**

1011	Multiplicand (11 decimal)
x 1101	Multiplier (13 decimal)
1011	Partial products
0000	**Note:** if multiplier bit is 1 then copy multiplicand (place value)
1011	otherwise put zero
1011	
10001111	Product (143 decimal)

- **Principles**
  - Work out partial product for each digit
  - Shift each partial product
  - Add partial products
  - Note: need double length result

Binary Multiplier (Unsigned)

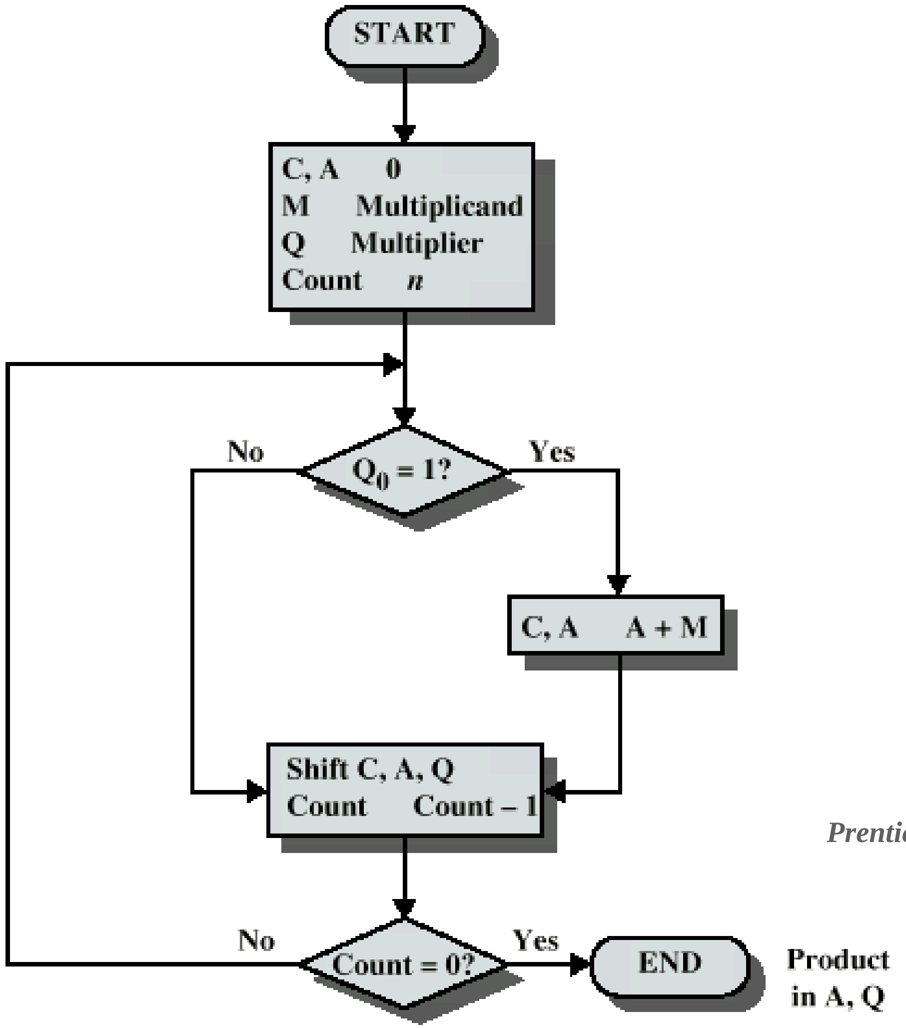


(a) Block Diagram

Execution of Example

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	

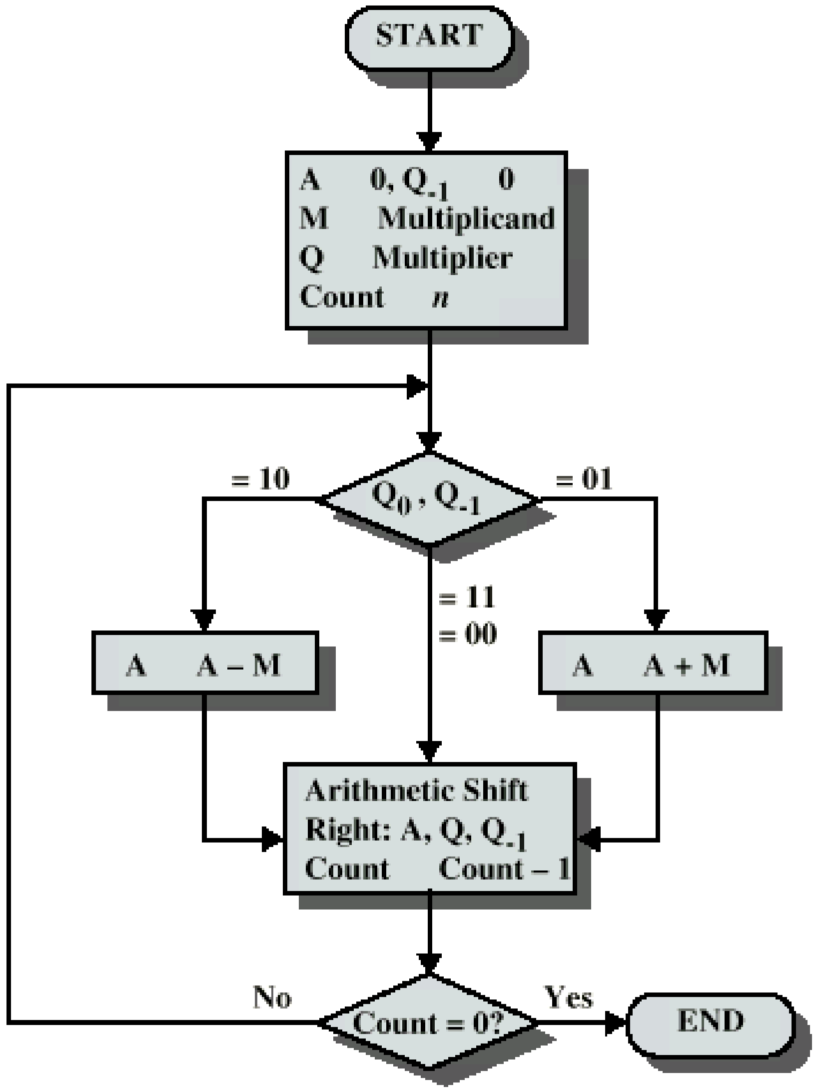
Flowchart for Unsigned Binary Multiplication



Signed Multiplication

- **Unsigned binary multiplication algorithm**
  - Does not work for signed multiplication!
- **Solution 1**
  - Convert to positive if required
  - Multiply as above
  - If signs were different, negate answer
- **Solution 2**
  - Booth's algorithm

