ARITHMETIC FOR COMPUTERS

FOR CSE & CST, IV SEM

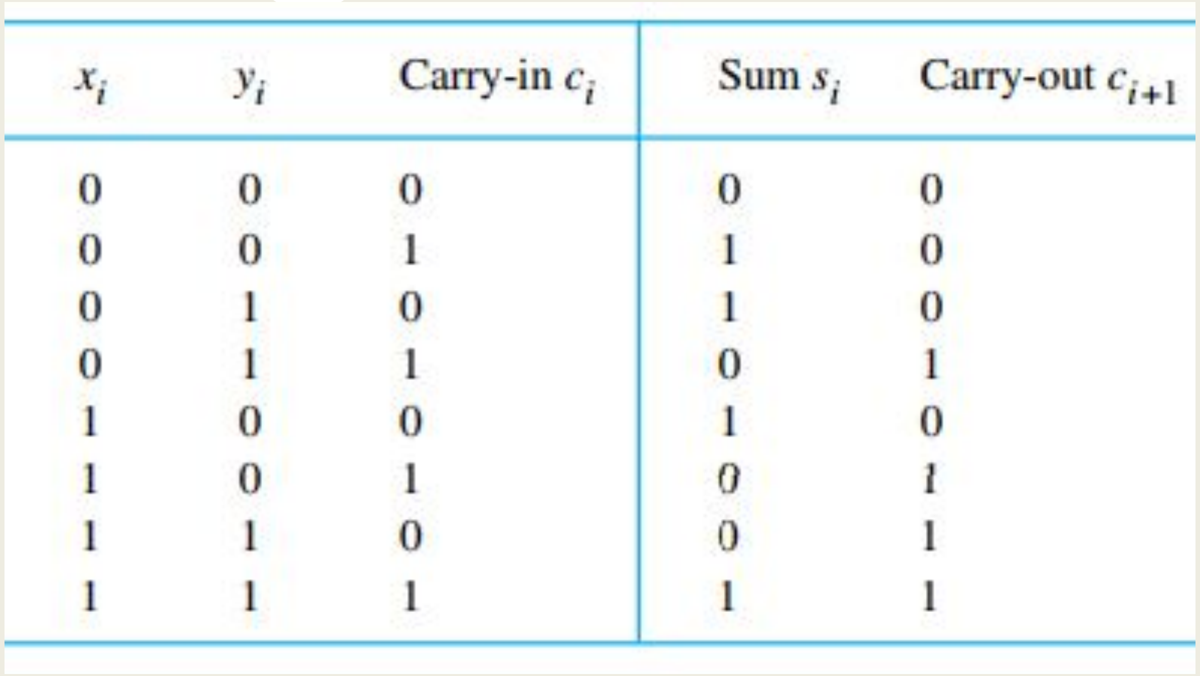
PRESENTED BY : TAPAS CH. SINGH

Subject Content

- **ARITHMETIC FOR COMPUTERS:** Addition and Subtraction, Fast Adders, Binary Multiplication, Fast Multiplication, Binary Division and its techniques, Floating Point Numbers, Representation, Arithmetic Operations.

Addition and Subtraction of Signed Numbers

- Figure shows the truth table for the sum and carry-out functions for adding equally weighted bits *xi* and *yi* in two numbers *X* and *Y* .
- Each stage of the addition process must accommodate a carry-in bit. We use *ci* to represent the carry-in to stage *i*, which is the same as the carry-out from stage (*i*-1).

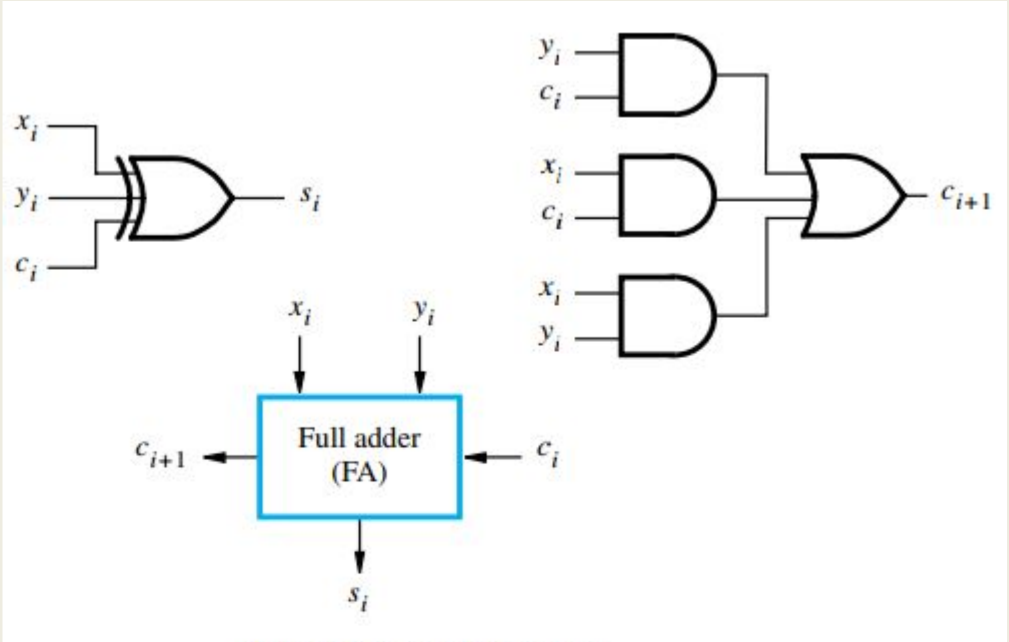


x_i	y_i	Carry-in c_i	Sum s_i	Carry-out c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$s_i = \overline{x}_i\overline{y}_i c_i + \overline{x}_i y_i \overline{c}_i + x_i \overline{y}_i \overline{c}_i + x_i y_i c_i = x_i \oplus y_i \oplus c_i$$

