

# Computer Arithmetic

## Chapter 10

Dr. Ronald H.Y. Chung

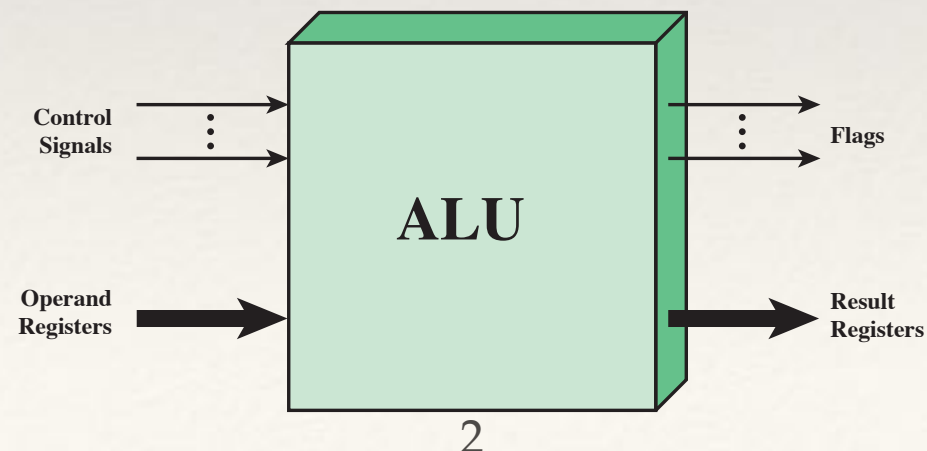


THE UNIVERSITY OF HONG KONG

DEPARTMENT OF  
COMPUTER SCIENCE

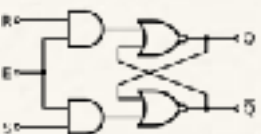
# Arithmetic & Logic Unit (ALU)

- ❖ Part of the computer that actually performs arithmetic and logical operations on data
- ❖ All of the other elements of the computer system are there mainly to bring data into the ALU for it to process and then to take the results back out
- ❖ Based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations
- ❖ Operands for arithmetic and logic operations are presented to the ALU in registers, and the results of an operation are stored in registers.
- ❖ The ALU may also set flags as the result of an operation
- ❖ An overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored.
  - ❖ The flag values are also stored in registers within the processor.
- ❖ The processor provides signals that control the operation of the ALU and the movement of the data into and out of the ALU

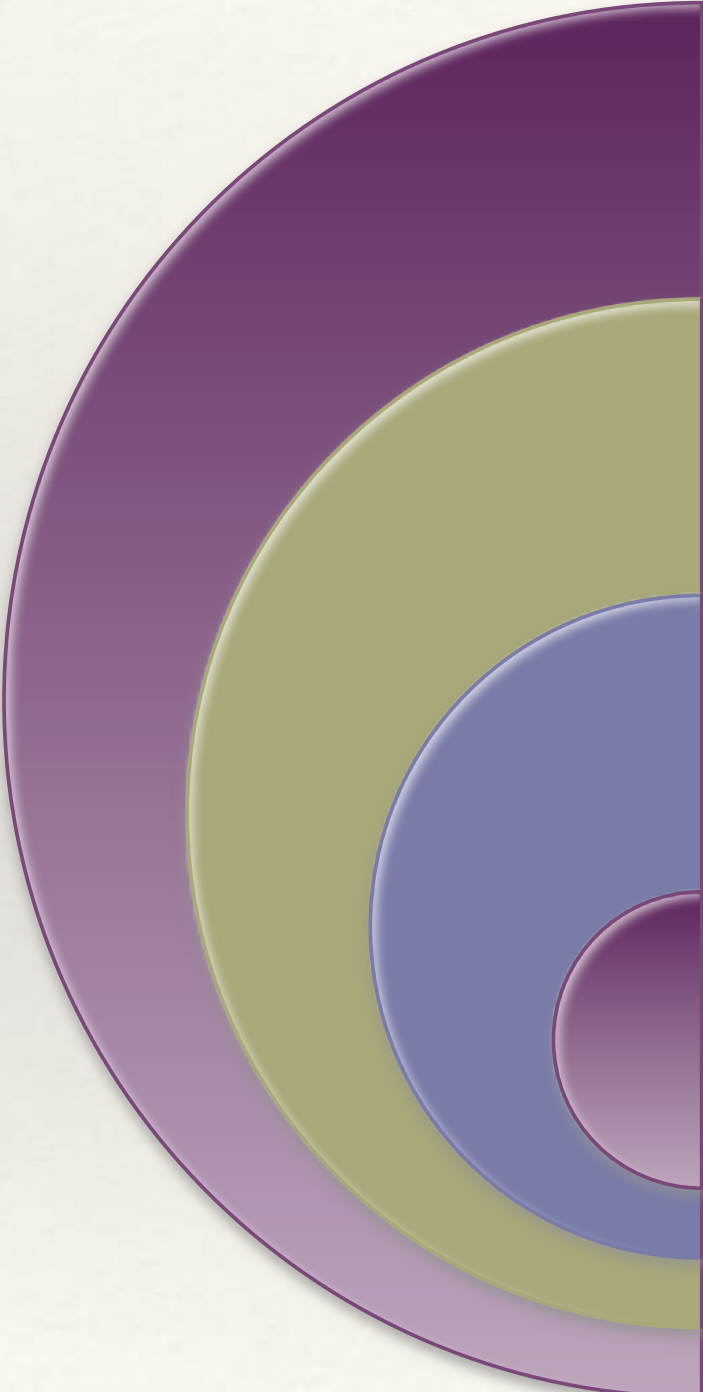


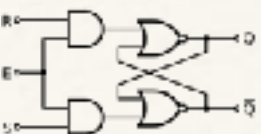
# Integer Representation

- ❖ In the binary number system arbitrary numbers can be represented with:
  - ❖ The digits zero and one
  - ❖ The minus sign (for negative numbers)
  - ❖ The period, or *radix point* (for numbers with a fractional component)
- ❖ For purposes of computer storage and processing we do not have the benefit of special symbols for the minus sign and radix point
- ❖ Only binary digits (0,1) may be used to represent numbers



# Sign-Magnitude Representation

	<p>There are several alternative conventions used to represent negative as well as positive integers</p>	<ul style="list-style-type: none"><li>• All of these alternatives involve treating the most significant (leftmost) bit in the word as a sign bit</li><li>• If the sign bit is 0 the number is positive</li><li>• If the sign bit is 1 the number is negative</li></ul>
	<p>Sign-magnitude representation is the simplest form that employs a sign bit</p>	
	<p>Drawbacks:</p>	<ul style="list-style-type: none"><li>• Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation</li><li>• There are two representations of 0</li></ul>
	<p>Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU</p>	



# Characteristics of Twos Complement Representation and Arithmetic

<b>Range</b>	$-2^{n-1}$ through $2^{n-1} - 1$
<b>Number of Representations of Zero</b>	One
<b>Negation</b>	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
<b>Expansion of Bit Length</b>	Add additional bit positions to the left and fill in with the value of the original sign bit.
<b>Overflow Rule</b>	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
<b>Subtraction Rule</b>	To subtract $B$ from $A$ , take the twos complement of $B$ and add it to $A$ .





# Different Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
-0	1000	—	—
-1	1001	1111	0110
-2	1010	1110	0101
-3	1011	1101	0100
-4	1100	1100	0011
-5	1101	1011	0010
-6	1110	1010	0001
-7	1111	1001	0000
-8	—	1000	—

Unsigned Representation

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

Sign-Magnitude Representation

$$A = (1 - 2a_{n-1}) \sum_{i=0}^{n-2} 2^i a_i$$

Twos Complement Representation

$$A = (-2^{n-1} a_{n-1}) + \sum_{i=0}^{n-2} 2^i a_i$$

Biased Representation

$$A = \sum_{i=0}^{n-1} 2^i a_i - 2^{n-1} + 1$$



# Conversion between 2's Complement Binary and Decimal

- ❖ A useful illustration of the nature of twos complement representation is a value box
  - ❖ The value on the far right in the box is 1 ( $2^0$ ) and each succeeding position to the left is double in value
  - ❖ The leftmost position is negated
- ❖ The range of number that can be represented is  $-2^{n-1}$  to  $2^{n-1} - 1$

-128	64	32	16	8	4	2	1

(a) An eight-position two's complement value box

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

$$-128 \quad +2 \quad +1 \quad = -125$$

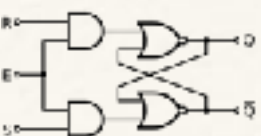
(b) Convert binary 10000011 to decimal

−128	64	32	16	8	4	2	1
1	0	0	0	1	0	0	0

$$-120 = -128 + 8$$

(c) Convert decimal  $-120$  to binary

### Figure 10.2 Use of a Value Box for Conversion Between Twos Complement Binary and Decimal



# Range Extension

- ❖ Range of numbers that can be expressed is extended by increasing the bit length
- ❖ In sign-magnitude notation this is accomplished by moving the sign bit to the new leftmost position and fill in with zeros
- ❖ This procedure will not work for twos complement negative integers
  - ❖ Rule is to move the sign bit to the new leftmost position and fill in with copies of the sign bit
  - ❖ For positive numbers, fill in with zeros, and for negative numbers, fill in with ones
  - ❖ This is called *sign extension*

+18	=	00010010	(sign magnitude, 8 bits)
+18	=	00000000000010010	(sign magnitude, 16 bits)
-18	=	10010010	(sign magnitude, 8 bits)
-18	=	10000000000010010	(sign magnitude, 16 bits)

+18	=	00010010	(twos complement, 8 bits)
+18	=	00000000000010010	(twos complement, 16 bits)
-18	=	11101110	(twos complement, 8 bits)
-32,658	=	1000000001101110	(twos complement, 16 bits)

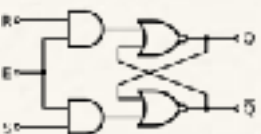
-18	=	11101110	(twos complement, 8 bits)
-18	=	1111111111101110	(twos complement, 16 bits)



# Fixed-Point Representation

The radix point (binary point) is fixed and assumed to be to the right of the rightmost digit

Programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location



# Negation

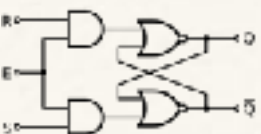
- ❖ Twos complement operation

- ❖ Take the Boolean complement of each bit of the integer (including the sign bit)
- ❖ Treating the result as an unsigned binary integer, add 1

$$\begin{aligned} +18 &= 00010010 \text{ (twos complement)} \\ \text{bitwise complement} &= 11101101 \\ &+ \quad \quad \quad 1 \\ \hline &11101110 = -18 \end{aligned}$$

- ❖ The negative of the negative of that number is itself:

$$\begin{aligned} -18 &= 11101110 \text{ (twos complement)} \\ \text{bitwise complement} &= 00010001 \\ &+ \quad \quad \quad 1 \\ \hline &00010010 = +18 \end{aligned}$$



# Negation Special Cases

0 = 00000000 (twos complement)

Bitwise complement = 11111111

Add 1 to LSB + 1

Result 100000000

Overflow is ignored, so:

$$-0 = 0$$

-128 = 10000000 (twos complement)

Bitwise complement = 01111111

Add 1 to LSB + 1

Result 10000000

So:

$$-(-128) = -128 \text{ X}$$

Monitor MSB (sign bit)

