

**CS 110 (CS)**

**Intro to Computer Architecture,  
Hofstra University**

**Syllabus**

# Objectives

- Booth Algorithm and Division
- Understand the fundamental concepts of floating-point representation.
- Understand the concepts of error detecting and correcting codes

## Multiplication in unsigned binary representation:

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

**Figure 10.7** Multiplication of Unsigned Binary Integers

### Critical observations:

- 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.
- **The partial products:** when the multiplier bit is 0, the partial product is 0. When the multiplier is 1, the partial product is the multiplicand.
- **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.
- The multiplication of two  $n$ -bit binary integers results in a product of up to  $2n$  bits in length (e.g.,  $11 * 11 = 1001$ ).
- All the partial products need not be stored, eliminates the requirement to have so many registers.
- For each **1 on the multiplier**, an add and a shift operation are required; but for each 0, only a shift is required.

## Multiplication in unsigned binary numbers:

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

**Figure 10.7** Multiplication of Unsigned Binary Integers

1011	
× 1101	
00001011	$1011 \times 1 \times 2^0$
00000000	$1011 \times 0 \times 2^1$
00101100	$1011 \times 1 \times 2^2$
01011000	$1011 \times 1 \times 2^3$
10001111	

**Figure 10.10** Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

## Multiplication in unsigned binary numbers:

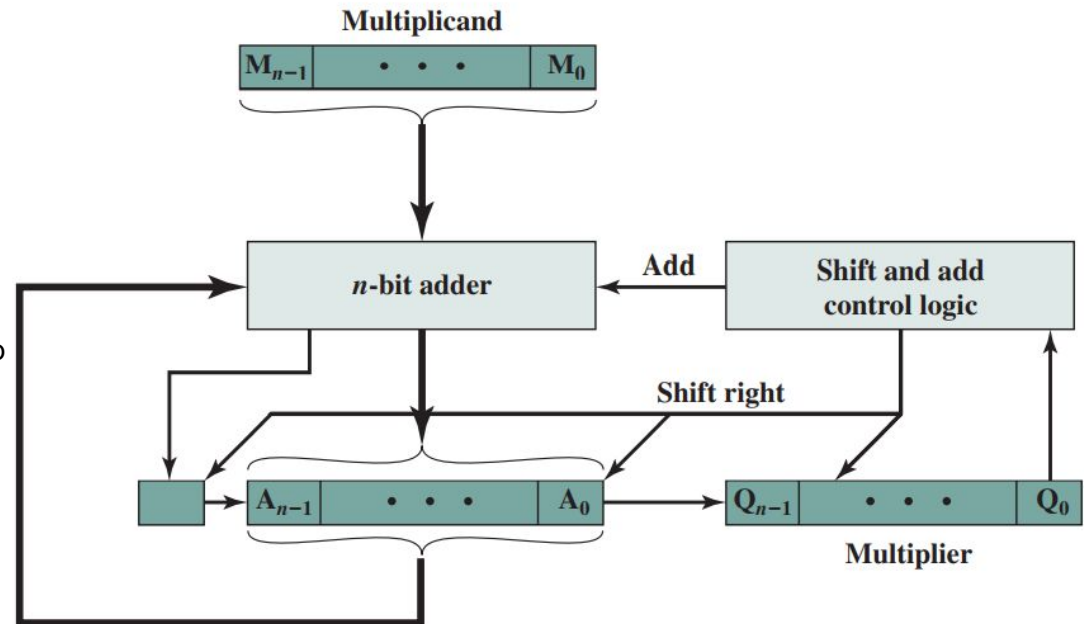
Try multiplying the following two numbers in binary:

$$\begin{array}{r} 14 \\ \times 13 \\ \hline 182 \\ \hline \end{array}$$

## Hardware for multiplication of unsigned binary numbers:

### Setup:

- 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.



(a) Block diagram

### Explanation:

1. **Control logic reads the bits of the multiplier one at a time.**
2. 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.
3. If  $Q_0$  is 0,
  - then no addition is performed, just the shift.
4. Step 2-4 are repeated for each bit of the original multiplier.
5. The resulting  $2n$ -bit product is contained in the A and Q registers.

**How it works?** See the flow chart on the next page and verify!

## Flowchart for multiplication of unsigned binary numbers:

### Explanation:

1. Control logic reads the bits of the multiplier one at a time.
2. 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.
3. If  $Q_0$  is 0,
  - then no addition is performed, just the shift.
4. Step 2-4 are repeated for each bit of the original multiplier.
5. The resulting  $2n$ -bit product is contained in the A and Q registers.

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

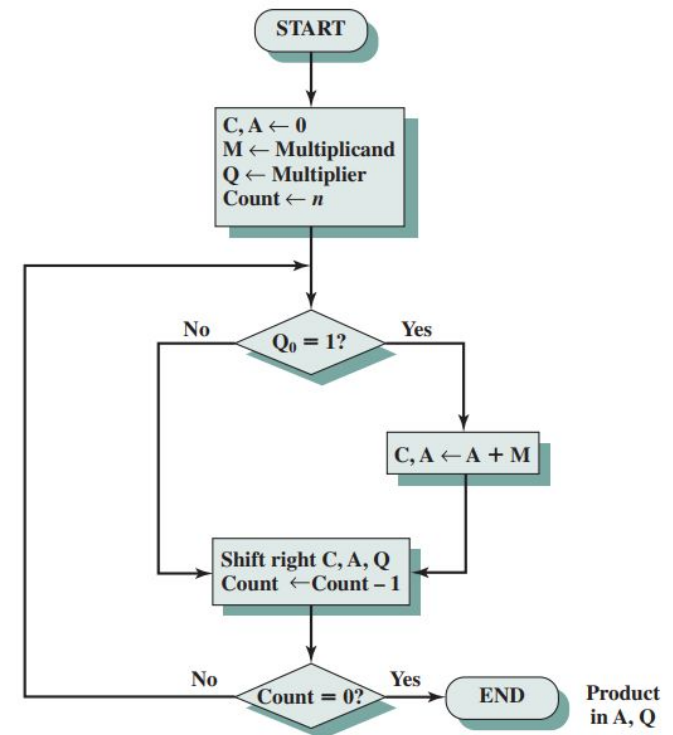


Figure 10.9 Flowchart for Unsigned Binary Multiplication

Fill in the values and actions, why did we do four cycle?