Booth's Algorithm



Example of Booth's Algorithm

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	A A − M	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	A A + M	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Examples

```
    0111
   ×0011      (0)
--------
11111001      1-0
0000000       1-1
000111        0-1
--------
00010101      (21)
```

(a) (7) × (3) = (21)

```
    0111
   ×1101      (0)
--------
11111001      1-0
0000111       0-1
111001        1-0
--------
11101011      (-21)
```

(b) (7) × (–3) = (–21)

```
    1001
   ×0011      (0)
--------
00000111      1-0
0000000       1-1
111001        0-1
--------
11101011      (-21)
```

(c) (–7) × (3) = (–21)

```
    1001
   ×1101      (0)
--------
00000111      1-0
1111001       0-1
000111        1-0
--------
00010101      (21)
```

(d) (–7) × (–3) = (21)

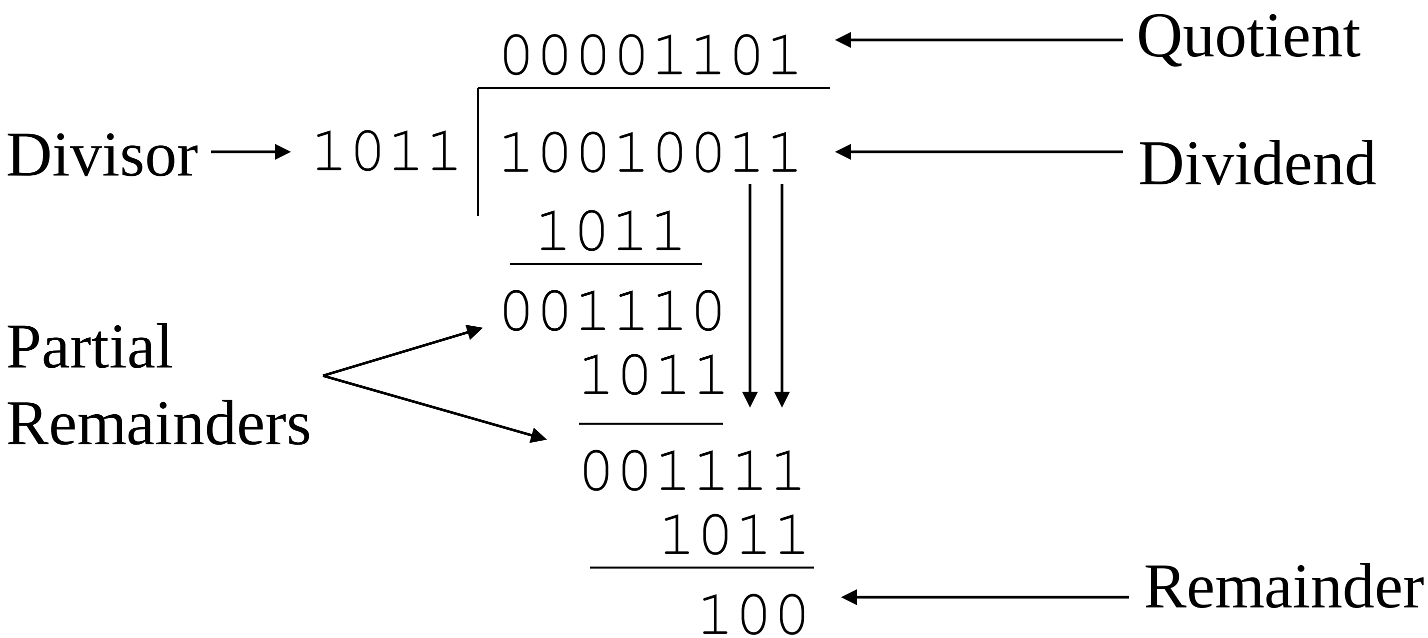
Figure 9.14 Examples Using Booth's Algorithm