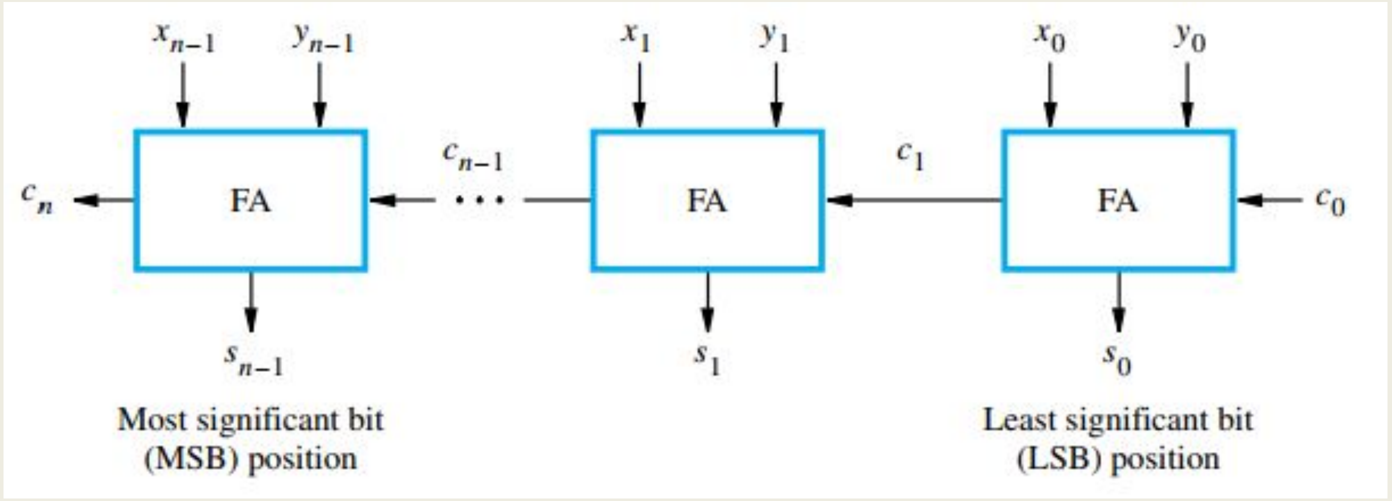
$$c_{i+1} = y_i c_i + x_i c_i + x_i y_i$$

- The logic expression for *Si* in Figure can be implemented with a 3-input XOR gate, used in Figure as part of the logic required for a single stage of binary addition. The carry-out function, *ci*+1, is implemented with an AND-OR circuit, as shown in fig.



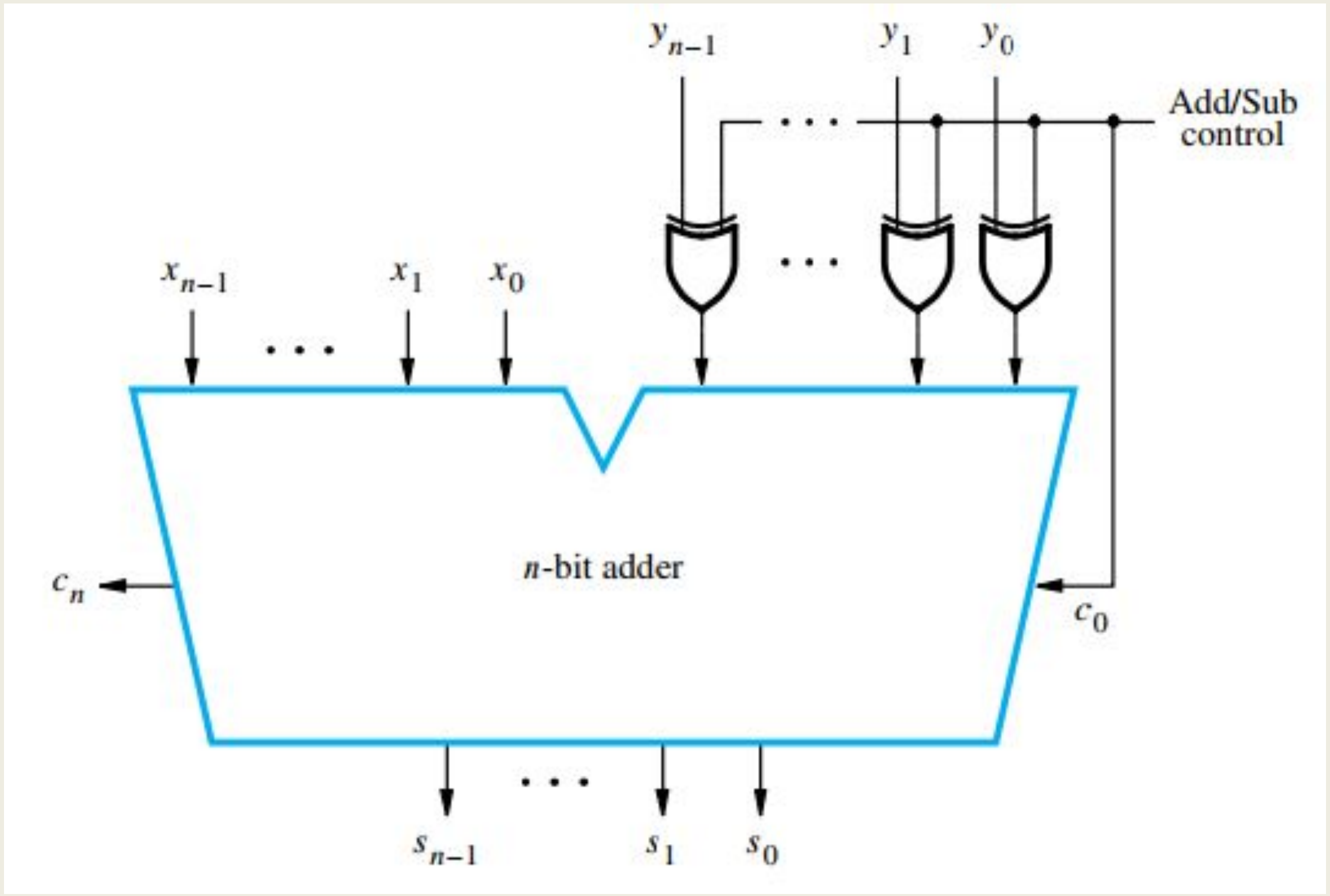
(a) Logic for a single stage

An n-bit ripple-carry adder



- A cascaded connection of *n* full-adder blocks can be used to add two *n*-bit numbers, as shown in Figure. Since the carries must propagate, or ripple, through this cascade, the configuration is called a *ripple-carry adder*.

Binary addition/subtraction logic circuit



- In order to perform the subtraction operation X - Y on 2's complement numbers X and Y , we form the 2's-complement of Y and add it to X . The logic circuit shown in Figure can be used to perform either addition or subtraction based on the value applied to the Add/Sub input control line. This line is set to 0 for addition, applying Y unchanged to one of the adder inputs along with a carry-in signal, C0, of 0. When the Add/Sub control line is set to 1, the Y number is 1's complemented (that is, bit-complemented) by the XOR gates and C0 is set to 1 to complete the 2's-complementation of Y .

Design of Fast Adders

- If an *n*-bit ripple-carry adder is used in the addition/subtraction., it may have too much delay in developing its outputs, so through S_{n-1} and C_n.

- The delay through a network of logic gates depends on the integrated circuit electronic technology used in fabricating the network and on the number of gates in the paths from inputs to outputs.