It should change during negation



# Additions of Numbers in 2's Complement Representation

- ❖ Addition proceeds as if the two numbers were unsigned integers.
  - ❖ The first four examples illustrate successful operations.
  - ❖ If the result of the operation is positive, we get a positive number in twos complement form, which is the same as in unsigned-integer form.
  - ❖ If the result of the operation is negative, we get a negative number in twos complement form.
- ❖ Note that, in some instances, there is a carry bit beyond the end of the word (indicated by shading), which is ignored.
- ❖ **Overflow Rule:**
  - ❖ *If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.*

$$\begin{array}{r} 1001 = -7 \\ +0101 = 5 \\ \hline 1110 = -2 \end{array}$$

(a)  $(-7) + (+5)$

$$\begin{array}{r} 1100 = -4 \\ +0100 = 4 \\ \hline 10000 = 0 \end{array}$$

(b)  $(-4) + (+4)$

$$\begin{array}{r} 0011 = 3 \\ +0100 = 4 \\ \hline 0111 = 7 \end{array}$$

(c)  $(+3) + (+4)$

$$\begin{array}{r} 1100 = -4 \\ +1111 = -1 \\ \hline 11011 = -5 \end{array}$$

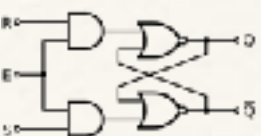
(d)  $(-4) + (-1)$

$$\begin{array}{r} 0101 = 5 \\ +0100 = 4 \\ \hline 1001 = \text{Overflow} \end{array}$$

(e)  $(+5) + (+4)$

$$\begin{array}{r} 1001 = -7 \\ +1010 = -6 \\ \hline 10011 = \text{Overflow} \end{array}$$

(f)  $(-7) + (-6)$





# Subtraction Rule

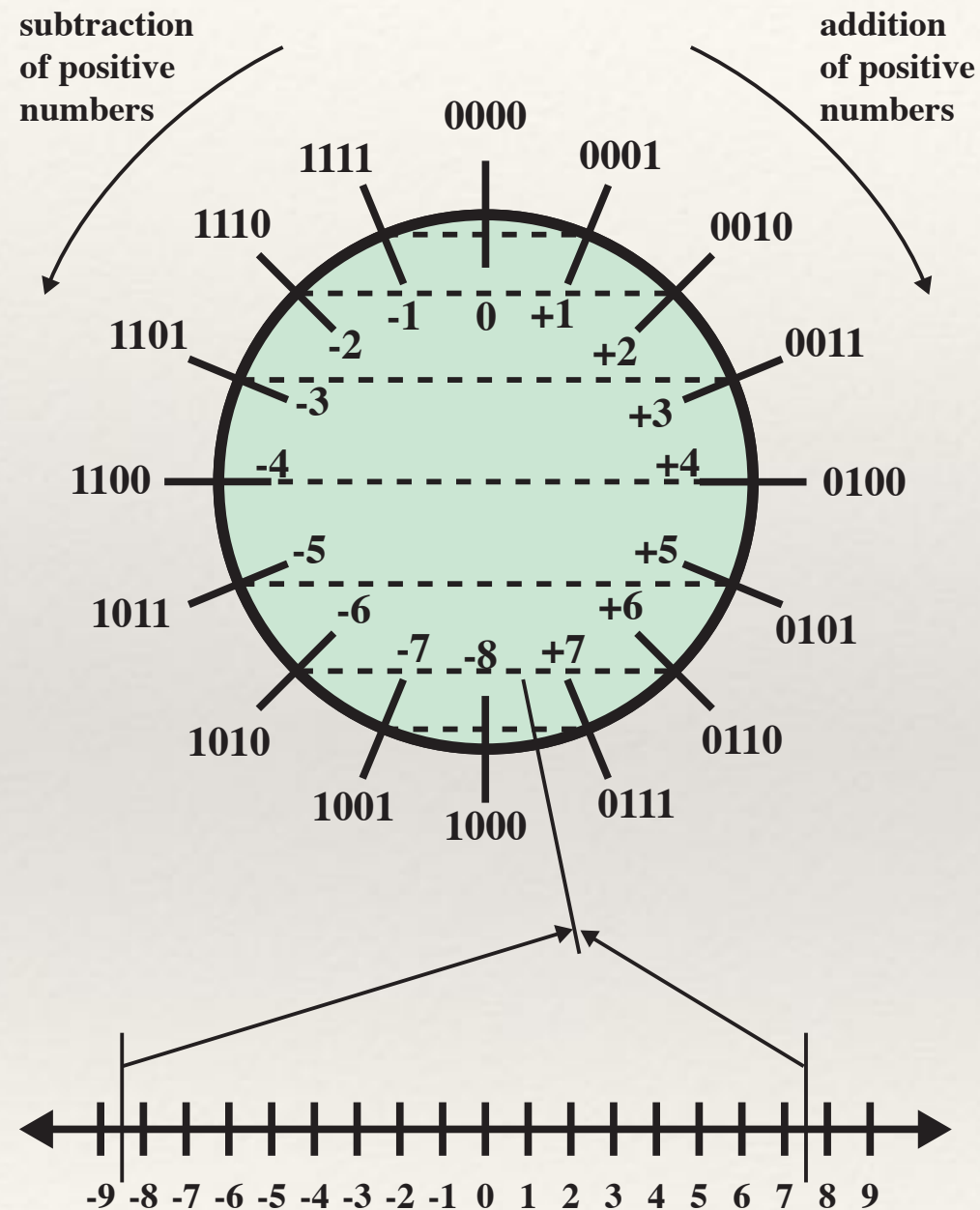
## ❖ Subtraction Rule:

- ❖ To subtract one number (subtrahend) from another (minuend), take the two's complement (negation) of the subtrahend and add it to the minuend.

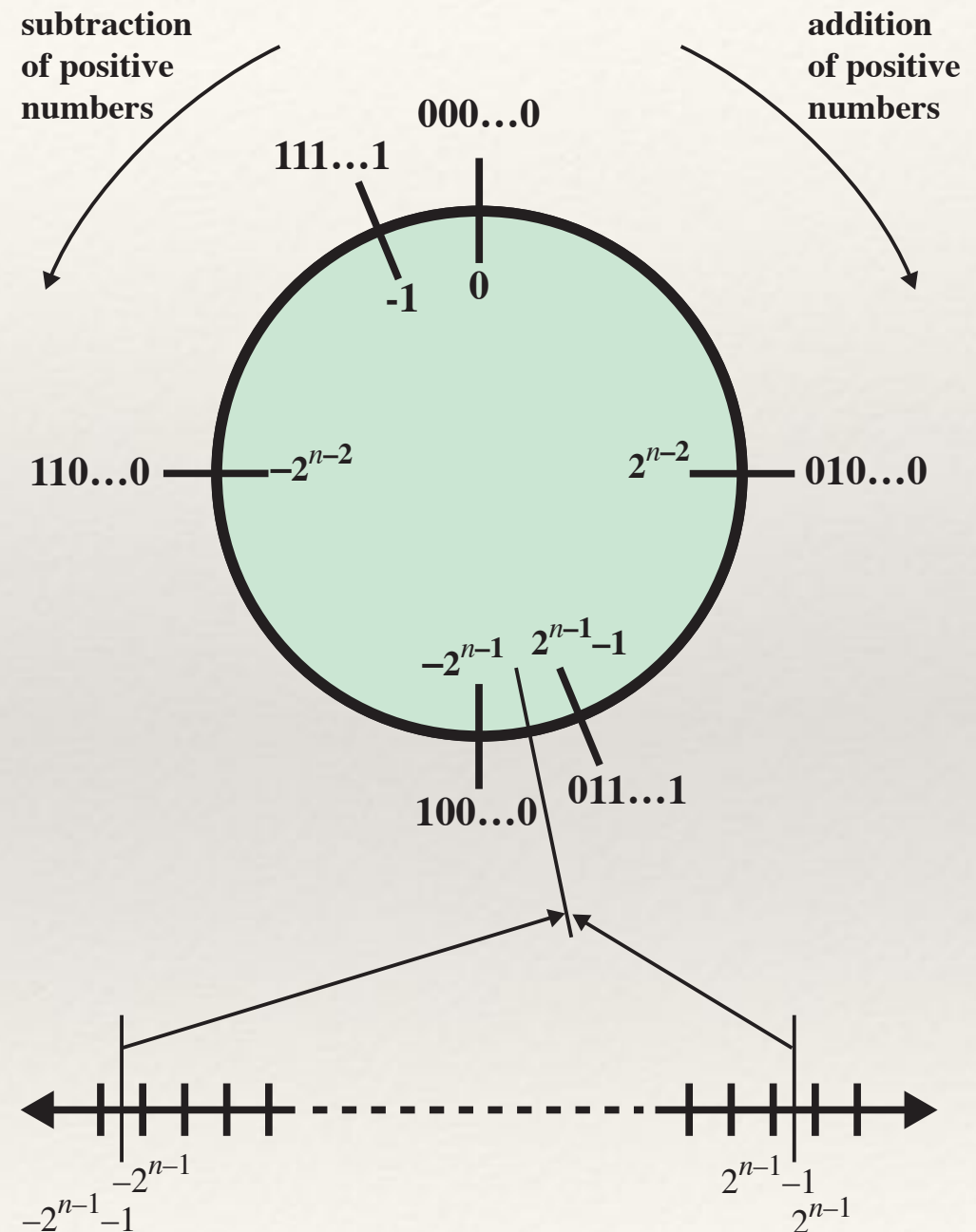
$\begin{array}{r} 0010 = 2 \\ +1001 = -7 \\ \hline 1011 = -5 \end{array}$ <p>(a) M = 2 = 0010 S = 7 = 0111 -S = 1001</p>	$\begin{array}{r} 0101 = 5 \\ +1110 = -2 \\ \hline 10011 = 3 \end{array}$ <p>(b) M = 5 = 0101 S = 2 = 0010 -S = 1110</p>
$\begin{array}{r} 1011 = -5 \\ +1110 = -2 \\ \hline 11001 = -7 \end{array}$ <p>(c) M = -5 = 1011 S = 2 = 0010 -S = 1110</p>	$\begin{array}{r} 0101 = 5 \\ +0010 = 2 \\ \hline 0111 = 7 \end{array}$ <p>(d) M = 5 = 0101 S = -2 = 1110 -S = 0010</p>
$\begin{array}{r} 0111 = 7 \\ +0111 = 7 \\ \hline 1110 = \text{Overflow} \end{array}$ <p>(e) M = 7 = 0111 S = -7 = 1001 -S = 0111</p>	$\begin{array}{r} 1010 = -6 \\ +1100 = -4 \\ \hline 10110 = \text{Overflow} \end{array}$ <p>(f) M = -6 = 1010 S = 4 = 0100 -S = 1100</p>