Division

- **Unsigned binary division**
  - Can be implemented by shift and subtract
- **Signed binary division**
  - More complex than multiplication
  - The unsigned binary division algorithm can be extended to negative numbers.

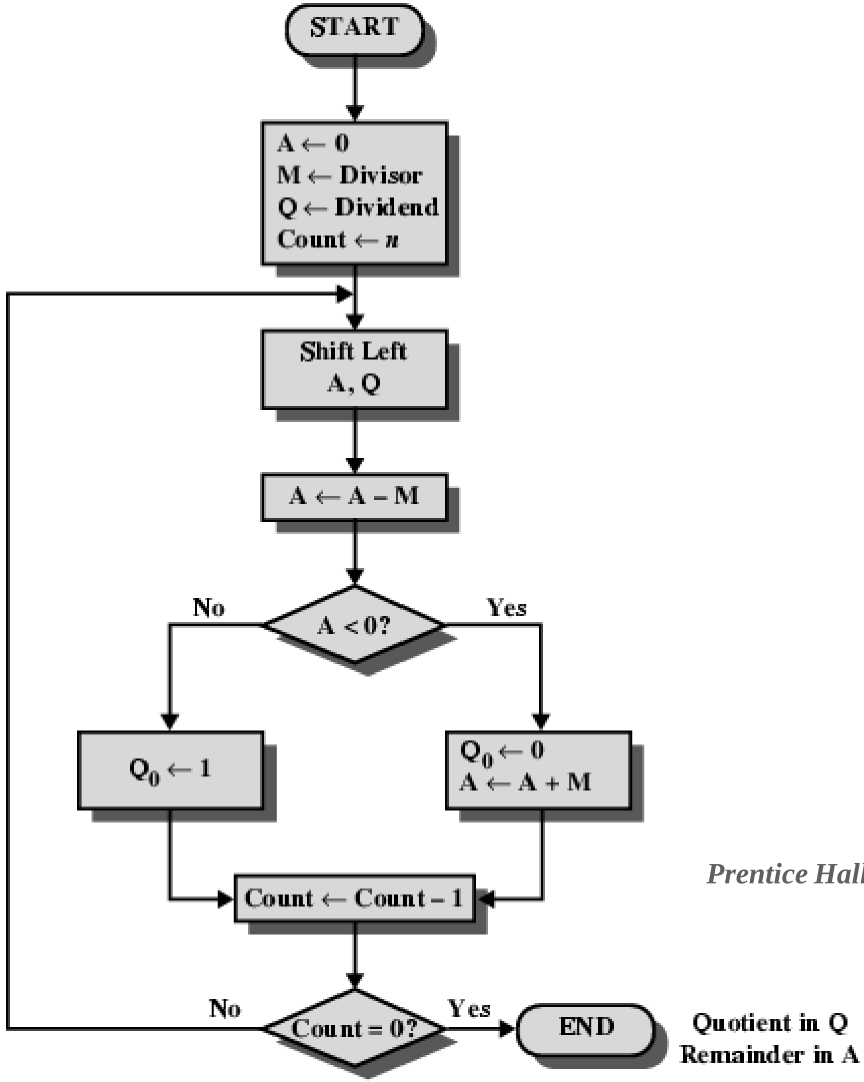
Division of Unsigned Binary Integers

- **Unsigned binary division**
  - **Can be implemented by shift and subtract**
  - **The multiplication hardware can be used for the division as well**