## Multiplication of signed binary numbers

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

**Figure 10.7** Multiplication of Unsigned Binary Integers

1011	
× 1101	
00001011	$1011 \times 1 \times 2^0$
00000000	$1011 \times 0 \times 2^1$
00101100	$1011 \times 1 \times 2^2$
01011000	$1011 \times 1 \times 2^3$
10001111	

**Figure 10.10** Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

**Bummer! The above process won't work for multiplication of signed numbers:**

- We multiplied 11 (1011) by 13 (1101) to get 143 (10001111).
- If we interpret these as two's complement numbers, we have - 5(1011) times - 3 (1101) equals - 113 (10001111).
- This example demonstrates that **straightforward multiplication will not work** if both the multiplicand and multiplier are negative.
- In fact, it will not work if **either the multiplicand or the multiplier is negative**. Can you verify that?



# Multiplication of signed binary numbers

## Why?

- The problem is that **each contribution of the negative multiplicand as a partial product must be a negative number** on a  $2n$ -bit field; the sign bits of the partial products must line up.
- This is demonstrated in Figure 10.11, which shows that multiplication of 1001 by 0011. If these are treated as unsigned integers, the multiplication of  $9 * 3 = 27$  proceeds simply.
- However, if 1001 is interpreted as the two's complement value -7, then each partial product must be a negative i.e. two's complement number of  $2n$  (8) bits, as shown in Figure 10.11b.
- Note that this is accomplished by padding out each partial product to the left with binary 1s**

1011	
× 1101	
00001011	$1011 \times 1 \times 2^0$
00000000	$1011 \times 0 \times 2^1$
00101100	$1011 \times 1 \times 2^2$
01011000	$1011 \times 1 \times 2^3$
10001111	

**Figure 10.10** Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

1001 (9)		1001 (-7)	
× 0011 (3)		× 0011 (3)	
00001001	$1001 \times 2^0$	11111001	$(-7) \times 2^0 = (-7)$
00010010	$1001 \times 2^1$	11110010	$(-7) \times 2^1 = (-14)$
00011011	(27)	11101011	(-21)

(a) Unsigned integers

(b) Twos complement integers

**Figure 10.11** Comparison of Multiplication of Unsigned and Twos Complement Integers

## Multiplication of signed binary numbers

- If **the multiplier is negative**, straightforward multiplication will not work.
- **A typical approach:**
  - One would be to **convert both multiplier and multiplicand to positive numbers, perform the multiplication**, and then take the two's complement of the result if and only if the sign of the two original numbers differed.
- Implementers have preferred to use techniques **that do not require this final transformation step**.

# Booth algorithm: motivation

- $978 * 1000001 = 978 * (1000000 + 1) = 978 * 1000000 + 978 * 1$
- $978 * 999999 = 978 * (1000000 - 1) = 978 * 1000000 - 978 * 1$ 
  - **Alternatively**,  $978 * 900000 + 978 * 90000 + 978 * 9000 + 978 * 900 + 978 * 90 + 978 * 9$

$$\begin{aligned} M \times (00011110) &= M \times (2^4 + 2^3 + 2^2 + 2^1) \\ &= M \times (16 + 8 + 4 + 2) \\ &= M \times 30 \end{aligned}$$

The number of such operations can be reduced to two if we observe that

$$2^n + 2^{n-1} + \dots + 2^{n-K} = 2^{n+1} - 2^{n-K}$$

$$\begin{aligned} M \times (00011110) &= M \times (2^5 - 2^1) \\ &= M \times (32 - 2) \\ &= M \times 30 \end{aligned}$$

So the product can be generated by one addition and one subtraction of the multiplicand. This scheme extends to any number of blocks of 1s in a multiplier, including the case in which a single 1 is treated as a block.

$$\begin{aligned} M \times (01111010) &= M \times (2^6 + 2^5 + 2^4 + 2^3 + 2^1) \\ &= M \times (2^7 - 2^3 + 2^2 - 2^1) \end{aligned}$$

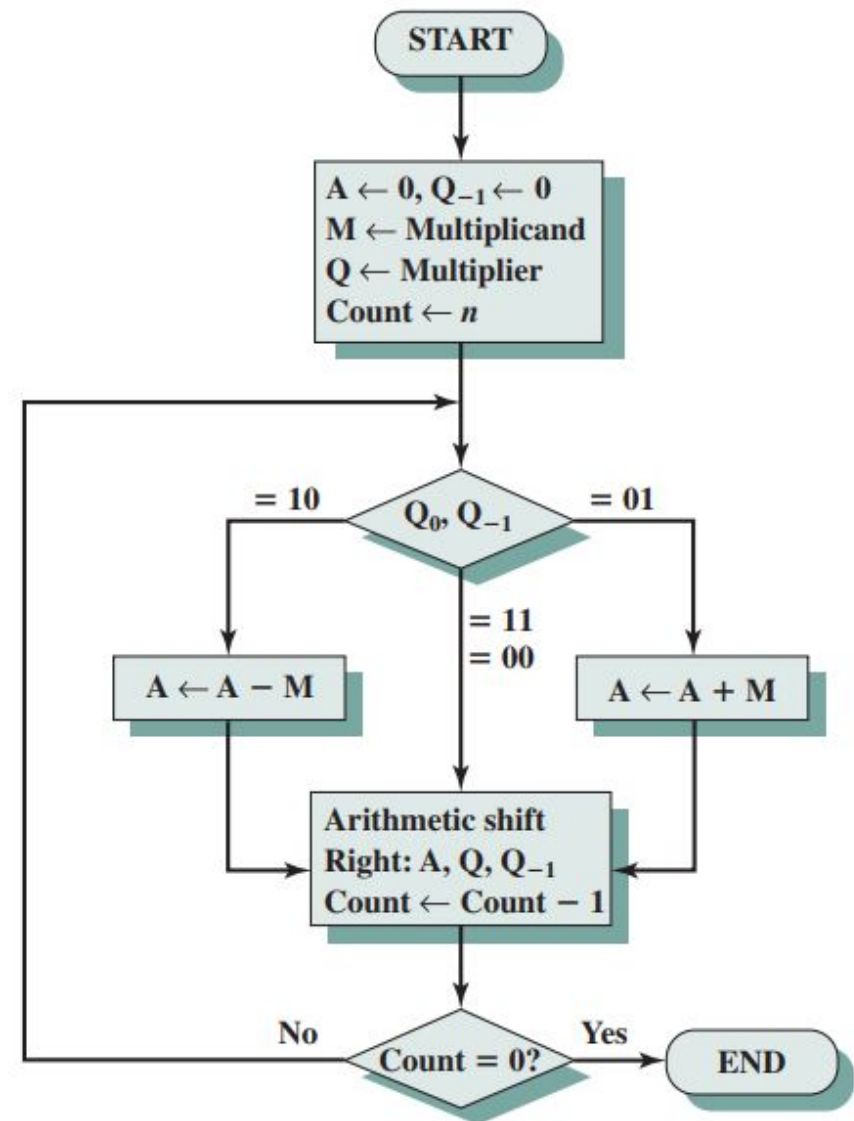
Booth's algorithm conforms to this scheme by performing a subtraction when the first 1 of the block is encountered (1-0) and an addition when the end of the block is encountered (0-1).