- Two approaches can be taken to reduce delay in adders. The first approach is to use the fastest possible electronic technology. The second approach is to use a logic gate network called a carry-look ahead network,

-

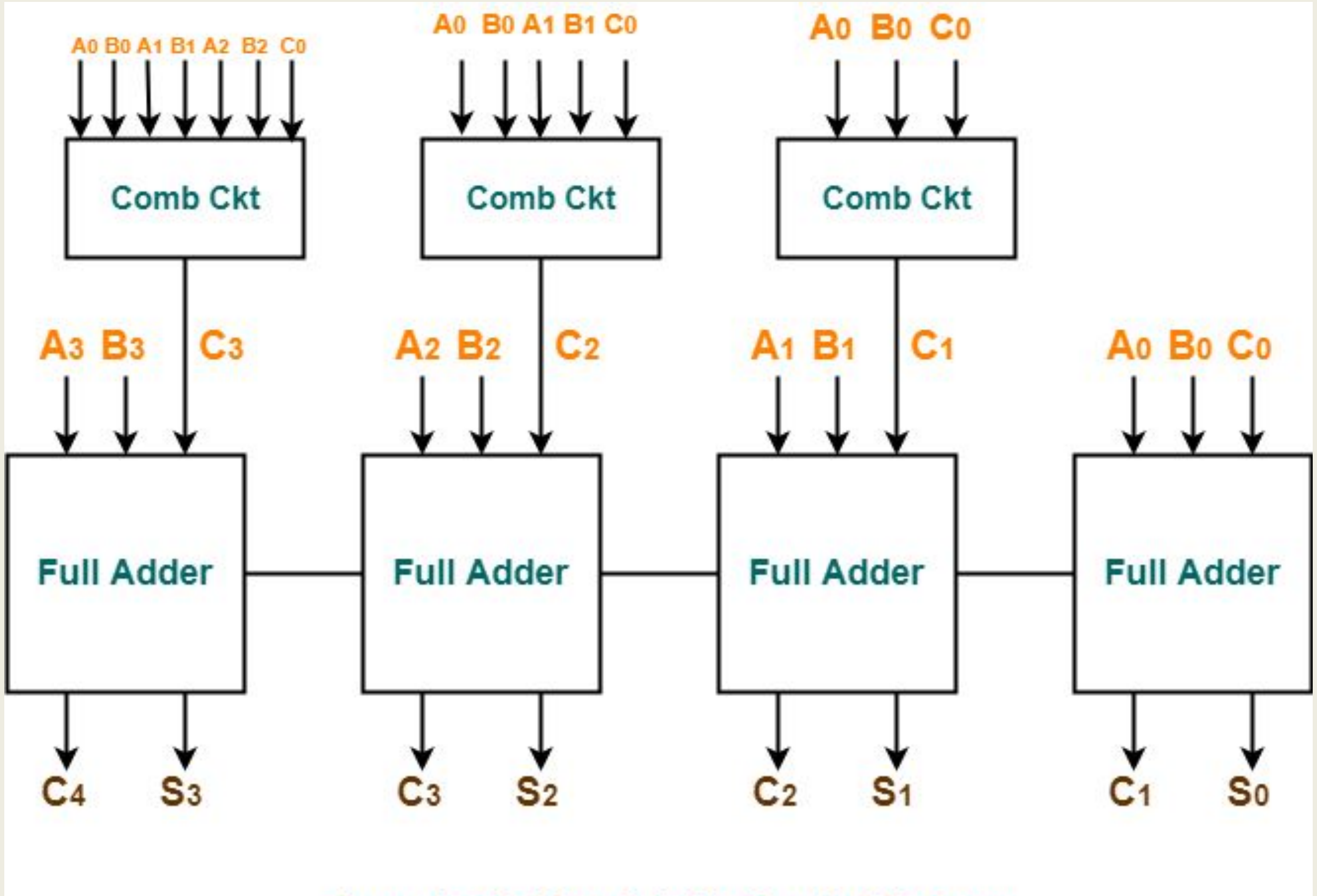
Carry-Look ahead Addition

- A fast adder circuit must speed up the generation of the carry signals. The logic expressions
  for Si (sum) and $Ci+1$ (carry-out) of stage i (see Figure) are
- $Si = Xi \oplus Yi \oplus Ci$
  and
  $Ci+1 = XiYi + Xici + YiCi$
- Factoring the second equation into
  $Ci+1 = XiYi + (Xi + Yi)Ci$
- we can write
  $Ci+1 = Gi + PiCi$
- where
  $Gi = XiYi$ and $Pi = Xi + Yi$

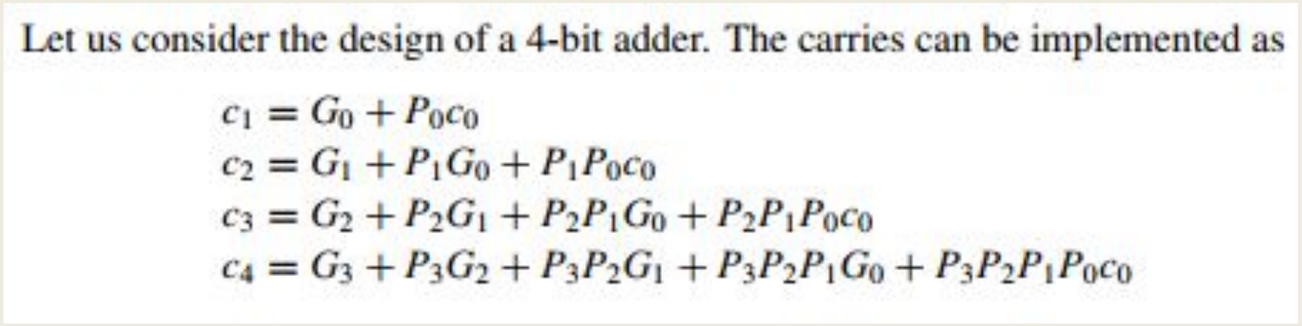
- The expressions G_i and P_i are called the *generate* and *propagate* functions for stage i. If the generate function for stage i is equal to 1, then $C_{i+1} = 1$, independent of the input carry, C_i.
- This occurs when both x_i and y_i are 1. The propagate function means that an input carry will produce an output carry when either x_i is 1 or y_i is 1.
- The propagate function means that an input carry will produce an output carry when either x_i is 1 or y_i is 1. All G_i and P_i functions can be formed independently and in parallel in one logic-gate delay after the X and Y operands are applied to the inputs of an n-bit adder.
- Each bit stage contains an AND gate to form G_i, an OR gate to form P_i, and a three-input XOR gate to form s_i. A simpler circuit can be derived by observing that an adequate propagate function can be realized as $P_i = x_i \oplus y_i$, which differs from $P_i = x_i + y_i$ only when $x_i = y_i = 1$.
- Expanding ci in terms of i - 1 subscripted variables and substituting into the ci+1 expression, we obtain