# Geometric Depiction of 2's Complement Integers



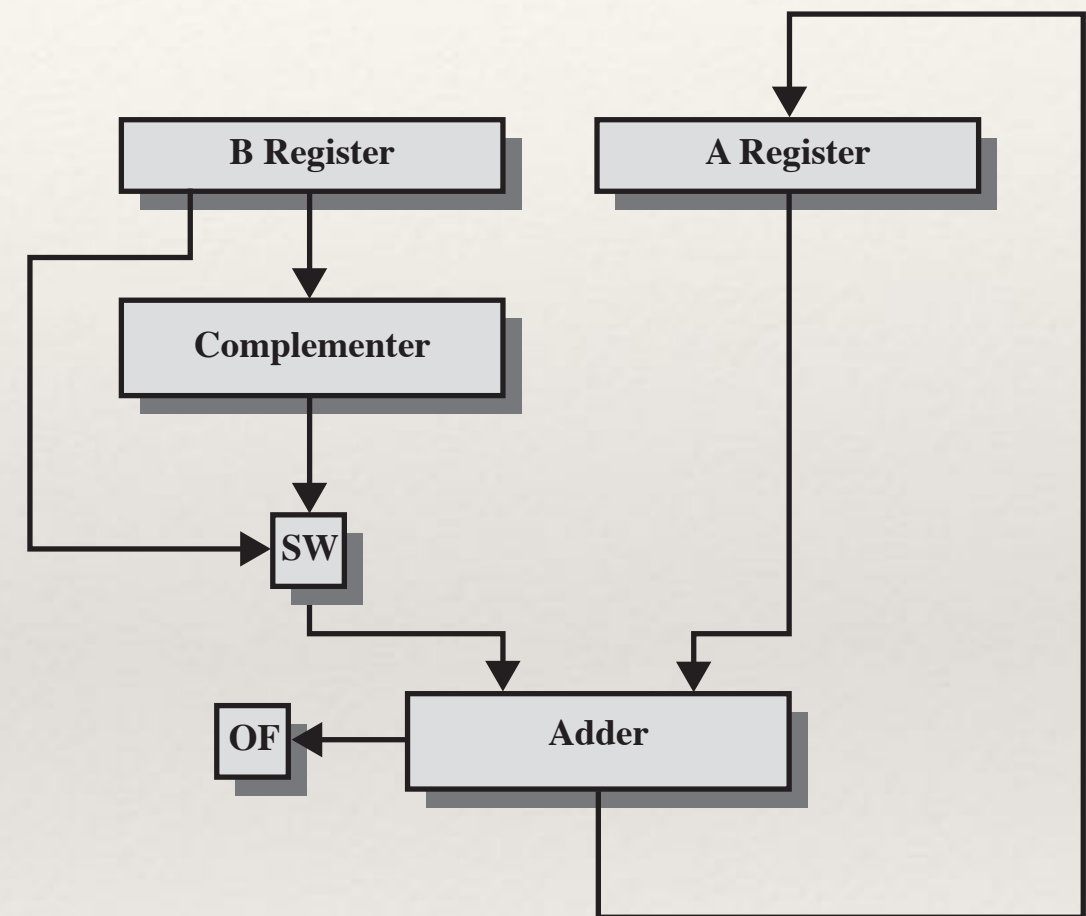
(a) 4-bit numbers



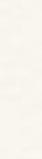
(b)  $n$ -bit numbers

# Hardware elements for Addition and Subtraction

- ❖ The central element is a binary adder
  - ❖ It is presented with two numbers for addition and produces a sum and an overflow indication.
  - ❖ It treats the two numbers as unsigned integers.
  - ❖ For addition, the two numbers are presented to the adder from two registers, designated in this case as A and B registers.
  - ❖ The result may be stored in one of these registers or in a third.
  - ❖ The overflow indication is stored in a 1-bit overflow flag (0 = no overflow; 1 = overflow).
  - ❖ For subtraction, the subtrahend (B register) is passed through a twos complementer so that its twos complement is presented to the adder.
  - ❖ Control signals are needed to control whether or not the complementer is used, depending on whether the operation is addition or subtraction.



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



# Multiplication

❖ Compared with addition and subtraction, multiplication is a complex operation, whether performed in hardware or software

1. Multiplication involves the generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
2. The partial products are easily defined. When the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
3. The total product is produced by summing the partial products. For this operation, each successive partial product is shifted one position to the left relative to the preceding partial product.
4. The multiplication of two  $n$ -bit binary integers results in a product of up to  $2n$  bits in length (e.g.,  $11 * 11 = 1001$ ).

1011	<b>Multiplicand (11)</b>
× 1101	<b>Multiplier (13)</b>
1011	} <b>Partial products</b>
0000	
1011	
1011	
10001111	<b>Product (143)</b>

## ❖ Observations

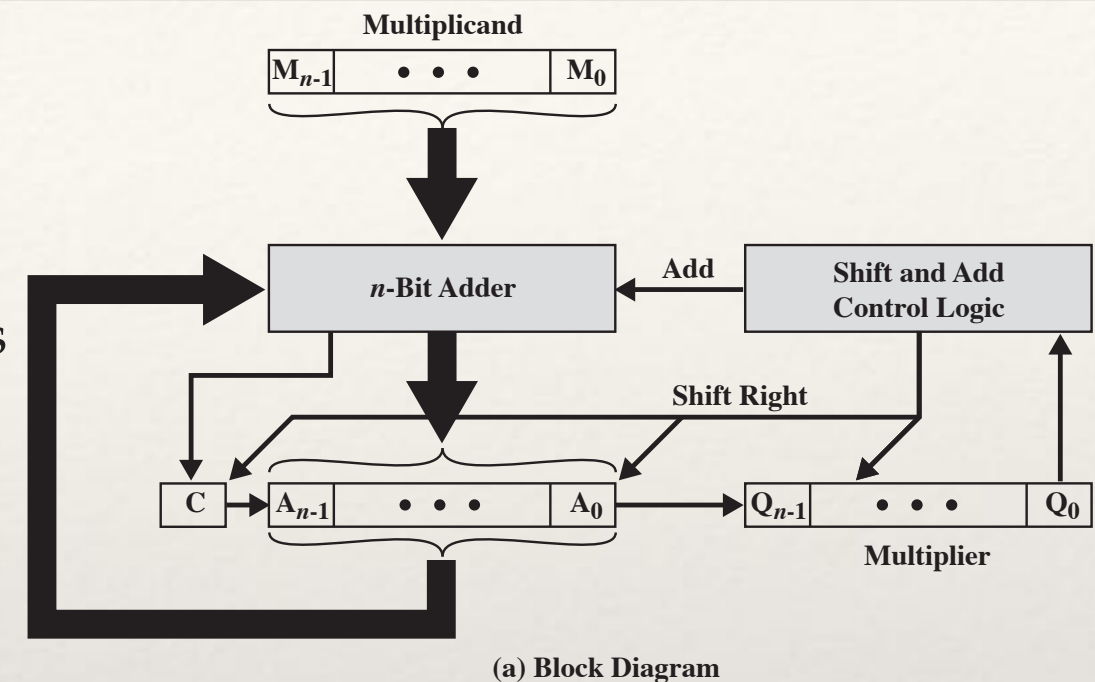
- ❖ We can perform a running addition on the partial products rather than waiting until the end.
  - ❖ This eliminates the need for storage of all the partial products
- ❖ We can save some time on the generation of partial products. For each 1 on the multiplier, an add and a shift operation are required; but for each 0, only a shift is required.





# Hardware Implementation of Unsigned Binary Multiplication

- ❖ The multiplier and multiplicand are loaded into two registers (Q and M).
- ❖ A third register, the A register, is also needed and is initially set to 0.
- ❖ There is also a 1-bit C register, initialized to 0, which holds a potential carry bit resulting from addition.

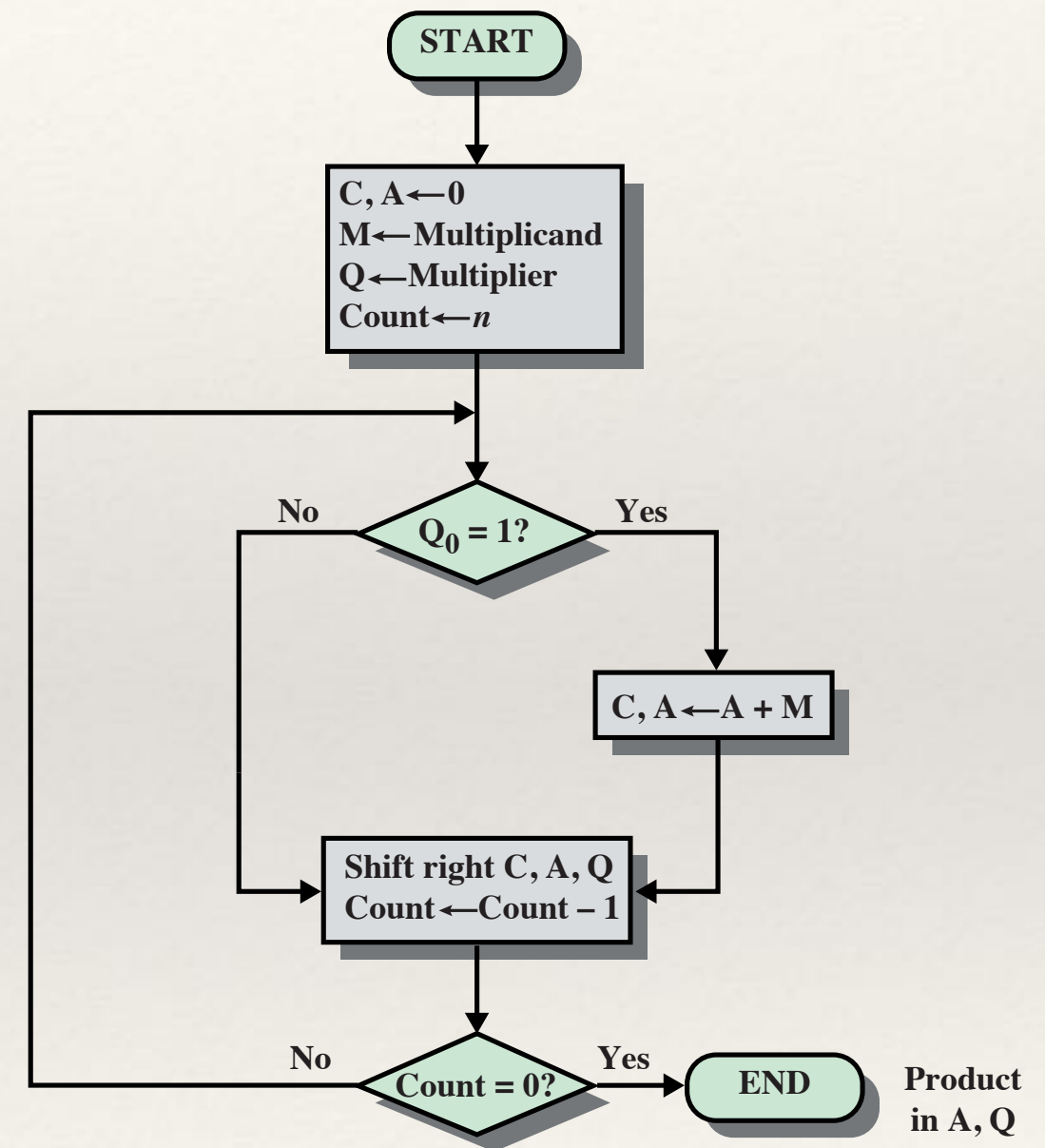


C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	} First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	} Second Cycle
0	1101	1111	1011	Add	
0	0110	1111	1011	Shift	} Third Cycle
1	0001	1111	1011	Add	
0	1000	1111	1011	Shift	} Fourth Cycle

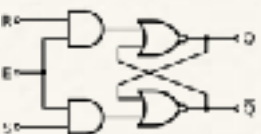
**(b) Example from Figure 9.7 (product in A, Q)**

# Unsigned Multiplier Operations

- ❖ Control logic reads the bits of the multiplier one at a time.
  - ❖ If  $Q_0$  is 1
    - ❖ Then the multiplicand is added to the A register and the result is stored in the A register, with the C bit used for overflow.
    - ❖ Then all of the bits of the C, A, and Q registers are shifted to the right one bit, so that the C bit goes into  $A_{n-1}$ ,  $A_0$  goes into  $Q_{n-1}$  and  $Q_0$  is lost.
  - ❖ If  $Q_0$  is 0
    - ❖ Then no addition is performed, just the shift.
  - ❖ This process is repeated for each bit of the original multiplier.
- ❖ The resulting  $2n$ -bit product is contained in the A and Q registers.



Product  
in A, Q



# Twos Complement Multiplication

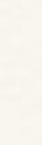
- ❖ For twos complement integers, subtraction can be performed by treating them as unsigned integers
  - ❖ But it is not the case for multiplication
- ❖ Consider the example  $1001_2 * 0011_2$ 
  - ❖ If  $1001_2$  is interpreted as the twos complement value  $-7$ , then each partial product must be a negative twos complement number of  $2n$  (8) bits

1001	(9)
×0011	(3)
00001001	$1001 \times 2^0$
00010010	$1001 \times 2^1$
00011011	(27)

(a) Unsigned integers

1001	(-7)
×0011	(3)
11111001	$(-7) \times 2^0 = (-7)$
11110010	$(-7) \times 2^1 = (-14)$
11101011	(-21)

(b) Twos complement integers



# Revisiting Twos Complement Representation

❖ What is the number represented by  $00011110_2$ ?