Dividend = Quotient * Divisor + Remainder

Flowchart for Unsigned Binary Division



Signed Division

- **Signed binary division**
  - **More complex than multiplication**
  - **The unsigned binary division algorithm can be extended to negative numbers.**
    1. **Load the divisor into M and the dividend into A, Q**
       - *The dividend must be expressed as a 2n-bit 2′s complement number*
    2. **Shift A, Q left by 1 bit position**
    3. **If M and A have the same signs, perform A <- A – M; otherwise A + M**
    4. **If the sign of A is the same as before or A = 0, Q0 <- 1; Otherwise Q0 <- 0 and restore the previous value of A**
    5. **Repeat 2 through 4 n times**
    6. **Remainder in A. If the signs of the divisor and dividend are the same, the quotient is in Q; Otherwise, the quotient is the 2′s complement of Q**

Examples of Signed Division

A	Q	M = 0011
0000	0111	Initial value
0000	1110	shift
1101		subtract
0000	1110	restore
0001	1100	shift
1110		subtract
0001	1100	restore
0011	1000	shift
0000		subtract
0000	1001	set $Q_0 = 1$
0001	0010	shift
1110		subtract
0001	0010	restore

(a) (7)/(3)

A	Q	M = 1101
0000	0111	Initial value
0000	1110	shift
1101		add
0000	1110	restore
0001	1100	shift
1110		add
0001	1100	restore
0011	1000	shift
0000		add
0000	1001	set $Q_0 = 1$
0001	0010	shift
1110		add
0001	0010	restore

(b) (7)/(–3)

A	Q	M = 0011
1111	1001	Initial value
1111	0010	shift
0010		add
1111	0010	restore
1110	0100	shift
0001		add
1110	0100	restore
1100	1000	shift
1111		add
1111	1001	set $Q_0 = 1$
1111	0010	shift
0010		add
1111	0010	restore

(c) (–7)/(3)

A	Q	M = 1101
1111	1001	Initial value
1111	0010	shift
0010		subtract
1111	0010	restore
1110	0100	shift
0001		subtract
1110	0100	restore
1100	1000	shift
1111		subtract
1111	1001	set $Q_0 = 1$
1111	0010	shift
0010		subtract
1111	0010	restore

(d) (–7)/(–3)

Real Numbers

- **Numbers with fractions**
- **Could be done in pure binary**
  - $1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9.625$
- **Where is the binary point?**
- **Fixed? (Fixed-point)**
  - Very limited
    - Very large numbers cannot be represented
    - Very small fractions cannot be represented
    - The same applies to results after computation
- **Moving? (Floating-point)**
  - How do you show where it is?
  - Use the exponent to slide (place) the binary point
  - Example
    - $976,000,000,000,000 = 9.76 * 10^{14}$
    - $0.0000000000000976 = 9.76 * 10^{-14}$

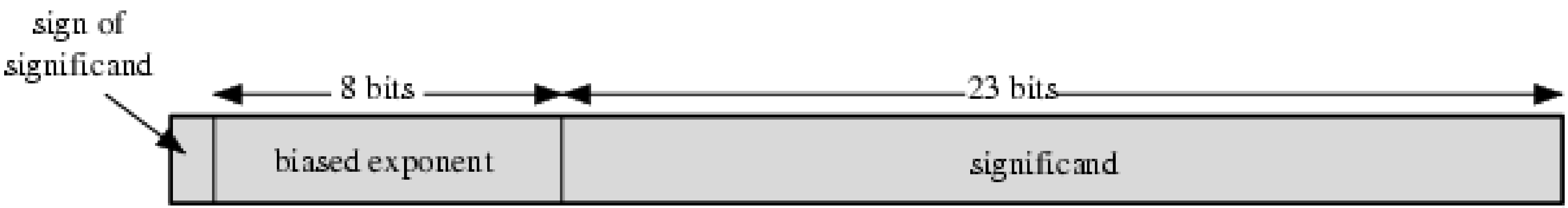
Floating Point Representation

Sign bit	Biased Exponent (E)	Significand or Mantissa (S)

$\pm S \times B^{\pm E}$

- Point is actually fixed between sign bit and body of mantissa
- Exponent indicates place value (point position)
- Base B
  - Implicit and need not be stored since it is the same for all numbers
- Exponent E
  - Biased representation
    - A fixed value called bias (typically $2^{k-1} - 1$ when k is the length of the exponent) is subtracted to get the true exponent value
    - For 8-bit exponent, a bias of 127 is used and can represent −127 to 128
    - Nonnegative FP numbers can be treated as integers for comparison purposes
- Significand (or Mantissa) S
  - Normalized representation
    - The most significant digit of the significand is nonzero
    - $+/- 1.bbb...b \times 2^{+/-E}$
    - Since the MSB is always 1, it is unnecessary to store this bit
    - Thus, a 23-bit significand is used to store a 24-bit significand with a value in [1, 2)