  $$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} C_{i-1}$$
-

4-bit adder



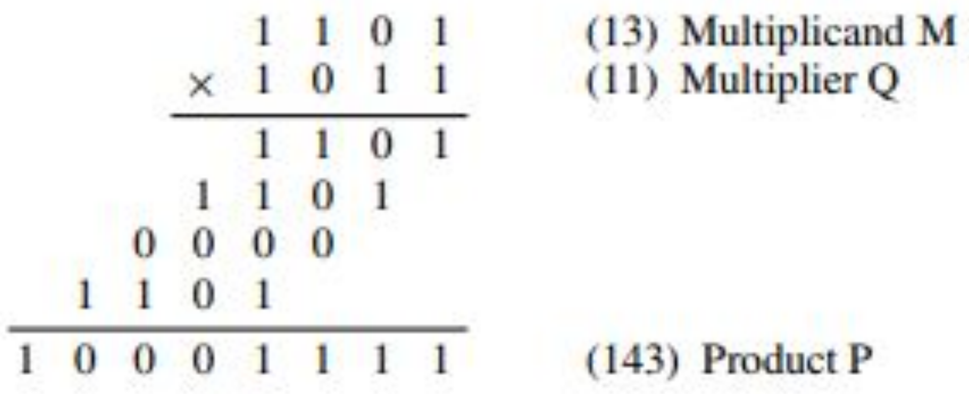
Carry Look Ahead Adder Logic Diagram



Let us consider the design of a 4-bit adder. The carries can be implemented as

$$
\begin{aligned}
c_1 &= G_0 + P_0c_0 \\
c_2 &= G_1 + P_1G_0 + P_1P_0c_0 \\
c_3 &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0 \\
c_4 &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0
\end{aligned}
$$

Multiplication of Unsigned Numbers

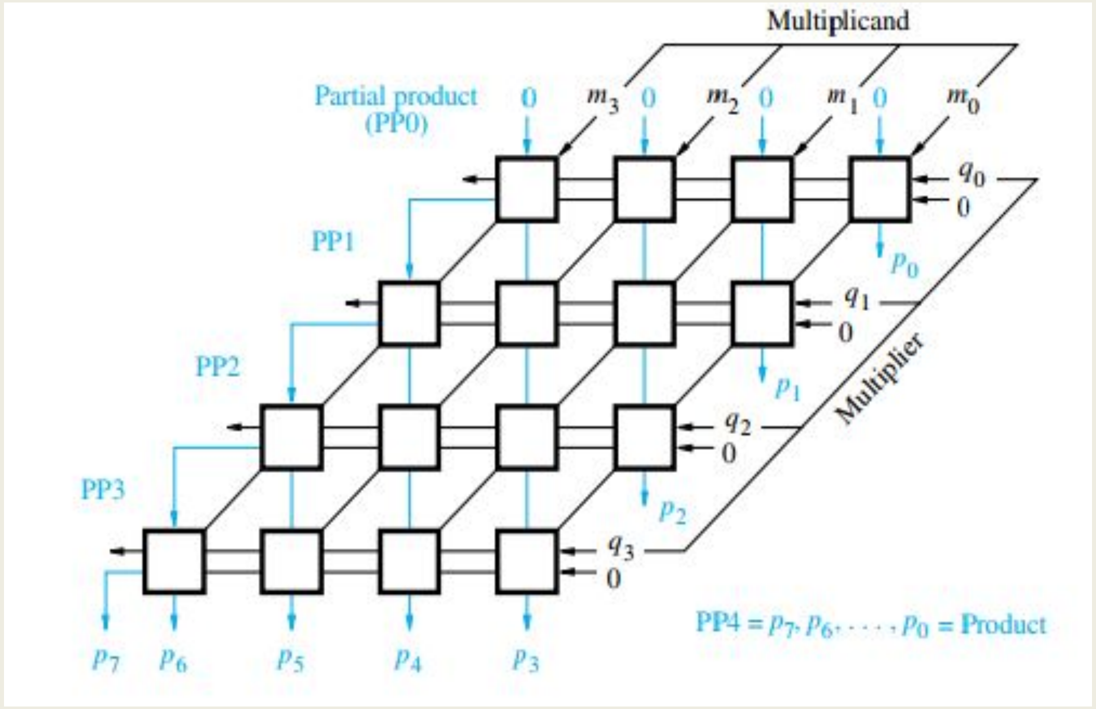


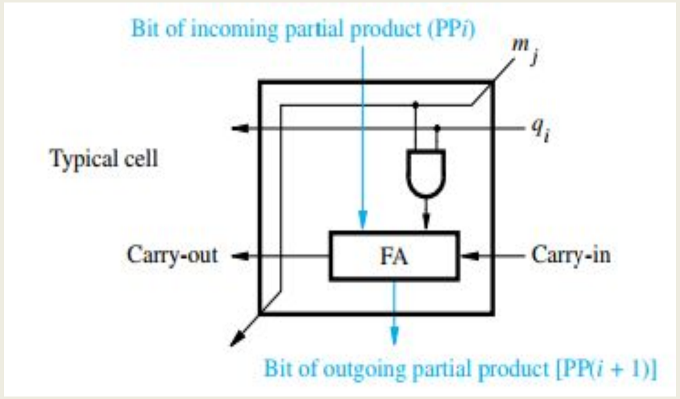
```
                1  1  0  1        (13)  Multiplicand M
             ×  1  0  1  1        (11)  Multiplier Q
             -------------
                1  1  0  1
             1  1  0  1
          0  0  0  0
       1  1  0  1
----------------------------
 1  0  0  0  1  1  1  1           (143)  Product P
```