❖ It is equal to  $2^4 + 2^3 + 2^2 + 2^1 = 30$

❖ What is the value of  $00100000_2 - 00000010_2$ ?

❖ It is equal to  $2^5 - 2^1 = 30$

❖  $00100000_2 - 00000010_2 = 00011110_2$

❖ What is the number represented by  $01110110_2$ ?

❖ It is equal to  $2^6 + 2^5 + 2^4 + 2^2 + 2^1 = 118$

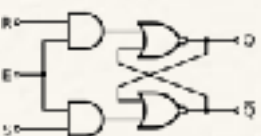
❖  $01110110_2 = 01110000_2 + 00000110_2$

❖  $01110000_2 = 10000000_2 - 00010000_2$

❖  $00000110_2 = 00001000_2 - 00000010_2$

❖ Thus

❖  $01110110_2 = 10000000_2 - 00010000_2 + 00001000_2 - 00000010_2$





# Twos Complement Multiplication

❖ Now, consider again  $1001_2 (-7) * 0011_2 (3)$

❖  $11111001_2 * 00000011_2 = 11111001_2 * (00000100_2 - 00000001_2)$

❖  $11111001_2 * 00000011_2 = 00000100_2 * 11111001_2 - 00000001_2 * 11111001_2$

❖  $11111001_2 * 00000011_2 = 11100100_2 + 00000111_2$

					1	0	0	1
x					0	0	1	1
	0	0	0	0	0	1	1	1
+	1	1	1	0	0	1		
	1	1	1	0	1	0	1	1

$\begin{array}{r} 0111 \\ \times 0011 \\ \hline 11111001 \\ 00000000 \\ 000111 \\ \hline 00010101 \end{array}$	$\begin{array}{r} 0111 \\ \times 1101 \\ \hline 11111001 \\ 0000111 \\ 111001 \\ \hline 11101011 \end{array}$
--	---

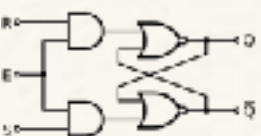
(a)  $(7) \times (3) = (21)$

(b)  $(7) \times (-3) = (-21)$

$\begin{array}{r} 1001 \\ \times 0011 \\ \hline 00000111 \\ 00000000 \\ 111001 \\ \hline 11101011 \end{array}$	$\begin{array}{r} 1001 \\ \times 1101 \\ \hline 00000111 \\ 1111001 \\ 000111 \\ \hline 00010101 \end{array}$
--	---

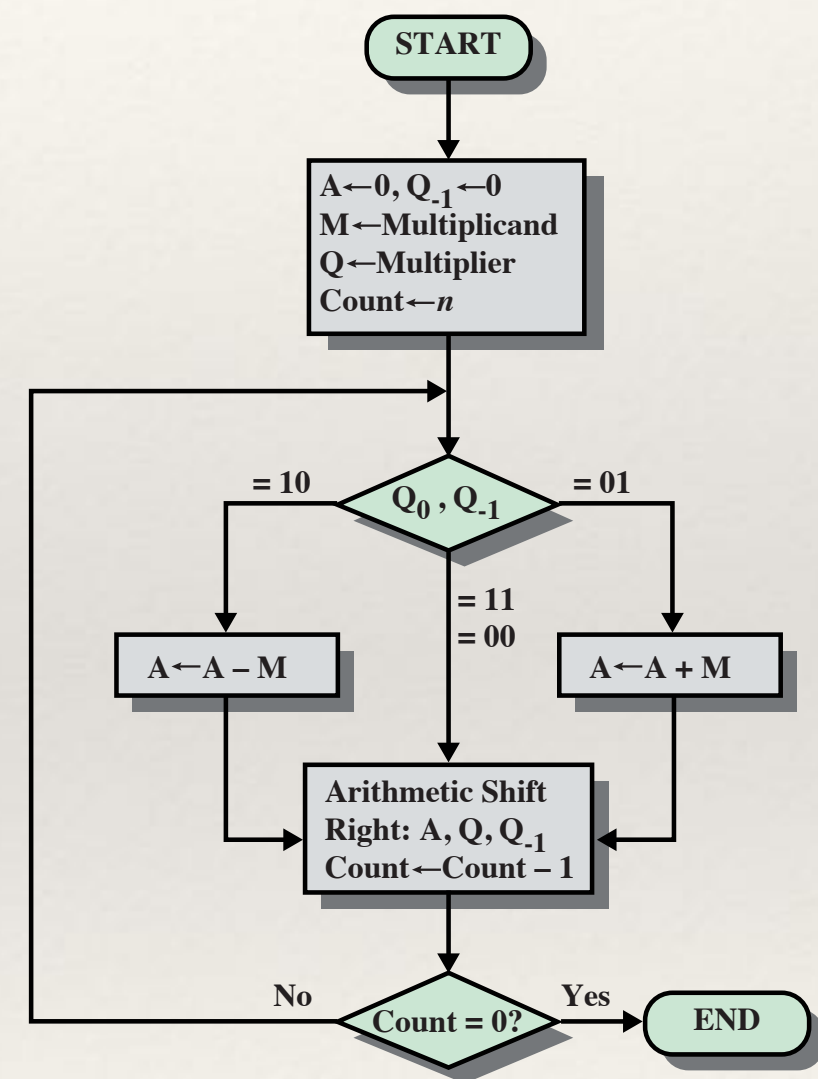
(c)  $(-7) \times (3) = (-21)$

(d)  $(-7) \times (-3) = (21)$



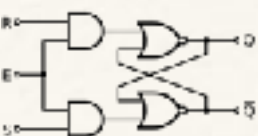
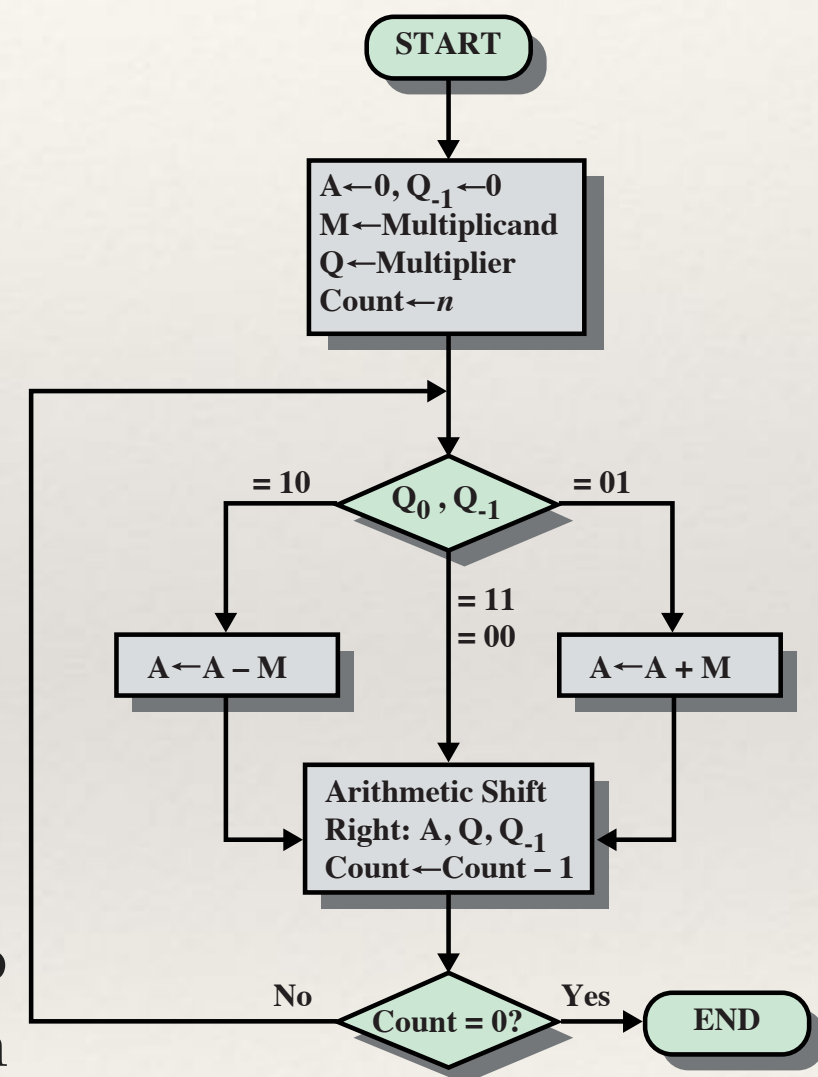
# Booth's Algorithm for Two's Complement Multiplication (1)

- ❖ As before, the multiplier and multiplicand are placed in the Q and M registers, respectively
- ❖ There is also a 1-bit register placed logically to the right of the least significant bit ( $Q_0$ ) of the Q register and designated  $Q_{-1}$  its use is explained shortly.
- ❖ The results of the multiplication will appear in the A and Q registers.
- ❖ A and  $Q_{-1}$  are initialized to 0.



# Booth's Algorithm for Two's Complement Multiplication (2)

- ❖ Now, as each bit  $Q_0$  is examined, the bit to its right  $Q_{-1}$  is also examined
  - ❖ If the two bits are the same (1-1 or 0-0), do nothing.
  - ❖ If the two bits differ, then the multiplicand is added to or subtracted from the A register, depending on whether the two bits  $Q_0 Q_{-1}$  are 0-1 or 1-0.
- ❖ Then all of the bits of the A, Q, and  $Q_{-1}$  registers are shifted to the right 1 bit.
  - ❖ Note that the leftmost bit of A, namely,  $A_{n-1}$  after it is shifted into  $A_{n-2}$ , also remains in  $A_{n-1}$ . This is required to preserve the sign of the number in A and Q. It is known as an arithmetic shift, because it preserves the sign bit.



# Example of Booth's Algorithm

A	Q	Q <sub>-1</sub>	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	$A \leftarrow A - M$ Shift	} First Cycle
1100	1001	1	0111		
1110	0100	1	0111	Shift	} Second Cycle
0101	0100	1	0111	$A \leftarrow A + M$ Shift	} Third Cycle
0010	1010	0	0111		
0001	0101	0	0111	Shift	} Fourth Cycle