# Booth algorithm implementation

A	Q	Q <sub>-1</sub>	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	
0010	1010	0	0111	Shift	} Third cycle
0001	0101	0	0111	Shift	
0001	0101	0	0111	Shift	} Fourth cycle

**Figure 10.13** Example of Booth's Algorithm ( $7 \times 3$ )

- The right shift is such that the leftmost bit of A, namely  $A_{n-1}$ , not only is shifted into  $A_{n-2}$ , but 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**.
- The result goes into AQ and any extra bit beyond  $2n$  bits is discarded



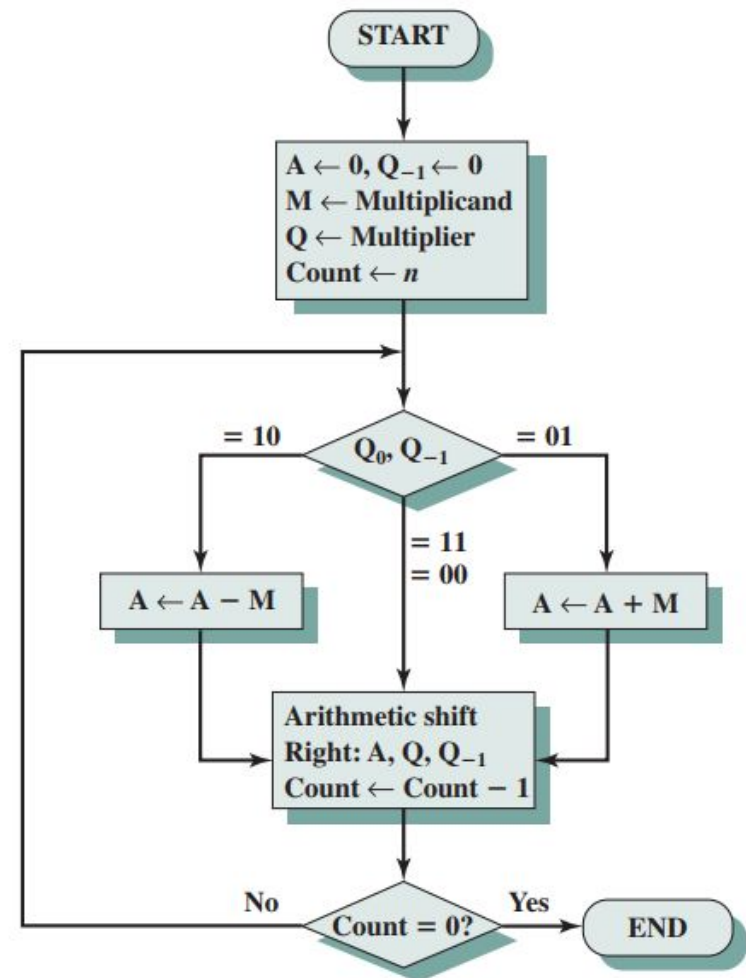
**Figure 10.12** Booth's Algorithm for Two's Complement Multiplication

# Booth algorithm implementation

Run (or trace) the flowchart for  $13 \times 11$

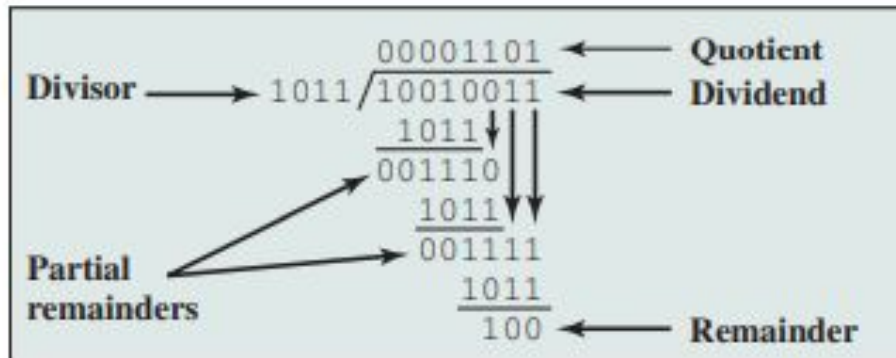
A	Q	Q <sub>-1</sub>	M	
0000	0011	0	0111	Initial values
1001	0011	0	0111	A ← A - M } First cycle
1100	1001	1	0111	
1110	0100	1	0111	Shift } Second cycle
0101	0100	1	0111	A ← A + M } Third cycle
0010	1010	0	0111	
0001	0101	0	0111	Shift } Fourth cycle

**Figure 10.13** Example of Booth's Algorithm ( $7 \times 3$ )



**Figure 10.12** Booth's Algorithm for Two's Complement Multiplication

# Division



**Figure 10.15** Example of Division of Unsigned Binary Integers

1. 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**; this is referred to as the divisor being able to divide the number.
2. Until this event occurs, 0s are placed in the quotient from left to right.
3. When the event occurs, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend.
3. The result is referred to as a **partial remainder**.

5. 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
6. 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

- The divisor is placed in the M register, the dividend in the Q register
- At each step, 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. (A result that requires a borrow out of the most significant bit is a negative result.)
- If it does, then Q0 gets a 1 bit.
- Otherwise, Q0 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.
- This process can, with some difficulty, be extended to negative numbers.