(a) Manual multiplication algorithm

Array Multiplier

- **Multiplication of Unsigned Numbers**



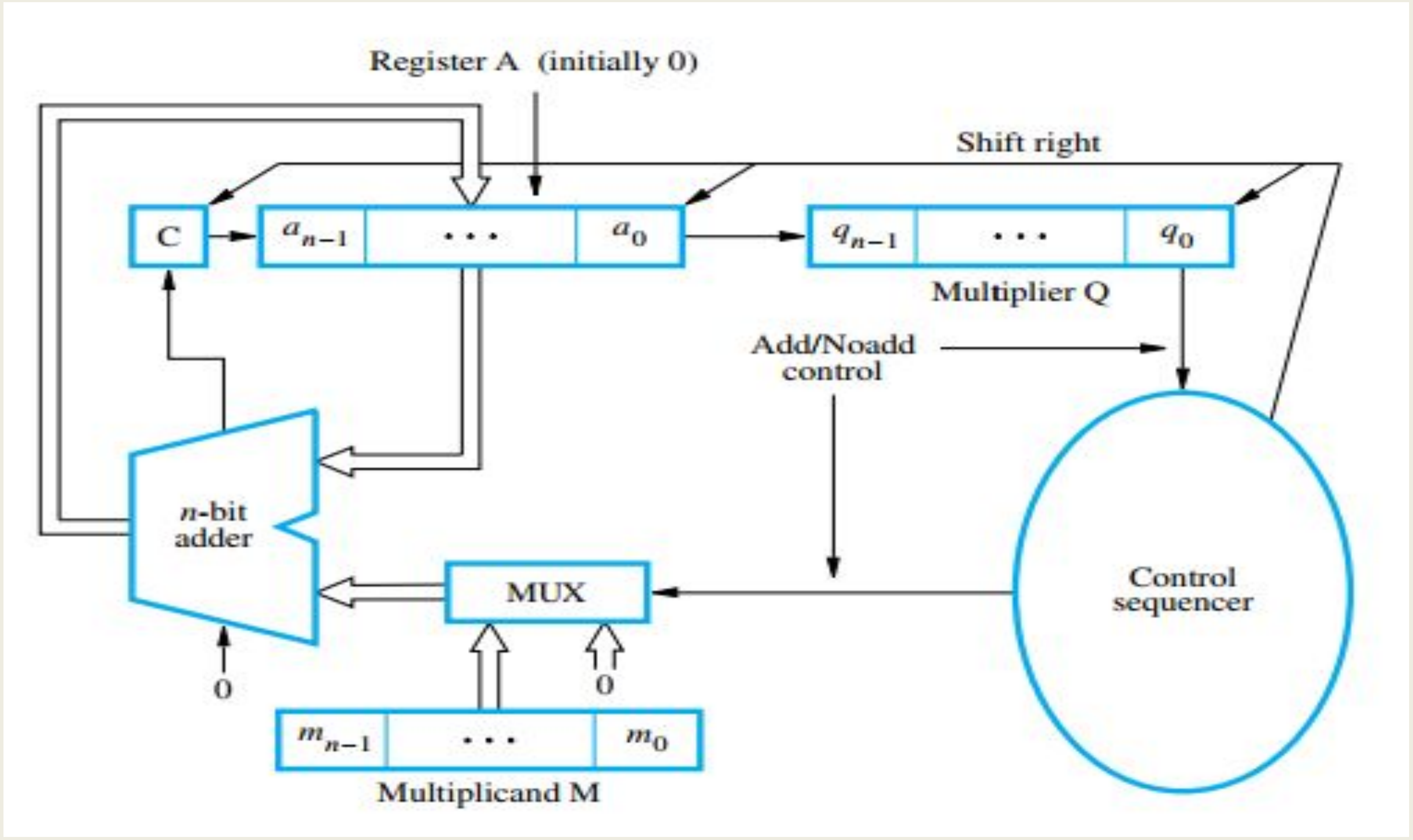


Bit of incoming partial product (PPi)
m_j
q_i
Typical cell
Carry-out
FA
Carry-in
Bit of outgoing partial product [PP(i + 1)]

- Binary multiplication of unsigned operands can be implemented in a combinational, two dimensional, logic array, as shown in Figure for the 4-bit operand case.
- The main component in each cell is a full adder, FA. The AND gate in each cell determines whether a multiplicand bit, *mj*, is added to the incoming partial-product bit, based on the value of the multiplier bit, *qi*. Each row *i*, where 0 ≤ *i* ≤ 3, adds the multiplicand (appropriately shifted) to the incoming partial product, PP*i*, to generate the outgoing partial product, PP(*i* + 1), if *qi* = 1.
- If *qi* = 0, PP*i* is passed vertically downward unchanged. PP0 is all 0s, and PP4 is the desired product. The multiplicand is shifted left one position per row by the diagonal signal path. We note that the row-by-row addition done in the array circuit differs from the usual hand addition described previously, which is done column-by-column.
-

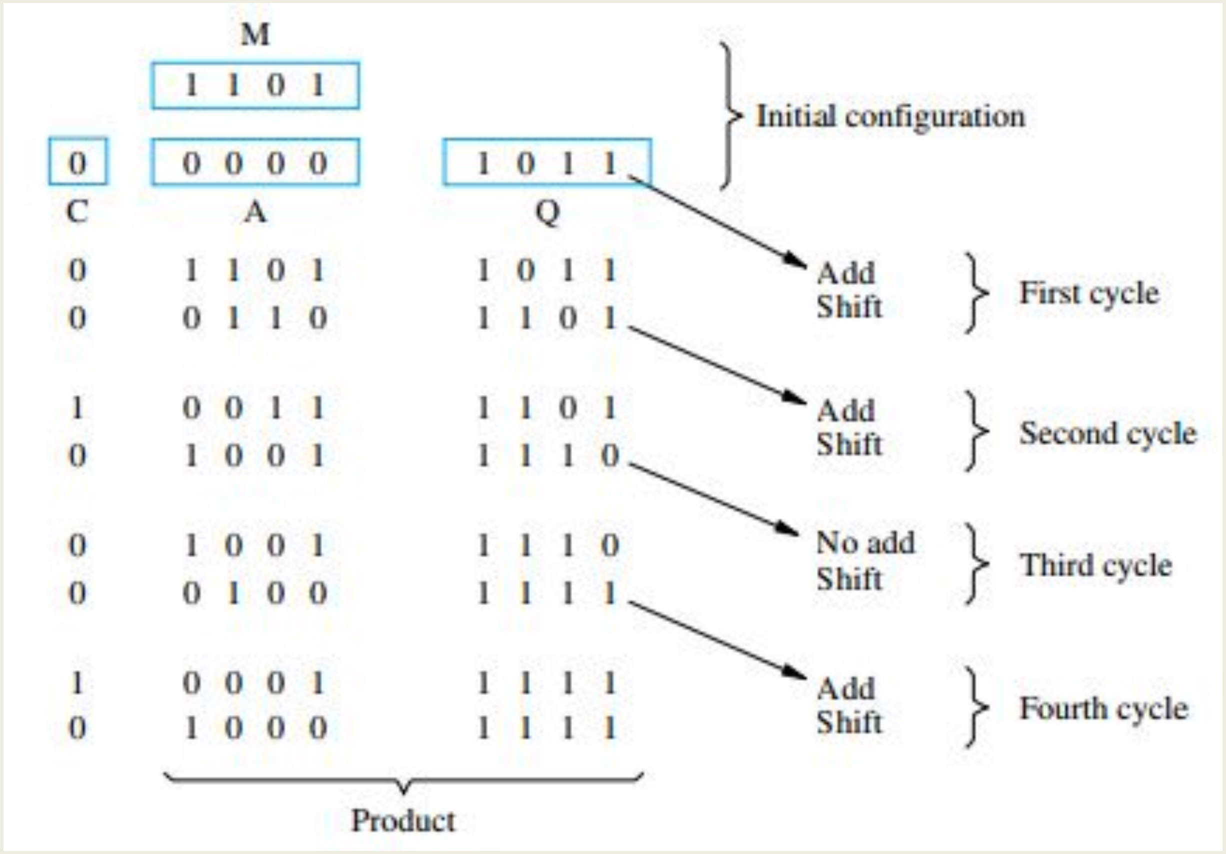
Sequential Circuit Multiplier

Multiplication of Signed Numbers



- The combinational array multiplier just described uses a large number of logic gates for multiplying numbers of practical size, such as 32- or 64-bit numbers.
- Multiplication of two *n*-bit numbers can also be performed in a sequential circuit that uses a single *n*-bit adder.
- This circuit performs multiplication by using a single *n*-bit adder *n* times to implement the spatial addition performed by the *n* rows of ripple-carry adders in Figure
- Registers A and Q are shift registers, concatenated as shown. Together, they hold partial product PP*i* while multiplier bit *qi* generates the signal Add/No add. This signal causes the multiplexer MUX to select 0 when *qi* = 0, or to select the multiplicand M when *qi* = 1, to be added to PP*i* to generate PP(*i* + 1). The product is computed in *n* cycles
- The carry-out from the adder is stored in flip-flop C, shown at the left end of register A
- At the start, the multiplier is loaded into register Q, the multiplicand into register M, and C and A are cleared to 0. At the end of each cycle, C, A, and Q are shifted right one bit position to allow for growth of the partial product as the multiplier is shifted out of register Q. Because of this shifting, multiplier bit *qi* appears at the LSB position of Q to generate the Add/Noadd signal at the correct time, starting with *q*0 during the first cycle, *q*1 during the second cycle, and so on.
-