Floating Point Examples

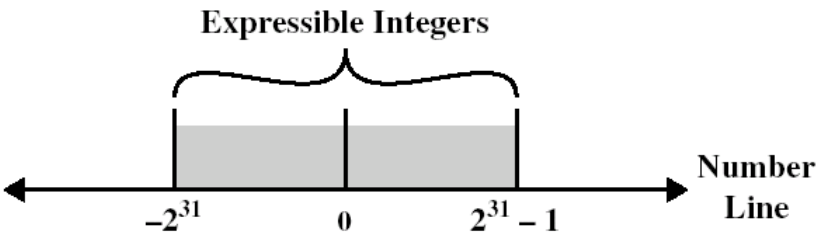


(a) Format

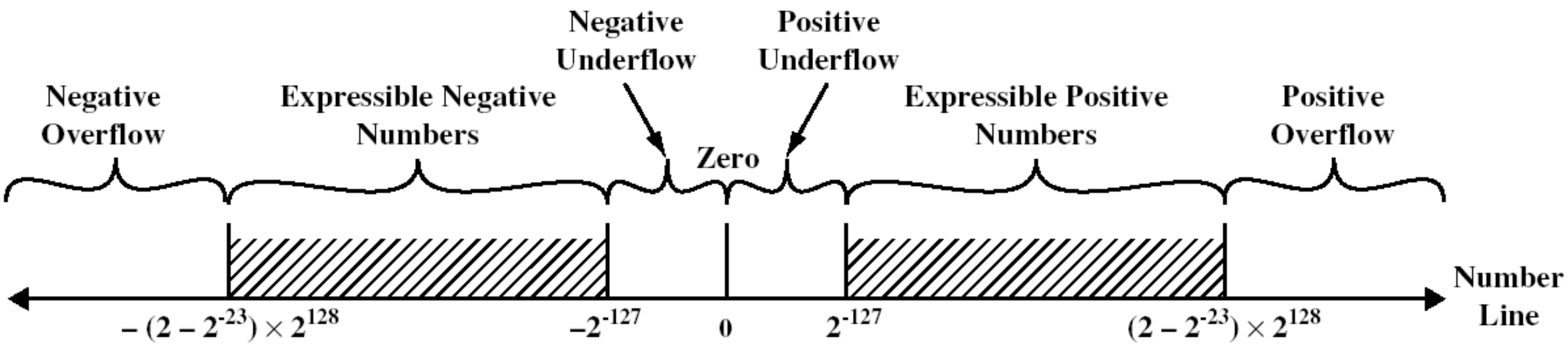
$$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.638125 \times 2^{20}$$
$$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.638125 \times 2^{20}$$
$$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.638125 \times 2^{-20}$$
$$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.638125 \times 2^{-20}$$

(b) Examples

Expressible Numbers

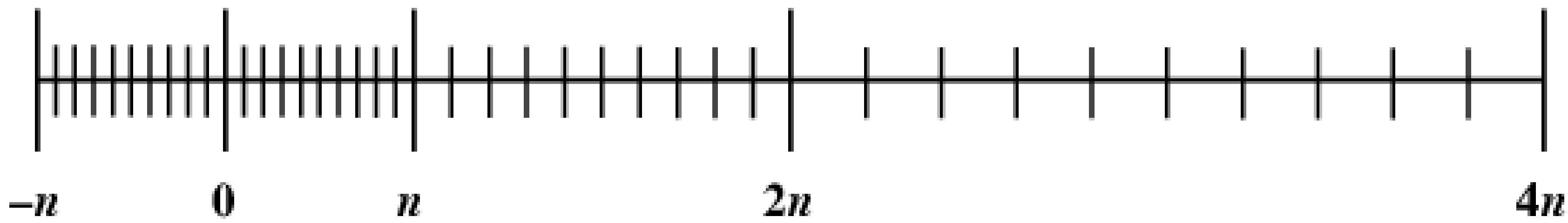


(a) Twos Complement Integers



(b) Floating-Point Numbers

Density of FP Numbers



- **Note that**
  - **The maximum number of different values that can be represented with 32 bits is still 2^{32}.**
  - **FP numbers are not spaced evenly along the number line**
    - **Larger numbers are spaced more sparsely than smaller numbers**