**Figure 10.13 Example of Booth's Algorithm (7× 3)**

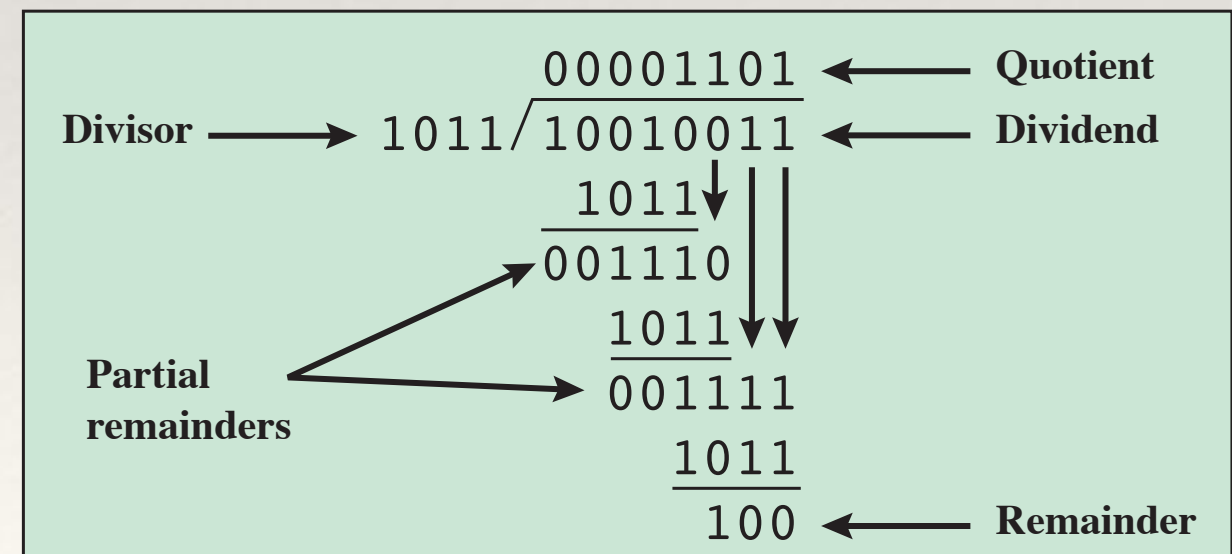




# Division

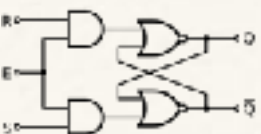
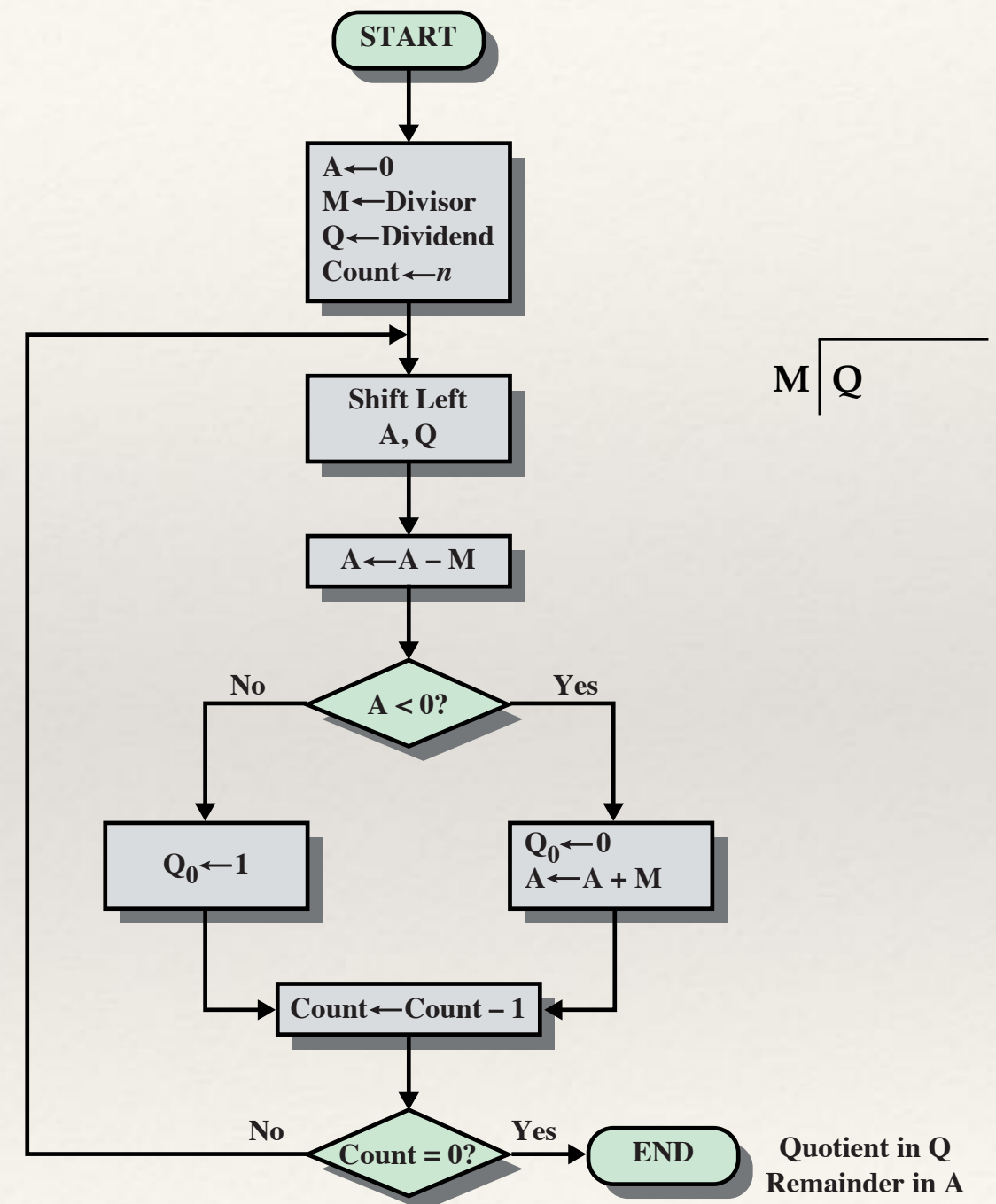
- ❖ Division is somewhat more complex than multiplication but is based on the same general principles.
  - ❖ As before, the basis for the algorithm is the paper-and-pencil approach, and the operation involves repetitive shifting and addition or subtraction.
  - ❖ First, the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor
  - ❖ Until this event occurs, 0s are placed in the quotient from left to right. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend to obtain a *partial remainder*

- ❖ From this point on, the division follows a cyclic pattern.
  - ❖ At each cycle, additional bits from the dividend are appended to the partial remainder until the result is greater than or equal to the divisor.
  - ❖ As before, the divisor is subtracted from this number to produce a new partial remainder.
  - ❖ The process continues until all the bits of the dividend are exhausted.



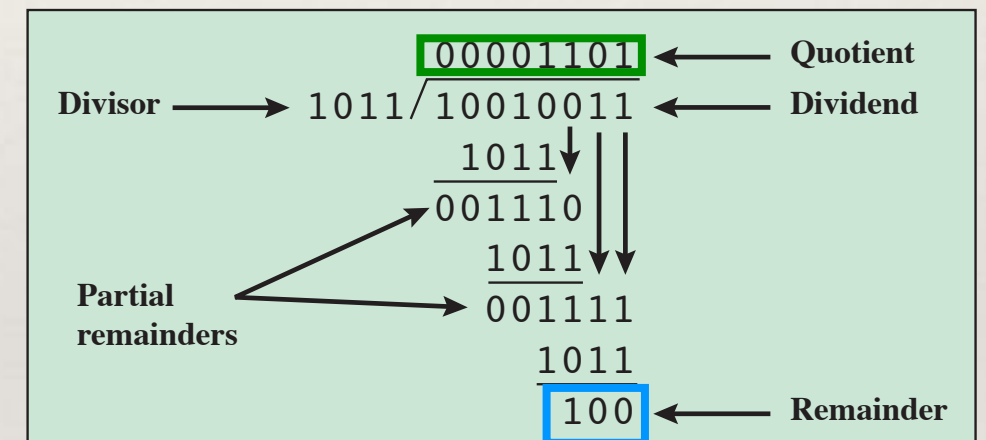
# Division Algorithm

- ❖ The divisor is placed in the M register, the dividend in the Q register.
- ❖ At each step, the A and Q registers together are shifted to the left 1 bit. M is subtracted from A to determine whether A divides the partial remainder.
  - ❖ If it does, then  $Q_0$  gets a 1bit ;
  - ❖ Otherwise,  $Q_0$  gets a 0 bit and M must be added back to A to restore the previous value.
- ❖ The count is then decremented, and the process continues for  $n$  steps.
- ❖ At the end, the quotient is in the Q register and the remainder is in the A register.



# Division Example Revisit

Count	A<M	M	A								Q							
-	-	1011	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1
8	Yes	1011	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0
8	Yes	1011	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0
7	Yes	1011	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0
7	Yes	1011	0	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0
6	Yes	1011	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0
6	Yes	1011	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0
5	Yes	1011	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0
5	Yes	1011	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0
4	No	1011	0	0	0	0	1	0	0	1	0	0	1	1	0	0	0	0
4	No	1011	0	0	0	0	0	1	1	1	0	1	1	0	0	0	0	1
3	No	1011	0	0	0	0	1	1	1	0	1	1	0	0	0	0	1	0
3	No	1011	0	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1
2	Yes	1011	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	0
2	Yes	1011	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	0
1	No	1011	0	0	0	0	1	1	1	1	0	0	0	0	1	1	0	0
1	No	1011	0	0	0	0	0	1	0	0	0	0	0	1	1	0	1	



# Handling Negative Numbers in Division

- ❖ The remainder is defined by

- ❖  $D = Q \times V + R$

- ❖  $D = \text{Integer}(D / V) \times V + R$

- ❖ Rule for the signs of Q & R

- ❖  $\text{sign}(Q) = \text{sign}(D) \times \text{sign}(V)$

- ❖  $\text{sign}(R) = \text{sign}(D)$

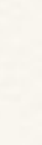
- ❖ So, to do division with negative numbers

- ❖ Convert all the numbers to positive numbers

- ❖ Apply Division Algorithm

- ❖ Assign the signs of Q and R according to the aforementioned rules

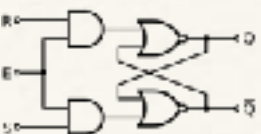
$D = 7$	$V = 3$	$\Rightarrow$	$Q = 2$	$R = 1$
$D = 7$	$V = -3$	$\Rightarrow$	$Q = -2$	$R = 1$
$D = -7$	$V = 3$	$\Rightarrow$	$Q = -2$	$R = -1$
$D = -7$	$V = -3$	$\Rightarrow$	$Q = 2$	$R = -1$





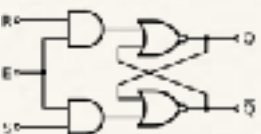
# Floating-Point Representation

- ❖ With a fixed-point notation it is possible to represent a range of positive and negative integers centered on or near 0
- ❖ By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well
- ❖ Limitations:
  - ❖ Very large numbers cannot be represented nor can very small fractions
  - ❖ The fractional part of the quotient in a division of two large numbers could be lost



# Floating-Point Representation (Cont'd)

- ❖ For decimal numbers, we get around this limitation by using scientific notation
  - ❖ 976,000,000,000,000 can be represented as  $9.76 \times 10^{14}$
- ❖ Scientific notation allows us to dynamically slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point
- ❖ This allows a range of very large and very small numbers to be represented with only a few digits
- ❖ We can employ the same principles with binary numbers
  - ❖  $\pm S \times B^{\pm E}$
- ❖ See need to store the following in the binary word
  - ❖ Sign: plus or minus
  - ❖ Significant **S**
  - ❖ Exponent **E**
  - ❖ Base **B** is implied and need not be stored



# Floating-Point Representation



$$\begin{aligned}
 1.1010001 \times 2^{10100} &= 0 \ 10010011 \ 101000100000000000000000 = 1.6328125 \times 2^{20} \\
 -1.1010001 \times 2^{10100} &= 1 \ 10010011 \ 101000100000000000000000 = -1.6328125 \times 2^{20} \\
 1.1010001 \times 2^{-10100} &= 0 \ 01101011 \ 101000100000000000000000 = 1.6328125 \times 2^{-20} \\
 -1.1010001 \times 2^{-10100} &= 1 \ 01101011 \ 101000100000000000000000 = -1.6328125 \times 2^{-20}
 \end{aligned}$$

- ❖ The leftmost bit stores the sign of the number (0 = positive, 1 = negative)
- ❖ The exponent value is stored in the next 8 bits. The representation used is known as a biased representation. A fixed value, called the bias, is subtracted from the field to get the true exponent value
  - ❖ Typically, the bias equals  $(2^{k-1} - 1)$ , where  $k$  is the number of bits in the binary exponent. In this case, the 8-bit field yields the number 0 through 255. With a bias of 127 ( $2^7 - 1$ )
  - ❖ The true exponent values are in the range -127 to +128.
- ❖ In this example, the base is assumed to be 2