- Example : (1101)2 X (1011)2 = ?
- Sequential circuit binary multiplier



The Booth Algorithm

- Booth algorithm gives a procedure for multiplying binary integers in signed-2's complement representation.

- The Booth algorithm generates a 2*n*-bit product and treats both positive and negative 2’scomplement *n*-bit operands uniformly.

Example of: Using Booths Algorithm solve (-9) x (-13) = +117

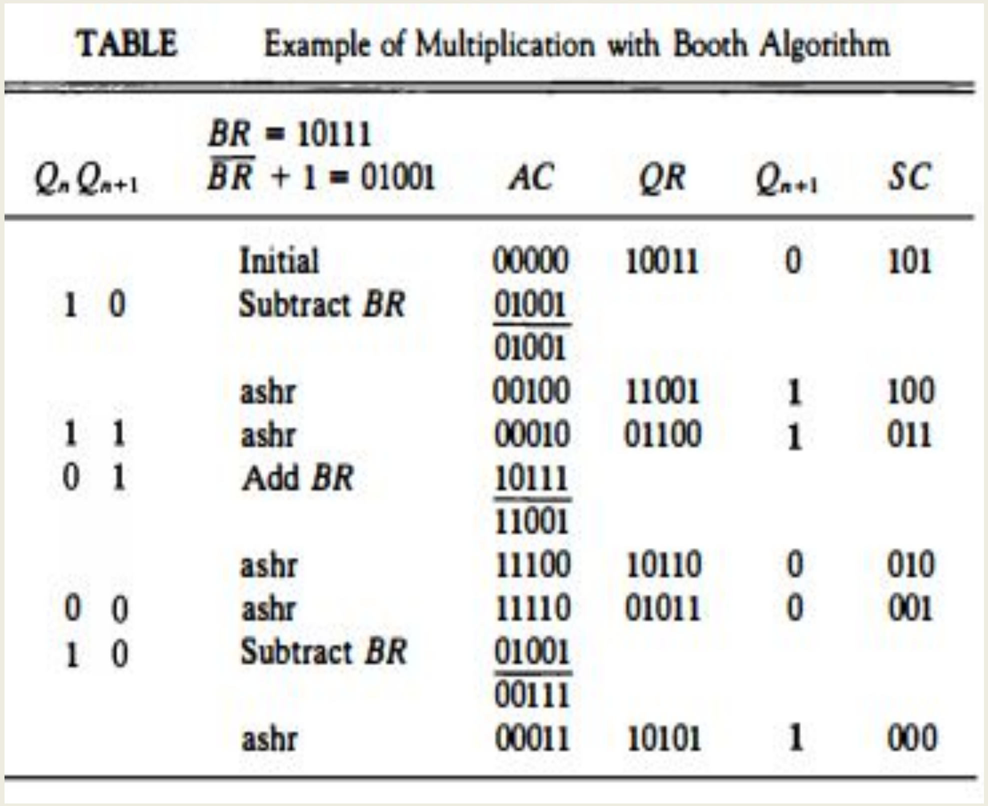
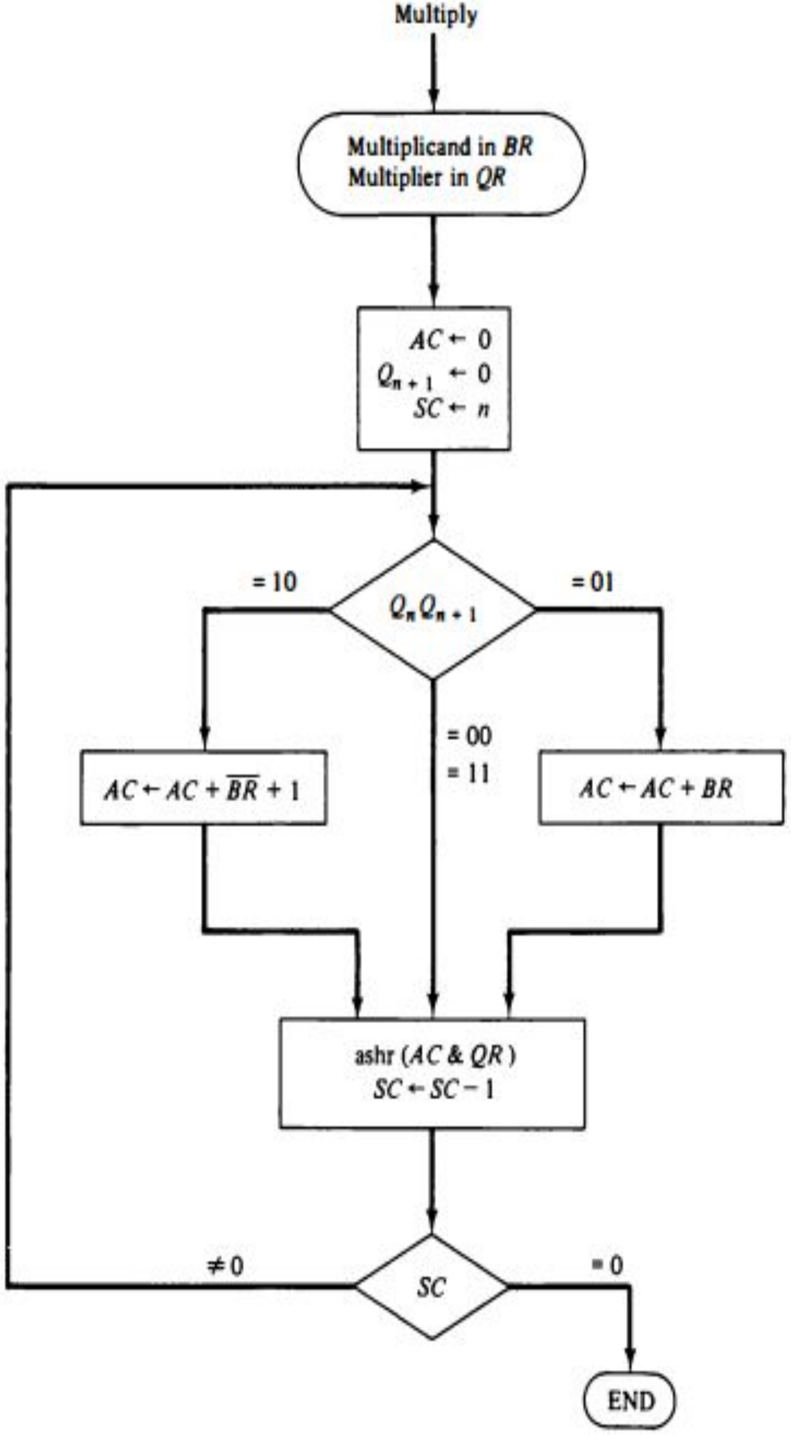