IEEE 754

- **Standard for floating point numbers**
  - To facilitate the portability of FP programs among different processors
  - Supported by virtually all commercial microprocessors
- **IEEE 754 formats**
  - 32-bit single precision
    - 8b exponent, 23b fraction
  - 64-bit double precision
    - 11b exponent, 52b fraction
  - Extended precision : double-extended
- **Characteristics**
  - Range of exponents : single (-126 ~ 127), double (-1022 ~ 1023)
  - Zero is represented by all 0′s (exponent 0 and fraction 0)
  - An exponent of all 1′s with a fraction of 0 represents $+\infty$, $-\infty$
  - An exponent of 0 with a nonzero fraction represents a denormalized number
  - An exponent of all 1′s with a nonzero fraction represents a NaN (Not a Number) which is used to signal various exceptions

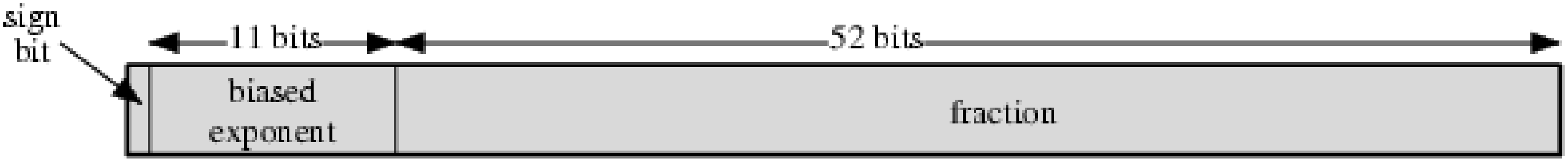
NaN (Not a Number)

The following practices may cause NaNs.

- All mathematical operations with a NaN as at least one operand
- The divisions 0/0, ∞/∞, ∞/-∞, -∞/∞, and -∞/-∞
- The multiplications $0 \times \infty$ and $0 \times -\infty$
- The additions $\infty + (-\infty)$, $(-\infty) + \infty$ and equivalent subtractions.
- Applying a function to arguments outside its domain
  - Taking the square root of a negative number
  - Taking the logarithm of zero or a negative number
  - Taking the inverse sine or cosine of a number which is less than -1 or greater than +1.