You don't subtract again because you have done so already

subtract to check if it is even possible to divide

You need to restore if it was not possible to divide

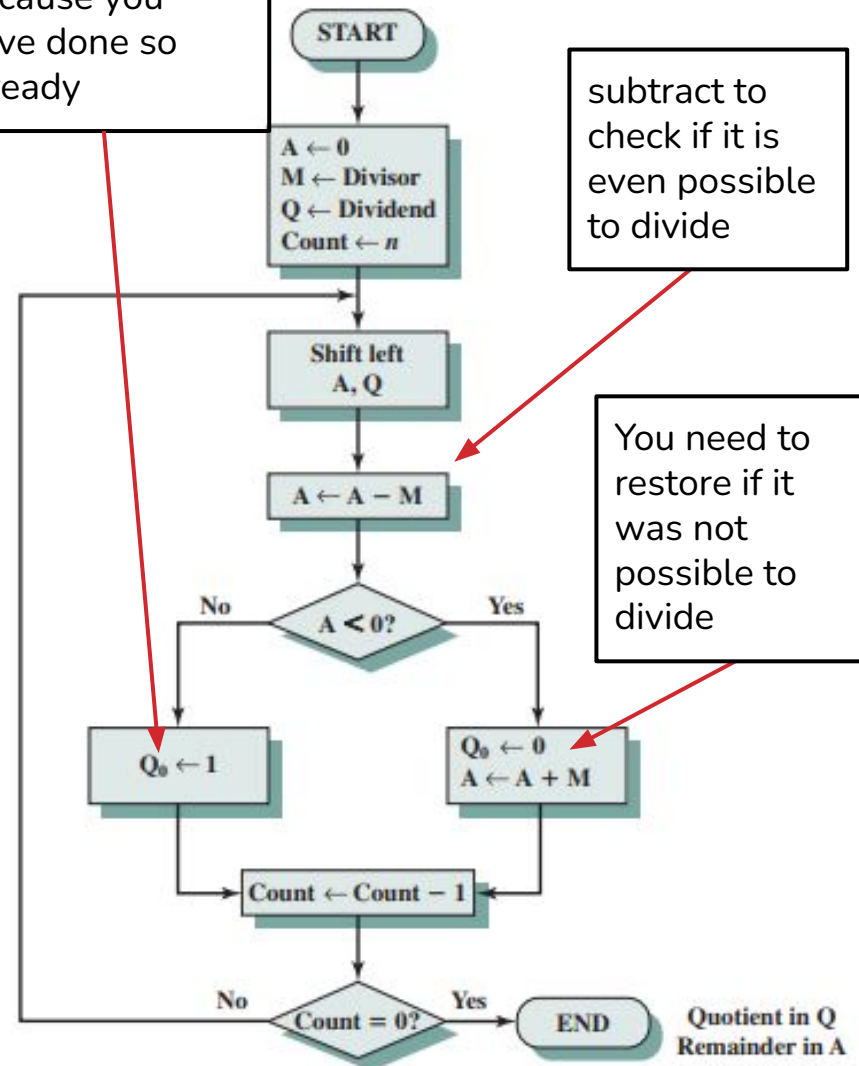
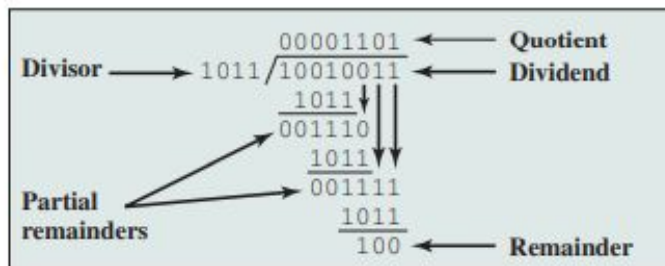


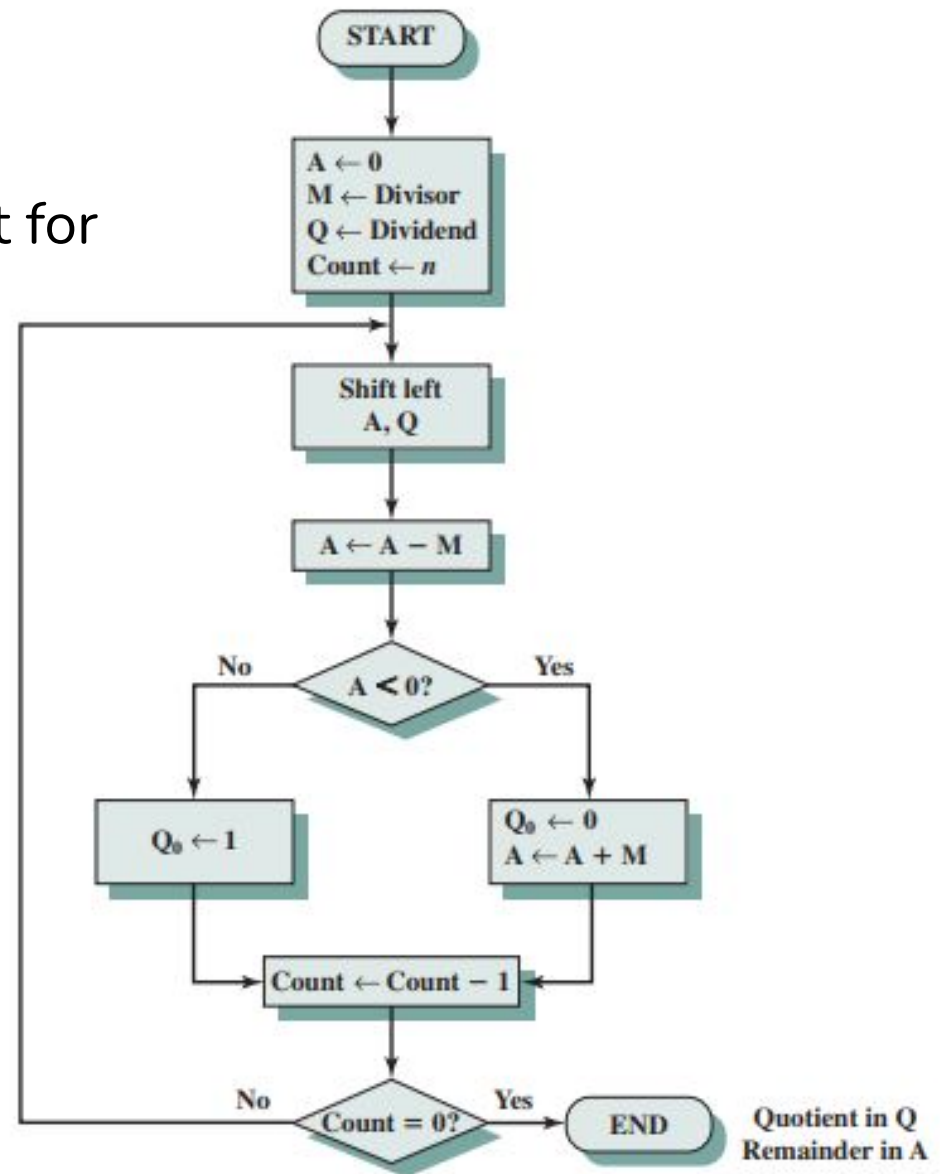
Figure 10.16 Flowchart for Unsigned Binary Division

## Division

- Can we run this flowchart for 7/3?