# Floating Point Representation - Significand

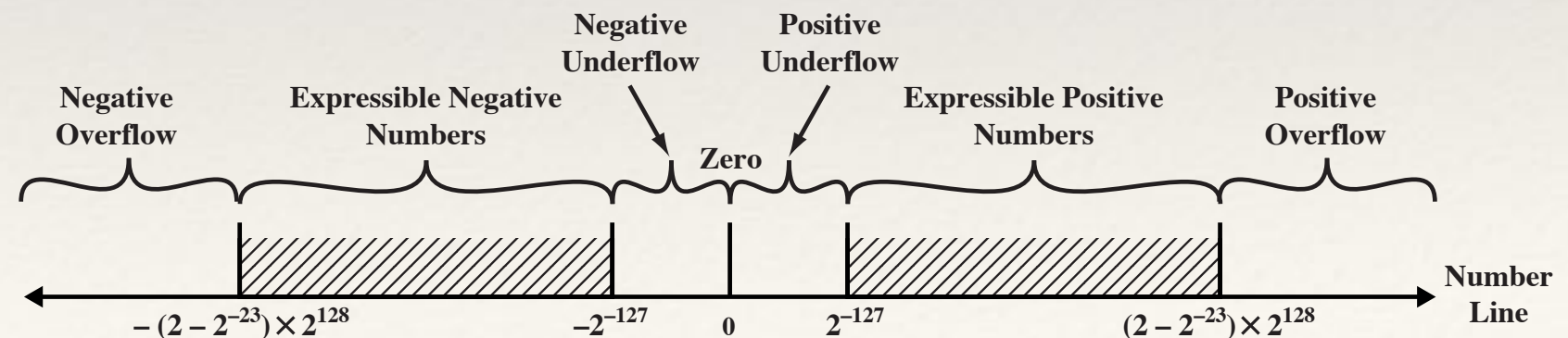
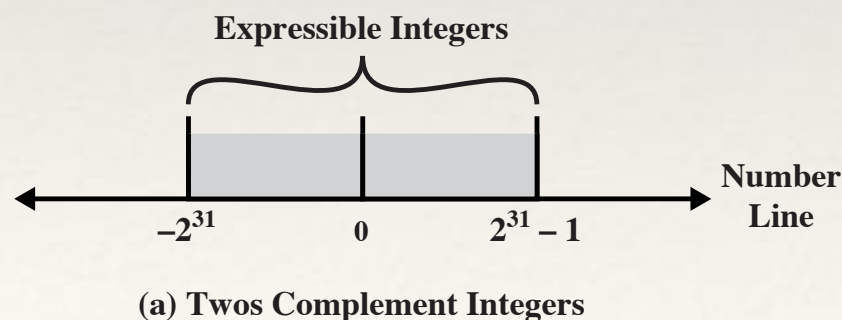
- ❖ Significant is the final portion of the word
- ❖ Any floating-point number can be expressed in many ways
  - ❖  $0110 \times 2^5$
  - ❖  $110 \times 2^2$
  - ❖  $0.0110 \times 2^6$
- ❖ It is typically required that the floating-point number be normalized
  - ❖ *Normal Number*
    - ❖ The most significant digit of the significand is nonzero
    - ❖ For base 2 representation, a normal number is therefore the one in which the most significant bit of the significand is one
      - ❖ i.e.  $\pm 1.bbb \dots b \times 2^{\pm E}$
    - ❖ Because the most significant bit is always one, it is unnecessary to store this bit
    - ❖ Thus 23-bit field is used to store a 24-bit significand





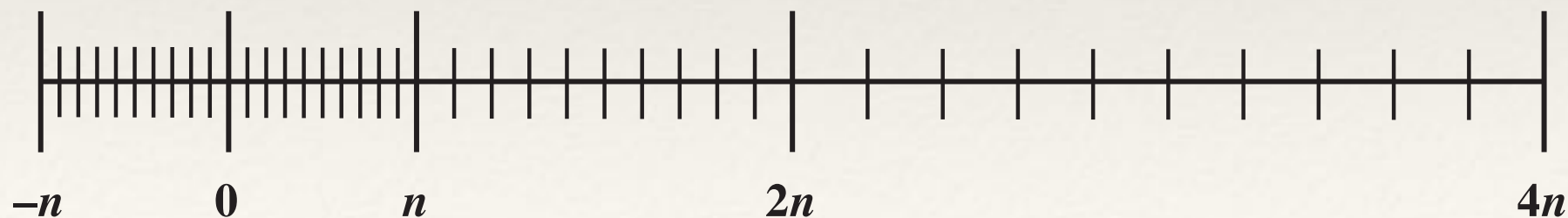
# Expressible Numbers in Typical 32-Bit Formats

- ❖ The representation as presented will not accommodate a value of 0
- ❖ Actual floating-point representations include a special bit pattern to designate zero, which shall be discussed later
- ❖ Overflow occurs when an arithmetic operation results in an absolute value greater than can be expressed with an exponent of 128
- ❖ Underflow occurs when the fractional magnitude is too small
  - ❖ Underflow is a less serious problem because the result can generally be satisfactorily approximated by 0
- ❖ Remark
  - ❖ We are not representing more individual values with floating-point notation
  - ❖ Most floating-point numbers that one would wish to represent are represented only approximately



# Density of Floating-Point Numbers

- ❖ The numbers represented in floating-point notation are not spaced evenly along the number line
- ❖ The possible values get closer together near the origin and farther apart as you move away
- ❖ There is one of the trade-offs of floating-point math
  - ❖ Many calculations produce results that are not exact and have to be rounded to the nearest value that the notation can represent.



# Trade-off between Range and Precision

- ❖ The example below shows 8 bits devoted to the exponent and 23 to the significand.
  - ❖ If we increase the number of bits in the exponent, we expand the range of expressible numbers.
  - ❖ But because only a fixed number of different values can be expressed, we have reduced the density of those numbers and therefore the precision.
- ❖ The only way to increase both range and precision is to use more bits.
- ❖ Thus, most computers offer, at least, single-precision numbers and double-precision numbers.
  - ❖ For example, a processor could support a single-precision format of 64 bits, and a double-precision format of 128 bits.
- ❖ So there is a trade-off between the number of bits in the exponent and the number of bits in the significand.
- ❖ The advantage of using a larger exponent is that a greater range can be achieved for the same number of exponent bits.
  - ❖ But remember, we have not increased the number of different values that can be represented.
  - ❖ Thus, for a fixed format, a larger exponent base gives a greater range at the expense of less precision.



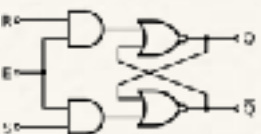
# IEEE Standard 754

Most important floating-point representation is defined

Standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs

Standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors

IEEE 754-2008 covers both binary and decimal floating-point representations



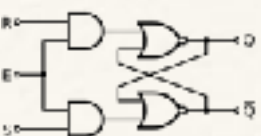


---

# IEEE 754-2008

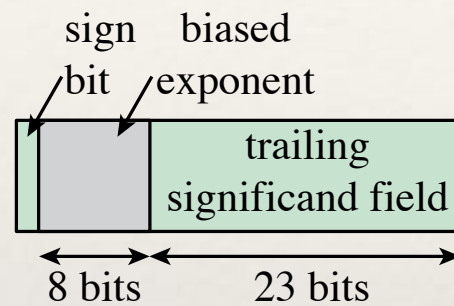
---

- ❖ Defines the following different types of floating-point formats:
- ❖ Arithmetic format
  - ❖ All the mandatory operations defined by the standard are supported by the format. The format may be used to represent floating-point operands or results for the operations described in the standard.
- ❖ Basic format
  - ❖ This format covers five floating-point representations, three binary and two decimal, whose encodings are specified by the standard, and which can be used for arithmetic. At least one of the basic formats is implemented in any conforming implementation.
- ❖ Interchange format
  - ❖ A fully specified, fixed-length binary encoding that allows data interchange between different platforms and that can be used for storage.

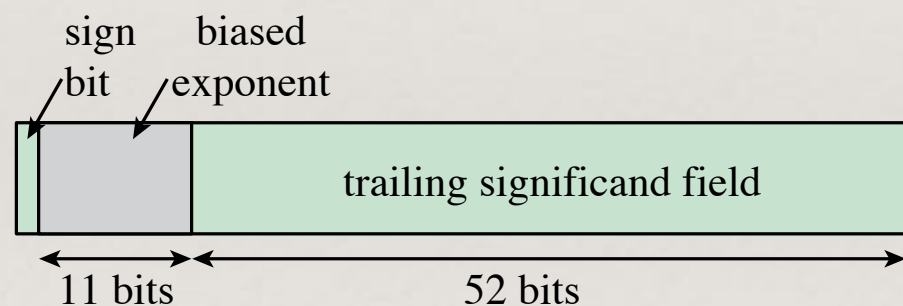


# IEEE 754 Formats

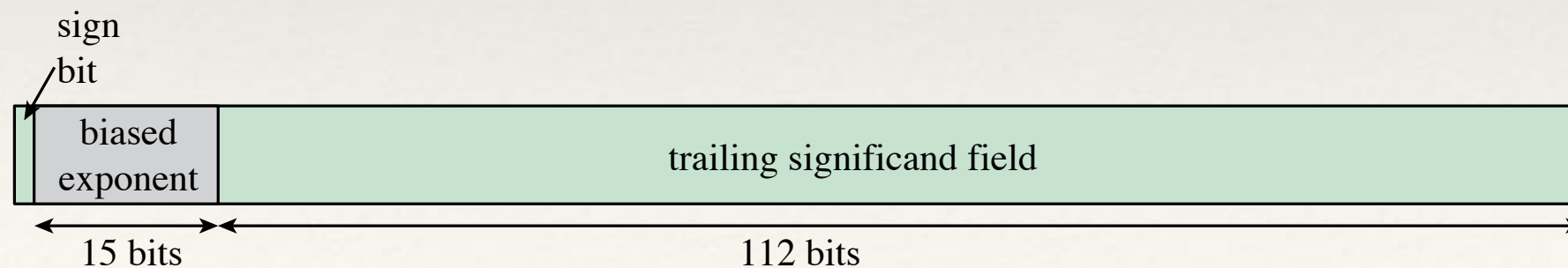
- ❖ The three basic binary formats have bit lengths of 32, 64, and 128 bits, with exponents of 8, 11, and 15 bits, respectively



**(a) binary32 format**



**(b) binary64 format**



**(c) binary128 format**

# IEEE 754 Format Parameters

Parameter	Format		
	binary32	binary64	binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	-126	-1022	-16382
Approx normal number range (base 10)	$10_{-38}, 10_{+38}$	$10_{-308}, 10_{+308}$	$10_{-4932}, 10_{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	$2_{23}$	$2_{52}$	$2_{112}$
Number of values	$1.98 \times 2_{31}$	$1.99 \times 2_{63}$	$1.99 \times 2_{128}$
Smallest positive normal number	$2_{-126}$	$2_{-1022}$	$2_{-16362}$
Largest positive normal number	$2_{128} - 2_{104}$	$2_{1024} - 2_{971}$	$2_{16384} - 2_{16271}$
Smallest subnormal magnitude	$2_{-149}$	$2_{-1074}$	$2_{-16494}$

\* not including implied bit and not including sign bit



# IEEE 754 Additional Formats

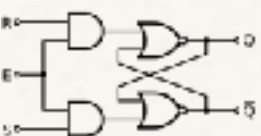
## ❖ *Extended Precision Formats*

- ❖ Provide additional bits in the exponent (extended range) and in the significand (extended precision)
- ❖ Lessens the chance of a final result that has been contaminated by excessive roundoff error
- ❖ Lessens the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format

- ❖ Affords some of the benefits of a larger basic format without incurring the time penalty usually associated with higher precision

## ❖ *Extendable Precision Format*

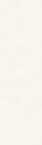
- ❖ Precision and range are defined under user control
- ❖ May be used for intermediate calculations but the standard places no constraint on format or length