IEEE 754 Formats

(a) Single format



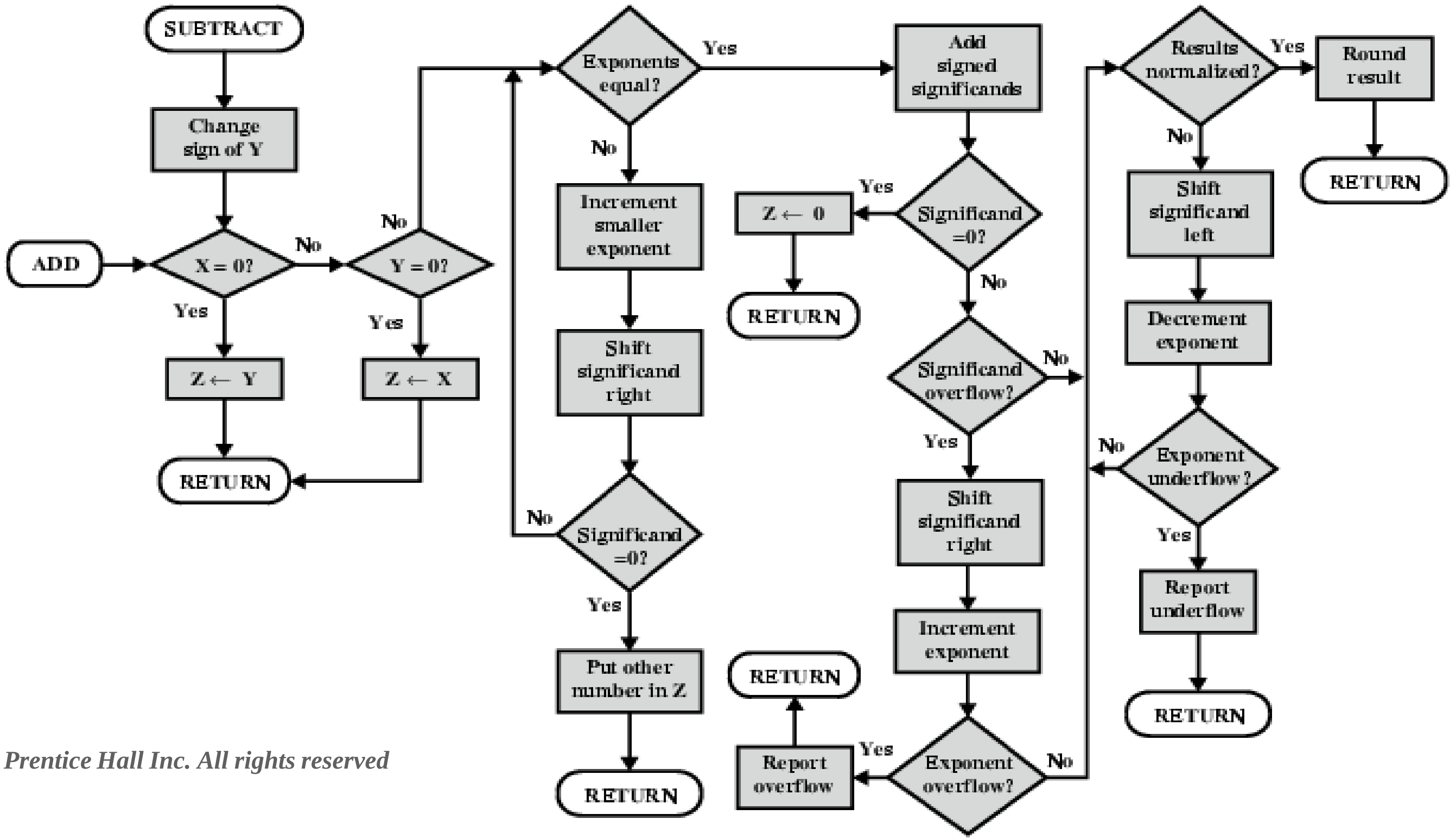
(b) Double format

FP Arithmetic +/-

4 Phases

- Check for zeros
- Align the significand of a smaller number (adjust the exponent)
- Add or subtract the significands
- Normalize the result