**Figure 10.15** Example of Division of Unsigned Binary Integers



**Figure 10.16** Flowchart for Unsigned Binary Division



## Division | 2's complement

To deal with negative numbers, we recognize that the remainder is defined by

$$D = Q \times V + R$$

That is, the remainder is the value of  $R$  needed for the preceding equation to be valid. Consider the following examples of integer division with all possible combinations of signs of  $D$  and  $V$ :

$D = 7$	$V = 3$	$\Rightarrow$	$Q = 2$	$R = 1$	What do we observe?
$D = 7$	$V = -3$	$\Rightarrow$	$Q = -2$	$R = 1$	
$D = -7$	$V = 3$	$\Rightarrow$	$Q = -2$	$R = -1$	How can we decide upon the size of $Q$ and $R$ from the signs of $D$ and $V$ ?
$D = -7$	$V = -3$	$\Rightarrow$	$Q = 2$	$R = -1$	

- We see that the **magnitudes of  $Q$  and  $R$  are unaffected** by the input signs.
- The signs of  $Q$  and  $R$  are easily derivable from the signs of  $D$  and  $V$ .
- Specifically,  $\text{sign}(R) = \text{sign}(D)$  and
- $\text{sign}(Q) = \text{sign}(D) * \text{sign}(V)$ .
- Hence, one way to do two's complement division is **to convert the operands into unsigned values** and, **at the end, to account for the signs by complementation where needed.**

# Division | 2's complement

## Algorithm and worked out example

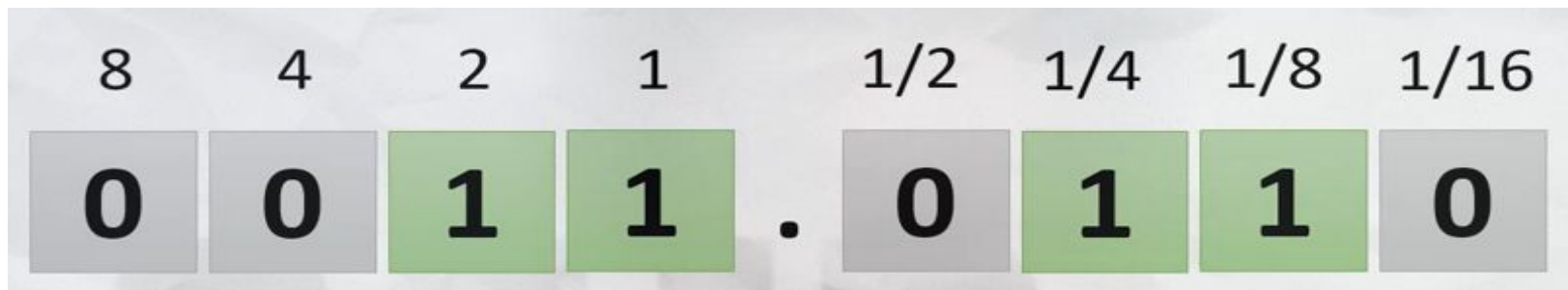
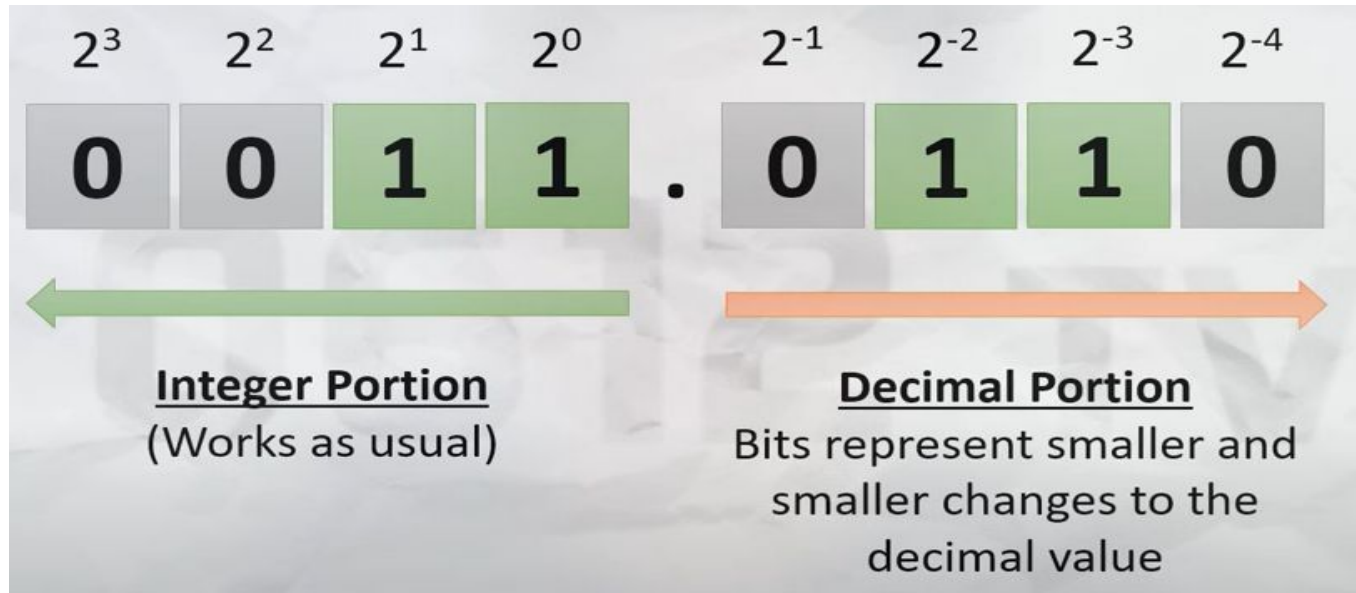
The algorithm assumes that the divisor  $V$  and the dividend  $D$  are positive and that  $|V| < |D|$ . If  $|V| = |D|$ , then the quotient  $Q = 1$  and the remainder  $R = 0$ . If  $|V| > |D|$ , then  $Q = 0$  and  $R = D$ . The algorithm can be summarized as follows:

1. Load the two's complement of the divisor into the M register; that is, the M register contains the negative of the divisor. Load the dividend into the A, Q registers. The dividend must be expressed as a  $2n$ -bit positive number. Thus, for example, the 4-bit 0111 becomes 00000111.
2. Shift A, Q left 1 bit position.
3. Perform  $A \leftarrow A - M$ . This operation subtracts the divisor from the contents of A.
4.
  - a. If the result is nonnegative (most significant bit of A = 0), then set  $Q_0 \leftarrow 1$ .
  - b. If the result is negative (most significant bit of A = 1), then set  $Q_0 \leftarrow 0$ . and restore the previous value of A.
5. Repeat steps 2 through 4 as many times as there are bit positions in Q.
6. The remainder is in A and the quotient is in Q.