# IEEE Formats and Format Types

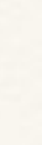
Format	Format Type		
	Arithmetic Format	Basic Format	Interchange Format
binary16			<b>X</b>
binary32	<b>X</b>	<b>X</b>	<b>X</b>
binary64	<b>X</b>	<b>X</b>	<b>X</b>
binary128	<b>X</b>	<b>X</b>	<b>X</b>
binary{k} ( $k = n \times 32$ for $n > 4$ )	<b>X</b>		<b>X</b>
decimal64	<b>X</b>	<b>X</b>	<b>X</b>
decimal128	<b>X</b>	<b>X</b>	<b>X</b>
decimal{k} ( $k = n \times 32$ for $n > 4$ )	<b>X</b>		<b>X</b>
extended precision	<b>X</b>		
extendable precision	<b>X</b>		



# Interpretation of IEEE 754 Floating-Point Numbers (1)

	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	$-0$
plus infinity	0	all 1s	0	$\infty$
Minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$ ; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$ ; first bit = 0	sNaN
positive normal nonzero	0	$0 < e < 255$	f	$2_{e-127}(1.f)$
negative normal nonzero	1	$0 < e < 255$	f	$-2_{e-127}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2_{e-126}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2_{e-126}(0.f)$

(a) binary32 format



# Interpretation of IEEE 754 Floating-Point Numbers (2)

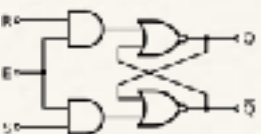
	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	$-0$
plus infinity	0	all 1s	0	$\infty$
Minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$ ; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$ ; first bit = 0	sNaN
positive normal nonzero	0	$0 < e < 2047$	f	$2_{e-1023}(1.f)$
negative normal nonzero	1	$0 < e < 2047$	f	$-2_{e-1023}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2_{e-1022}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2_{e-1022}(0.f)$

(b) binary64 format

# Interpretation of IEEE 754 Floating-Point Numbers (3)

	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	$-0$
plus infinity	0	all 1s	0	$\infty$
minus infinity	1	all 1s	0	$-\infty$
quiet NaN	0 or 1	all 1s	$\neq 0$ ; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$ ; first bit = 0	sNaN
positive normal nonzero	0	all 1s	f	$2_{e-16383}(1.f)$
negative normal nonzero	1	all 1s	f	$-2_{e-16383}(1.f)$
positive subnormal	0	0	$f \neq 0$	$2_{e-16383}(0.f)$
negative subnormal	1	0	$f \neq 0$	$-2_{e-16383}(0.f)$

(c) binary128 format





# Floating-Point Numbers and Arithmetic Operations

- ❖ For addition and subtraction, it is necessary to ensure that both operands have the same exponent value.
  - ❖ This may require shifting the radix point on one of the operands to achieve alignment.
- ❖ Multiplication and division are more straightforward.
- ❖ A floating-point operation may produce one of these conditions:
  - ❖ **Exponent overflow:** A positive exponent exceeds the maximum possible exponent value. In some systems, this may be designated as  $+\infty$  or  $-\infty$ .
  - ❖ **Exponent underflow:** A negative exponent is less than the minimum possible exponent value (e.g., -200 is less than -127). This means that the number is too small to be represented, and it may be reported as 0.
  - ❖ **Significand underflow:** In the process of aligning significands, digits may flow off the right end of the significand. As we shall discuss, some form of rounding is required.
  - ❖ **Significand overflow:** The addition of two significands of the same sign may result in a carry out of the most significant bit. This can be fixed by realignment, as we shall explain.

Floating Point Numbers
$X = X_s \times B^{X_E}$
$Y = Y_s \times B^{Y_E}$

Arithmetic Operations
$\left. \begin{aligned} X + Y &= (X_s \times B^{X_E - Y_E} + Y_s) \times B^{Y_E} \\ X - Y &= (X_s \times B^{X_E - Y_E} - Y_s) \times B^{Y_E} \end{aligned} \right\} X_E \leq Y_E$
$X \times Y = (X_s \times Y_s) \times B^{X_E + Y_E}$
$\frac{X}{Y} = \left( \frac{X_s}{Y_s} \right) \times B^{X_E - Y_E}$

$$X = 0.3 \times 10^2 = 30$$

$$Y = 0.2 \times 10^3 = 200$$

$$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$$

$$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$$

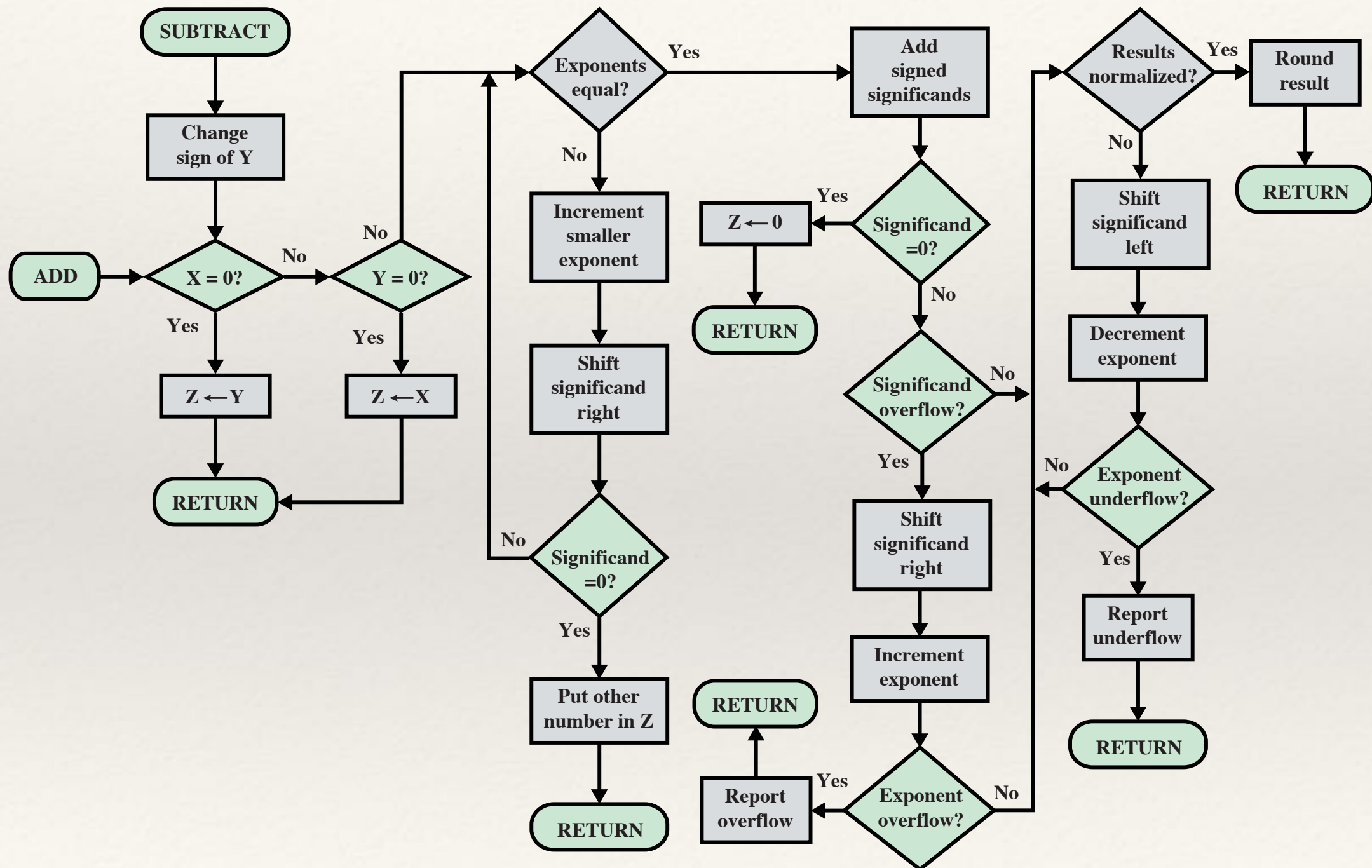
$$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$$