TABLE Example of Multiplication with Booth Algorithm

$Q_n Q_{n+1}$	BR = 10111 $\overline{BR}$ + 1 = 01001	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000

- asar: shift right



Fast Multiplication

- The first technique guarantees that the maximum number of summands (versions of the multiplicand) that must be added is $n/2$ for n-bit operands. The second technique leads to adding the summands in parallel.
  3 Methods for fast Multiplication:
- **Bit-Pair Recoding of Multipliers**
- **Carry-Save Addition of Summands**
- **Summand Addition Tree using 3-2 Reducers**

Bit-Pair Recoding of Multipliers

- Bit-pair recoding is the product of the multiplier results in using at most one summand for each pair of bits in the multiplier. It is derived directly from the Booth algorithm. Grouping the Booth-recoded multiplier bits in pairs will decrease the multiplication only by n/2 summands.
- Consider the following binary numbers:

$$A = 010111 \; (+23)$$

$$B = 110110 \; (-10 \rightarrow \text{2's compliment of } 110110)$$

- Multiply the signed 2's complement numbers using the bit-pair recoding of the multiplier.
- Thus, the resultant value is. -230
-

```
                0 1  0 1 1 1             23
            ×    −1    +2  −2          ×−10 (Booth recoding 0 −1 +1  0  −1  0)
         ---------------------
 1 1 1 1 1  1 0 1  0 0  1 0
 0 0 0 0 1  0 1  1  1 0
 1 1 1  0 1  0 0 1
 ------------------------
 1 1 1 1 0 0 0 1 1  0  1  0          = −230
```

- Consider the following binary numbers:

$$A = 110011 \ (-13 \rightarrow \text{2's compliment of } 110011)$$

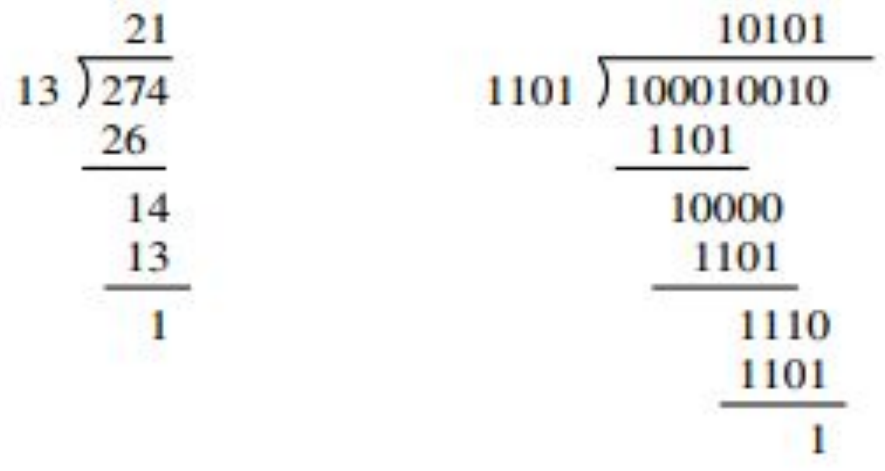
$$B = 101100 \ (-20 \rightarrow \text{2's compliment of } 101100)$$

- Multiply the signed 2’s complement numbers using the bit-pair recoding of the multiplier.

```
              1  1  0  0  1  1        −13
                ×    −1   −1          ×−20  (Booth recoding −1 +1 0 −1 0 0)
            ----------------
0000000 0 0 0  0 0
0000001 1 0 1                    -2's compliment
0000110 1
--------------------------
0001000 0 0 1  0 0           = (+260)
```