A	Q	
0000	0111	Initial value
0000 <u>1101</u> 1101	1110	Shift Use two's complement of 0011 for subtraction
0000	1110	Subtract Restore, set $Q_0 = 0$
0001 <u>1101</u> 1110	1100	Shift
0001	1100	Subtract Restore, set $Q_0 = 0$
0011 <u>1101</u> 0000	1000	Shift
0000	1001	Subtract, set $Q_0 = 1$
0001 <u>1101</u> 1110	0010	Shift
0001	0010	Subtract Restore, set $Q_0 = 0$

**Figure 10.17** Example of Restoring Two's Complement Division (7/3)

## Fixed (decimal) point representation for fractional numbers



## Limitations of fixed point

**Low capacity:** short range and low precision

Range for 8 bits:  $-2^7$  to  $2^7 - 1$

- Smallest Value:

0 0 0 0 . 0 0 0 0 = 0

- Largest Value:

1 1 1 1 . 1 1 1 1 = 15.9375

# Limitations of fixed point

Low capacity: Short range and low precision

Bit String	Value		Bit String	Value
.0000	.0		.1000	.5
.0001	.0625		.1001	.5625
.0010	.125		.1010	.625
.0011	.1875		.1011	.6875
.0100	.25		.1100	.75
.0101	.3125		.1101	.8125
.0110	.375		.1110	.875
.0111	.4375		.1111	.9375

You can't represent numbers that need higher resolution than 0.06

## Limitations of fixed point

Low capacity: Short range and low precision

### Numbers that don't work (with this scheme)

$2^3$	$2^2$	$2^1$	$2^0$		$2^{-1}$	$2^{-2}$	$2^{-3}$	$2^{-4}$
0	0	0	0	.	0	0	0	0

- 16 – Overflows the integer portion
- $2^{-5} = 0.03125$  – Overflows the decimal portion
- 16.03125 – Overflows on both ends!
- $0.2 = \frac{1}{5}$  - Cannot be represented by any combination of powers of 2 (this problem runs deep – More later)

None of the combinations of bits can exactly represent  $\frac{1}{5}$ , we will approximate and the error is known as truncation (or round off error)

# How to get rid of these limitations?

draw some ideas from decimal number system

Thus, 976,000,000,000,000 can be represented as

$$9.76 * 10^{14}$$

Similarly, 0.00000000000000976 can be represented as

$$9.76 * 10^{-14}$$

**Terminologies:**

in  $9.76 * 10^{-14}$

9.76 is Mantissa or significand

10 is base or radix

-14 is exponent

- Range is limited by the ???
- Precision is limited by the ????

Range is limited by the **exponent**

Precision is limited by the **mantissa**



# floating point representation | binary numbers

This same approach can be taken with binary numbers. We can represent a number in the form

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

This number can be stored in a binary word with three fields:

- Sign: plus or minus
- Significand S
- Exponent E

**We have one bit reserved for the sign of the number, what about the sign of the exponent?**



(a) Format

$$\text{number} = \text{sign} (1 + \text{significand}) * 2^{(\text{exponent} - 127)}$$



# Normalization

## Normalization in scientific representation of numbers:

- A number in scientific notation with no leading 0s is called a Normalised number, e.g.,  $1.0 \times 10^{-8}$
- The normalization simplifies the exchange of data that includes floating-point numbers
- It also simplifies the arithmetic algorithms to know that the numbers will always be in this form
- It increases the accuracy (precision) of the numbers that can be stored in a word, since each unnecessary leading 0 is replaced by another significant digit to the right of the decimal point
- Avoid (duplicate) representations of the same quantity

# Normalization

Normalization in scientific representation of numbers:

- Avoids (duplicate) representations of the same quantity:
  - If we use **0** on the left/right of the decimal point, we can represent 3.14, as:
    - $0.314 \times 10^1$
    - $0.0314 \times 10^2$
    - $0.00314 \times 10^3$  and so on
  - What if we fix the digit before or after the decimal point?
    - **FD.Mantissa** e.g.,  $3.14 \times 10^0$
    - **0.FDMantissa** e.g.,  $0.314 \times 10^1$
  - The usefulness of the idea might not be as evident as it would be when we will deal with binary and use the FD as 1 i.e., either **1.Mantissa** or **0.1Mantissa** formats. The standard normalization scheme is **1.Mantissa for all numbers but ZERO!**

# Normalization

- **EXAMPLE:  $(13.4375)_{10} = (1101.0111)_2$**
- **TRY TO REPRESENT THE DECIMAL IN ANY OTHER WAY THAT FOLLOWS THE 1.M or 0.1M format?**

# Normalization

Normalization in scientific representation of numbers:

- The usefulness of the idea might not be as evident as it would be when we will deal with binary and use the FD as 1 i.e., either 1.Mantissa or 0.1Mantissa formats.
- **The standard normalization scheme is 1.Mantissa**
- The FD (fixed digit[bit]) need not be stored opening space for another significant bit

$$\text{number} = \text{sign} (1 + \text{significand}) * 2^{(\text{exponent}-127)}$$

# Excess or biased representation of signed binary numbers

## Recall:

- Signed binary number can be represented by fixing the **MSB** as sign bit
- There was problem in the above approach so we use 2's complement representation for signed number
- Are there any other representations?
  - Offset binary, also referred to as excess-K, excess-N, excess-e, excess code or biased representation, is a method for signed number representation where a signed number  $n$  is represented by the bit pattern corresponding to the **unsigned number  $n+K$** ,  $K$  being the biasing value or offset. There is no standard for offset binary, but most often the  $K$  for an  $n$ -bit binary word is  $K = 2^{n-1}$
  - The exponent is stored as an unsigned value which is suitable for comparison

## Comparison in 2's complement

- Following two numbers are in two's complement format: (check which is bigger)

**11111110 or 00000111**

- **Comparison is not trivial in two's complement**
- A more straightforward way keeping the frequency of comparison in mind, is to represent the exponent using excess-k codes which are explained next

# Excess or biased representation of signed binary numbers

$$\text{number} = \text{sign} (1 + \text{significand}) * 2^{(\text{exponent}-127)}$$

The actual exponent = (exponent-offset),  
offset =  $2^{(8)}-1 = 127$

Decimal	Offset binary, $K = 8$	Two's complement
7	1111	0111
6	1110	0110
5	1101	0101
4	1100	0100
3	1011	0011
2	1010	0010
1	1001	0001
0	1000	0000
-1	0111	1111
-2	0110	1110
-3	0101	1101
-4	0100	1100
-5	0011	1011
-6	0010	1010
-7	0001	1001



(a) Format

# Some examples from binary to float and float to binary

## Steps to get to the floating point representation:

1. Identify and set the sign bit (0: for positive, and 1: negative)
2. Convert the decimal to binary
3. Find the normalized mantissa (significand) in the standard form (1.Mantissa)
4. Find the exponent in Excess-k bit notation
5. Put the values on the format

**Example, represent  $(1460.125)_{10}$  in 32 bit floating point with 1.M format.**

1.  $(1460.125)_{10} \Rightarrow$  sign bit is 0
2.  $(10110110100.001)_2$
3. Normalization:  $1.0110110100001 * 2^{10}$
4. Exponent:  $10 + 127 = 137 = 10001001$
5.  $0\ 10001001\ 01101101000010000000000$  (purple is just padding of zeros to make it to 32 bits)