$$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$$

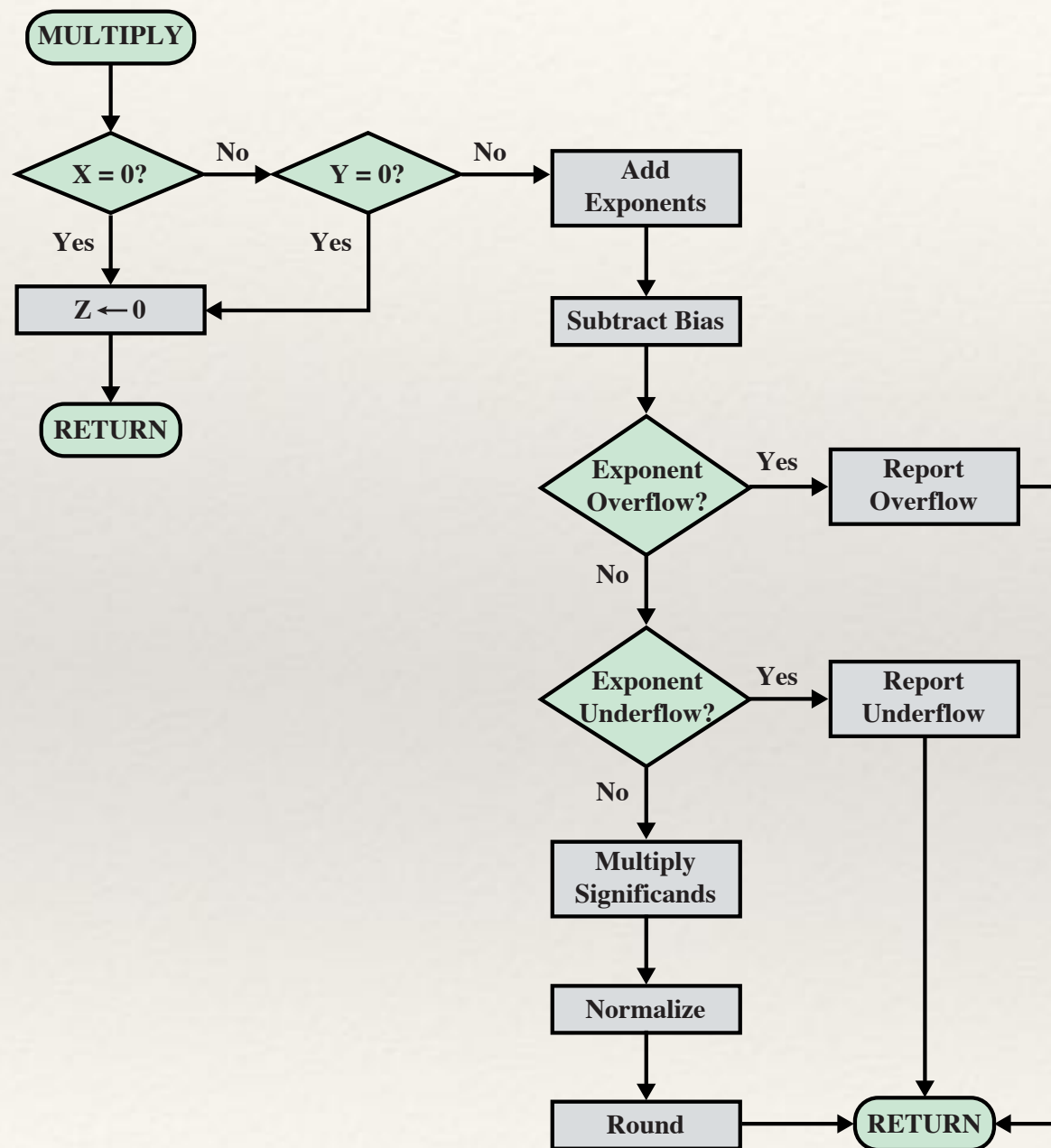


# Floating-Point Addition and Subtraction

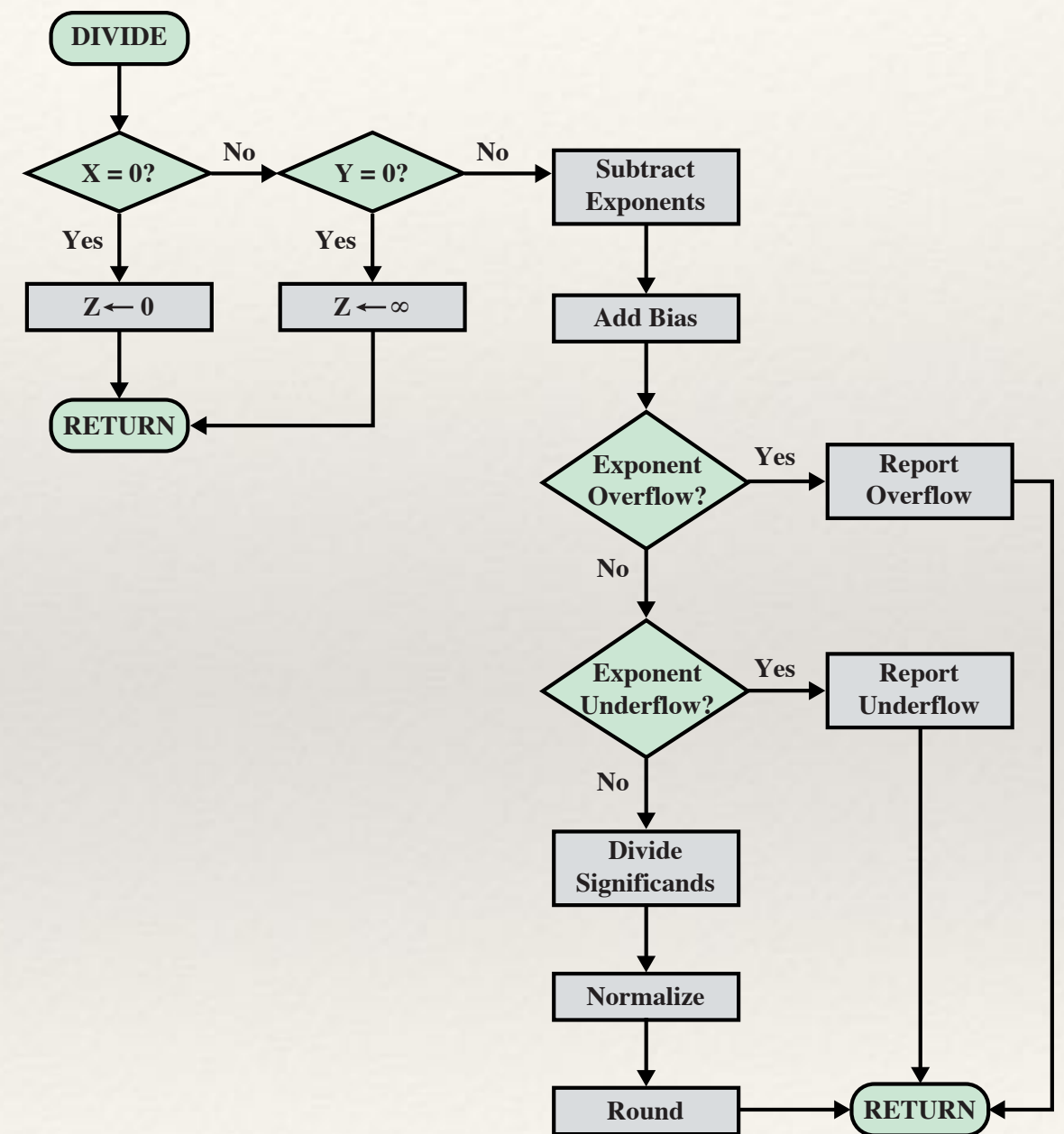
$$Z = X \pm Y$$



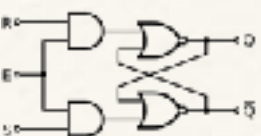
# Floating Point Multiplication and Division



$$Z = X \times Y$$



$$Z = X / Y$$



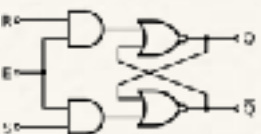
# Precision Consideration (1)

## ❖ Rounding

- ❖ The result of any operation on the significands is generally stored in a longer register. When the result is put back into the floating-point format, the extra bits must be eliminated in such a way as to produce a result that is close to the exact result
- ❖ This process is called rounding.

## ❖ IEEE standard approaches:

- ❖ Round to nearest:
  - ❖ The result is rounded to the nearest representable number.
  - ❖ If the result of a computation is exactly midway between two representable numbers, the value is rounded up if the last representable bit is currently 1 and not rounded up if it is currently 0
- ❖ Round toward  $+\infty$  :
  - ❖ The result is rounded up toward plus infinity.
- ❖ Round toward  $-\infty$ :
  - ❖ The result is rounded down toward negative infinity.
- ❖ Round toward 0:
  - ❖ The result is rounded toward zero.





# Precision Consideration (2)

- ❖ Prior to a floating-point operation, the exponent and significand of each operand are loaded into ALU registers.
  - ❖ In the case of the significand, the length of the register is almost always greater than the length of the significand plus an implied bit.
  - ❖ The register contains additional bits, called *guard bits*, which are used to pad out the right end of the significand with 0s.
- ❖ Consider numbers in the IEEE format, which has a 24-bit significand, including an implied 1 bit to the left of the binary point. Two numbers that are very close in value are  $x = 1.00 \dots 00 \times 2^1$  and  $y = 1.11 \dots 11 \times 2^0$ .
  - ❖ If the smaller number is to be subtracted from the larger, it must be shifted right 1 bit to align the exponents
  - ❖ During the process,  $y$  loses 1 bit of significance; the result is  $2^{-22}$
  - ❖ With the use of *Guard Bits*, the least significant bit is not lost due to alignment,
  - ❖ and the result is  $2^{-23}$ , a difference of a factor of 2 from the previous answer.

$$\begin{aligned}
 x &= 1.000\dots00 \times 2^1 \\
 -y &= 0.111\dots11 \times 2^1 \\
 z &= 0.000\dots01 \times 2^1 \\
 &= 1.000\dots00 \times 2^{-22}
 \end{aligned}$$

(a) Binary example, without guard bits

$$\begin{aligned}
 x &= 1.000\dots00 \ 0000 \times 2^1 \\
 -y &= 0.111\dots11 \ 1000 \times 2^1 \\
 z &= 0.000\dots00 \ 1000 \times 2^1 \\
 &= 1.000\dots00 \ 0000 \times 2^{-23}
 \end{aligned}$$

(b) Binary example, with guard bits



# Infinity

- ❖ Infinity is treated as the limiting case of real arithmetic, with the infinity values given the following interpretation:

- ❖  $-\infty < (\text{every finite number}) < +\infty$

For example:

$$5 + (+\infty) = +\infty$$

$$5 - (+\infty) = -\infty$$

$$5 + (-\infty) = -\infty$$

$$5 - (-\infty) = +\infty$$

$$5 * (+\infty) = +\infty$$

$$5 \div (+\infty) = +0$$

$$(+\infty) + (+\infty) = +\infty$$

$$(-\infty) + (-\infty) = -\infty$$

$$(-\infty) - (+\infty) = -\infty$$

$$(+\infty) - (-\infty) = +\infty$$



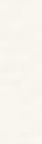
# Not a Number (NaN)

## ❖ Quiet vs Signaling NaN

quiet NaN	0 or 1	all 1s	$\neq 0$ ; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	$\neq 0$ ; first bit = 0	sNaN

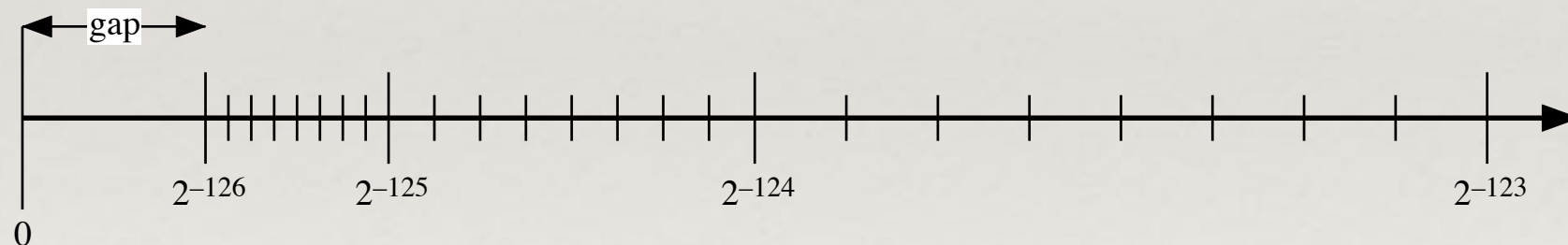
- ❖ A signaling NaN signals an invalid operation exception whenever it appears as an operand. Signaling NaNs afford values for uninitialized variables and arithmetic-like enhancements that are not the subject of the standard
- ❖ A quiet NaN propagates through almost every arithmetic operation without signaling an exception

Operation	Quiet NaN Produced by
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiply	$0 \times \infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	$x \text{ REM } 0$ or $\infty \text{ REM } y$
Square root	$\sqrt{x}$ where $x < 0$

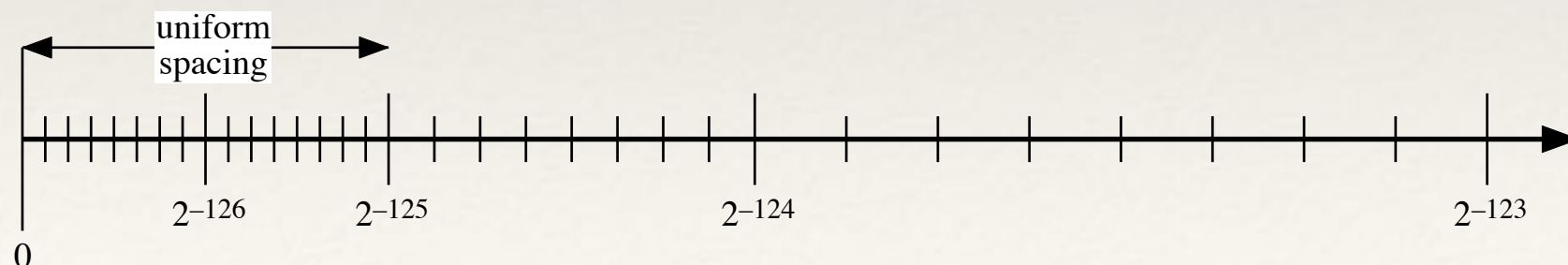


# Subnormal Numbers

- ❖ Subnormal numbers are included in IEEE 754 to handle cases of exponent underflow.
  - ❖ When the exponent of the result becomes too small (a negative exponent with too large a magnitude), the result is subnormalized by right shifting the fraction and incrementing the exponent for each shift until the exponent is within a representable range.
  - ❖ However, if only normal numbers are used, there is a gap between the smallest normal number and 0
  - ❖ In the case of the 32-bit IEEE 754 format, there are  $2^{23}$  representable numbers in each interval, and the smallest representable positive number is  $2^{-126}$ . With the addition of subnormal numbers, an additional  $2^{23} - 1$  numbers are uniformly added between 0 and  $2^{-126}$



(a) 32-bit format without subnormal numbers



(b) 32-bit format with subnormal numbers



# Chapter 10 – Summary

- ❖ ALU
- ❖ Integer representation
  - ❖ Sign-magnitude representation
  - ❖ Twos complement representation
  - ❖ Range extension
  - ❖ Fixed-point representation
- ❖ Floating-point representation
  - ❖ Principles
  - ❖ IEEE standard for binary floating-point representation
- ❖ Integer arithmetic
  - ❖ Negation
  - ❖ Addition and subtraction
  - ❖ Multiplication
  - ❖ Division
- ❖ Floating-point arithmetic
  - ❖ Addition and subtraction
  - ❖ Multiplication and division
  - ❖ Precision consideration
  - ❖ IEEE standard for binary floating-point arithmetic