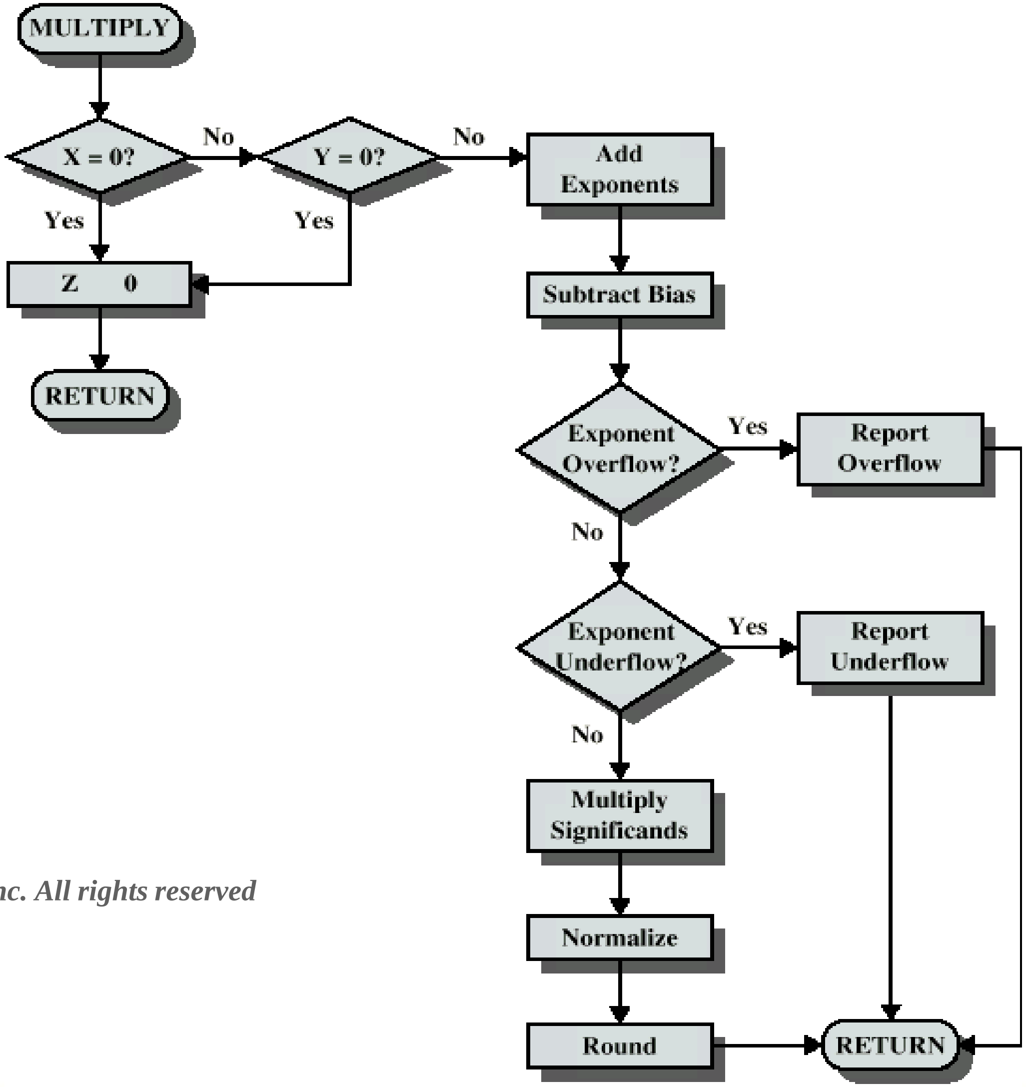
FP Addition & Subtraction Flowchart



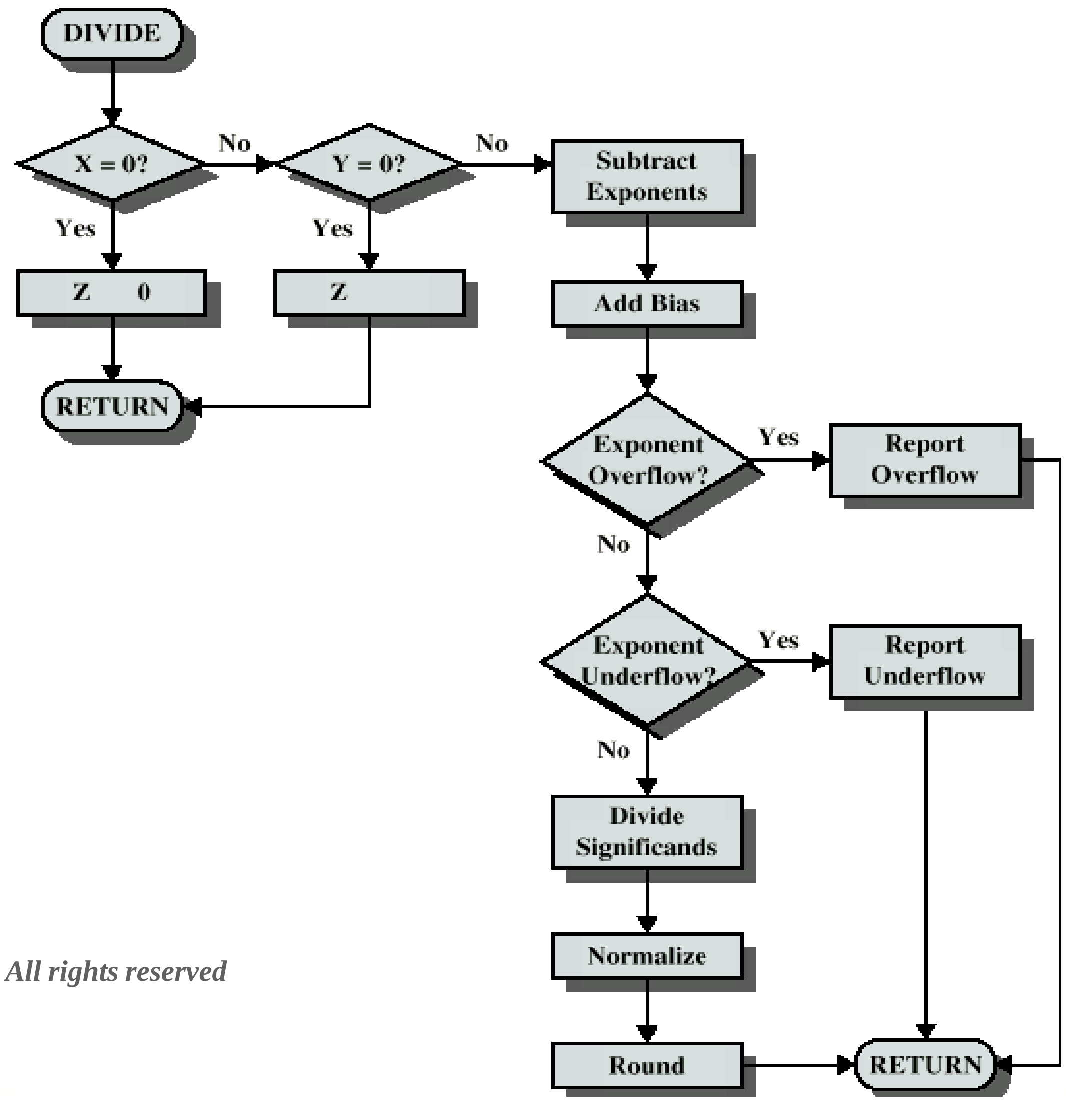
FP Arithmetic x/÷

- **Consists of the following phases**
  - Check for zero
  - Add/subtract exponents
  - Multiply/divide significands (watch sign)
  - Normalize and round

Floating Point Multiplication



Floating Point Division



Homework 4

- Read Chapter 4 (from Computer Organization and Design Textbook)
- Exercise
  - 3.2
  - 3.4
  - 3.6
  - 3.8
  - 3.12