(a) Format



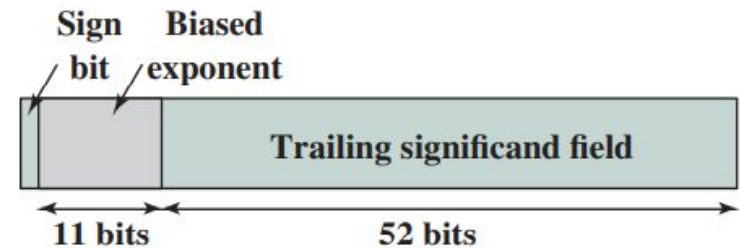
# Some examples from binary to float and float to binary

## Steps to get to the floating point representation:

1. Identify and set the sign bit (0: for positive, and 1: negative)
2. Convert the decimal to binary
3. Find the normalized mantissa (significand) in the standard form (1.Mantissa)
4. Find the exponent in Excess-k bit notation
5. Put the values on the format

Example, represent  $(1460.125)_{10}$  in double precision i.e. 64 bit floating point with 1.M format.

- 1.
- 2.
- 3.
- 4.
- 5.



(b) Binary64 format

# Some examples from binary to float and float to binary

**Convert 0.09375 into IEEE 754 single precision floating point binary**

**Step 1: Determine the sign bit**

Sign bit = 0

**Step 2: Convert to pure binary**

$$0.09375_{10} = 0.00011_2$$

$$0 \div 2 = 0 \quad \text{remainder } 0$$

$$0.09375 \times 2 = 0.1875 \quad 0$$

$$0.1875 \times 2 = 0.375 \quad 0$$

$$0.375 \times 2 = 0.75 \quad 0$$

$$0.75 \times 2 = 1.5 \quad 1$$

$$0.5 \times 2 = 1.0 \quad 1$$



**Step 3: Normalise for mantissa and unbiased exponent**

$$0.00011 = 1.1 \times 2^{-4}$$

**Step 4: Determine biased exponent**

$$-4 + 127 = 123_{10} = 01111011_2$$

**Step 5: Remove leading 1 from mantissa**

$$1.1 = 1$$

**0 01111011 100000000000000000000000**

# Some examples from binary to float and float to binary

**Convert -123.3 into IEEE 754 single precision floating point binary**

**Step 1: Determine the sign bit**

Sign bit = 1


**Step 2: Convert to pure binary**

$$123.3_{10} = 1111011.01001100110011001\overline{001}\dots_2$$

123	÷ 2	=	61	remainder	1
61	÷ 2	=	30	remainder	1
30	÷ 2	=	15	remainder	0
15	÷ 2	=	7	remainder	1
7	÷ 2	=	3	remainder	1
3	÷ 2	=	1	remainder	1
1	÷ 2	=	0	remainder	1



0.3	x 2	=	0.6	0
0.6	x 2	=	1.2	1
0.2	x 2	=	0.4	0
0.4	x 2	=	0.8	0
0.8	x 2	=	1.6	1
0.6	x 2	=	1.2	1
0.2	x 2	=	0.4	0



**Step 3: Normalise for mantissa and unbiased exponent**

$$1111011.0100110011001\overline{001}\dots = 1.1110110100110011001\overline{001}\dots \times 2^6$$

**Step 4: Determine biased exponent**

$$6 + 127 = 133_{10} = 10000101_2$$

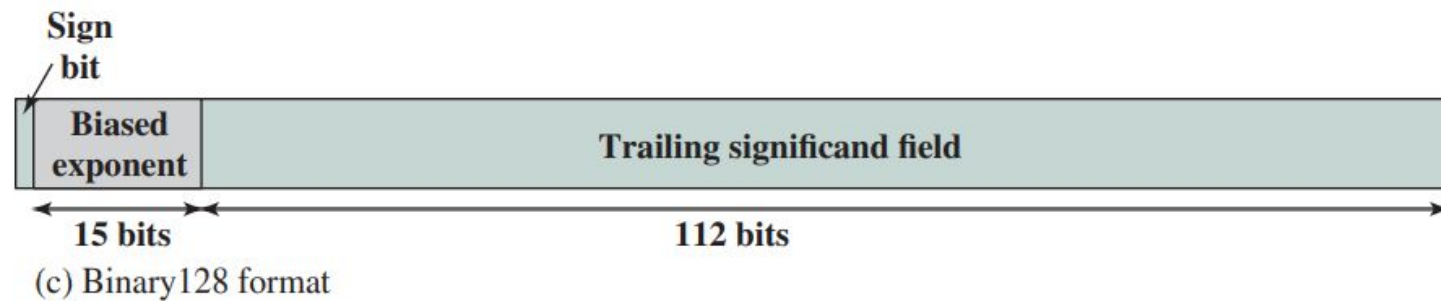
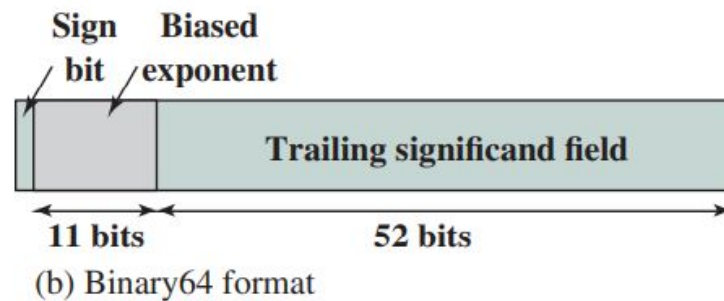
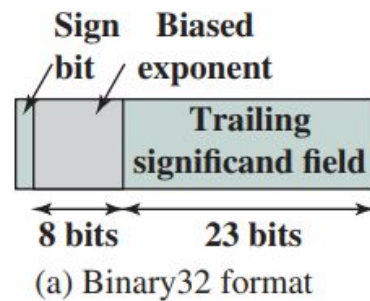
**Step 5: Remove leading 1 from mantissa**

$$1.1110110100110011001\overline{001}\dots = 11101101001100110011001$$

**Step 6: Round mantissa up or down if necessary**

Round off to fit to the number of bits allowed. If there is one add 1 else leave as is

# Floating point representation



**Figure 10.21** IEEE 754 Formats

## Special values

### Reserved Exponent Values

Exponent Value	Mantissa	Represents
11111111	All zeros	Infinity ( $\infty$ )
11111111	Not all zeros	Not a number (NaN)
00000000	All zeros	Zero
00000000	Not all zeros	Subnormal (very small)

# Floating point representation

**Table 10.3** 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	$2^{-149}$	$2^{-1074}$	$2^{-16494}$

Note: \* Not including implied bit and not including sign bit.

# Floating point arithmetic:

**Table 10.6** Floating-Point Numbers and Arithmetic Operations

Floating-Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$\left. \begin{aligned} X + Y &= (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E} \\ X - Y &= (X_S \times B^{X_E - Y_E} - Y_S) \times B^{Y_E} \end{aligned} \right\} X_E \leq Y_E$ $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left( \frac{X_S}{Y_S} \right) \times B^{X_E - Y_E}$

Examples:

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

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

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

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

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

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

- For addition and subtraction, it is necessary to ensure that both operands have the **same exponent value**.



## IEEE guideline: limitations

- Infinitely many reals, finitely many IEEE floating point numbers.
- Cannot represent some numbers, like  $1/3$  or  $1/10$ .

Addition is not necessarily associative:

Let  $x = -1.1 \times 2^{120}$ ,  $y = 1.1 \times 2^{120}$ ,  $z = 1.0$

$$\begin{aligned}(x+y)+z &= (-1.1 \times 2^{120} + 1.1 \times 2^{120}) + 1.0 \\ &= 0 + 1.0 = 1.0\end{aligned}$$

$$\begin{aligned}x+(y+z) &= -1.1 \times 2^{120} + (1.1 \times 2^{120} + 1.0) \\ &= -1.1 \times 2^{120} + 1.1 \times 2^{120} = 0.0\end{aligned}$$

Rounding errors:

$$0.1+0.1+0.1 \neq 0.3$$

$$2^{100} + 2^{-100} = 2^{100}$$



## IEEE guideline: limitations

### “Epsilon”

- The difference between 1.0 and the next available floating point number
- Useful for programmatic “almost equal” computations
- `std::numeric_limits<T>::epsilon`

How would you compute the square root of a number?

**Watch this**



[Stanford Seminar: Beyond Floating Point: Next Generation Computer Arithmetic - YouTube](#)

# Boolean Algebra!

Boolean algebra turns out to be a convenient tool in two areas:

- **Analysis:** It is an economical way of describing the function of digital circuitry.
- **Design:** Given a desired function, Boolean algebra can be applied to develop a simplified implementation of that function.

## Boolean operators!



Buffer	 $A$
NOT (inverter)	 $\bar{A}$ or $\neg A$

INPUT	OUTPUT
A	Q
0	0
1	1

INPUT	OUTPUT
A	Q
0	1
1	0

# Boolean Algebra!

## Boolean operators!



AND	 $A \cdot B$ or $A \wedge B$
OR	 $A + B$ or $A \vee B$

INPUT		OUTPUT
A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

INPUT		OUTPUT
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

# Boolean Algebra!

## Boolean operators!

NAND	 $\overline{A \cdot B} \text{ or } A \uparrow B$
NOR	 $\overline{A + B} \text{ or } A \downarrow B$

INPUT		OUTPUT
A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

INPUT		OUTPUT
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

# Boolean Algebra!

## Boolean operators!

XOR



$$A \oplus B \text{ or } A \vee B$$

INPUT		OUTPUT
A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

XNOR



$$\overline{A \oplus B} \text{ or } A \odot B$$

INPUT		OUTPUT
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

# Boolean Algebra!

## Universal Gate

- Charles Sanders Peirce (during 1880–81) showed that NOR gates alone (or alternatively NAND gates alone) can **be used to reproduce the functions of all the other logic gates**, but his work on it was unpublished until 1933.
- The first published proof was by Henry M. Sheffer in 1913, so the NAND logical operation is sometimes called Sheffer stroke; the logical NOR is sometimes called Peirce's arrow.
- Consequently, these gates are sometimes called universal logic gates.

# Boolean Algebra!

## Universal Gate

Typically, not all gate types are used in implementation. Design and fabrication are simpler if only one or two types of gates are used. Thus, it is important to identify *functionally complete* sets of gates. This means that any Boolean function can be implemented using only the gates in the set. The following are functionally complete sets:

- AND, OR, NOT
- AND, NOT
- OR, NOT
- NAND
- NOR



# Boolean Algebra!

Boolean algebra turns out to be a convenient tool in two areas:

- **Analysis:** It is an economical way of describing the function of digital circuitry.
- **Design:** Given a desired function, Boolean algebra can be applied to develop a simplified implementation of that function.

**Table 11.1** Boolean Operators

(a) Boolean Operators of Two Input Variables

<b>P</b>	<b>Q</b>	<b>NOT P</b> <b>(<math>\overline{P}</math>)</b>	<b>P AND Q</b> <b>(<math>P \cdot Q</math>)</b>	<b>P OR Q</b> <b>(<math>P + Q</math>)</b>	<b>P NAND Q</b> <b>(<math>\overline{P \cdot Q}</math>)</b>	<b>P NOR Q</b> <b>(<math>\overline{P + Q}</math>)</b>	<b>P XOR Q</b> <b>(<math>P \oplus Q</math>)</b>
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

(b) Boolean Operators Extended to More than Two Inputs ( $A, B, \dots$ )

<b>Operation</b>	<b>Expression</b>	<b>Output = 1 if</b>
AND	$A \cdot B \cdot \dots$	All of the set $\{A, B, \dots\}$ are 1.
OR	$A + B + \dots$	Any of the set $\{A, B, \dots\}$ are 1.
NAND	$\overline{A \cdot B \cdot \dots}$	Any of the set $\{A, B, \dots\}$ are 0.
NOR	$\overline{A + B + \dots}$	All of the set $\{A, B, \dots\}$ are 0.
XOR	$A \oplus B \oplus \dots$	The set $\{A, B, \dots\}$ contains an odd number of ones.

## Boolean Algebra!

The two bottommost expressions are referred to as DeMorgan's theorem. We can restate them as follows:

$$A \text{ NOR } B = \overline{A} \text{ AND } \overline{B}$$

$$A \text{ NAND } B = \overline{A} \text{ OR } \overline{B}$$

# Boolean Algebra!

**Table 11.2** Basic Identities of Boolean Algebra

Basic Postulates		
$A \cdot B = B \cdot A$	$A + B = B + A$	Commutative Laws
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$	Distributive Laws
$1 \cdot A = A$	$0 + A = A$	Identity Elements
$A \cdot \bar{A} = 0$	$A + \bar{A} = 1$	Inverse Elements
Other Identities		
$0 \cdot A = 0$	$1 + A = 1$	Associative Laws
$A \cdot A = A$	$A + A = A$	
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$	DeMorgan's Theorem
$\overline{A \cdot B} = \bar{A} + \bar{B}$	$\overline{A + B} = \bar{A} \cdot \bar{B}$	

## Fun programming exercise:

If you are given  $2n+1$  numbers, and  $n$  of them are repeated, what is the fastest way to find the one that is not repeated?