- Thus, the resultant value is .260

Integer Division

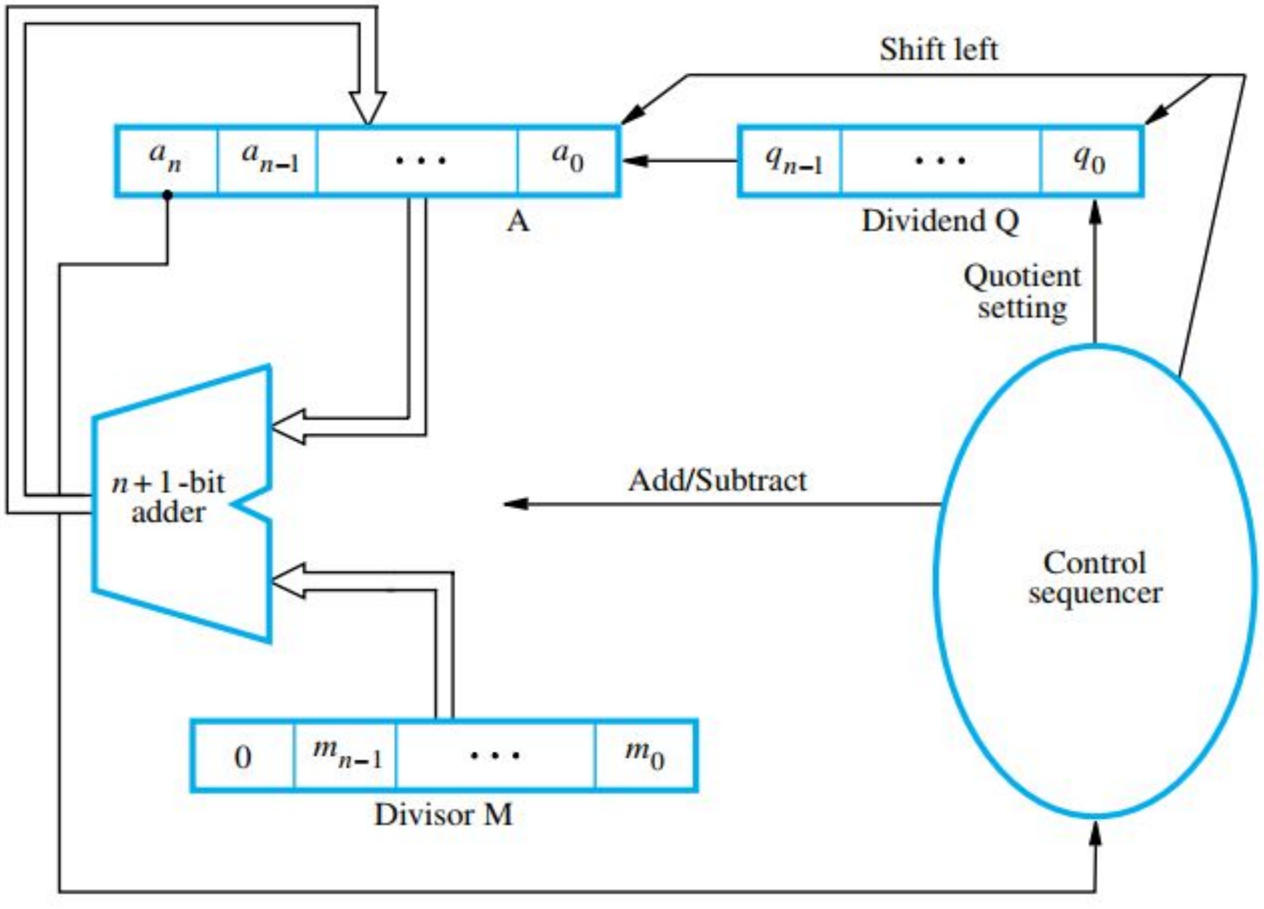


Longhand division examples.

Two types of division :-

(a) Restoring Division
(b) Non-Restoring Division

Restoring Division



Circuit arrangement for binary division.

- An *n*-bit positive divisor is loaded into register M and an *n*-bit positive dividend is loaded into register Q at the start of the operation.
- Register A is set to 0. After the division is complete, the *n*-bit quotient is in register Q and the remainder is in register A.
- The required subtractions are facilitated by using 2's complement arithmetic.
- The extra bit position at the left end of both A and M accommodates the sign bit during subtractions.

- Do the following three steps *n* times:

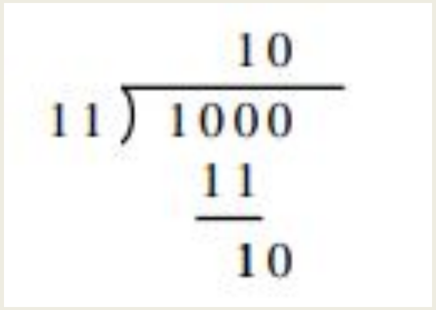
  1. Shift A and Q left one bit position.
  2. Subtract M from A, and place the answer back in A.
  3. If the sign of A is 1, set q_o to 0 and add M back to A (that is, restore A); otherwise, set q_o to 1.

Take the Example: Here M= 00011, -M = 11101 (2's comp.)

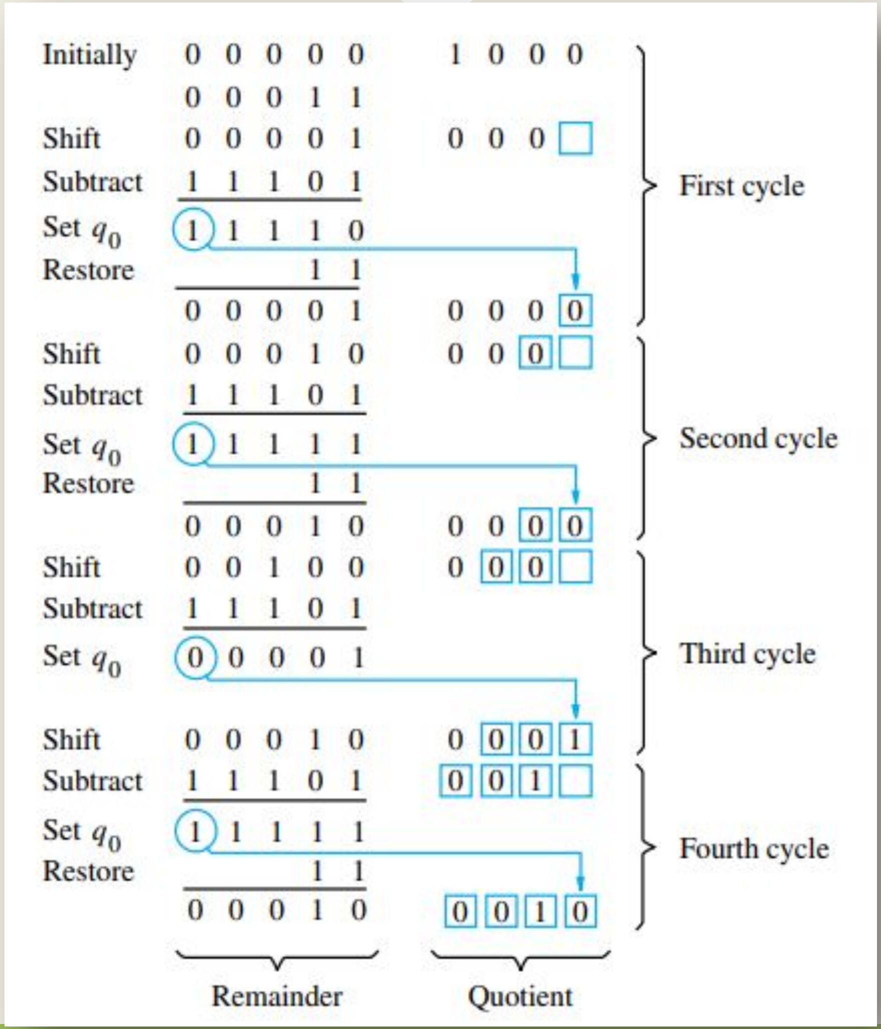
Q = 1000, A = Accumulator

Divisor : 11

Dividend: 1000



A restoring division example



	Remainder	Quotient	
Initially	0 0 0 0 0	1 0 0 0	
	0 0 0 1 1		
Shift	0 0 0 0 1	0 0 0 □	
Subtract	1 1 1 0 1		First cycle
Set q_0	$\bigcirc\!\!\!\!1$ 1 1 1 0		
Restore	1 1		
	0 0 0 0 1	0 0 0 $\boxed{0}$	
Shift	0 0 0 1 0	0 0 $\boxed{0}$ □	
Subtract	1 1 1 0 1		
Set q_0	1 1 1 1 1		Second cycle
Restore	1 1		
	0 0 0 1 0	0 0 $\boxed{0}$ $\boxed{0}$	
Shift	0 0 1 0 0	0 $\boxed{0}$ $\boxed{0}$ □	
Subtract	1 1 1 0 1		
Set q_0	0 0 0 0 1		Third cycle
Shift	0 0 0 1 0	0 $\boxed{0}$ $\boxed{0}$ $\boxed{1}$	
Subtract	1 1 1 0 1	$\boxed{0}$ $\boxed{0}$ $\boxed{1}$ □	
Set q_0	1 1 1 1 1		Fourth cycle
Restore	1 1		
	0 0 0 1 0	$\boxed{0}$ $\boxed{0}$ $\boxed{1}$ $\boxed{0}$	

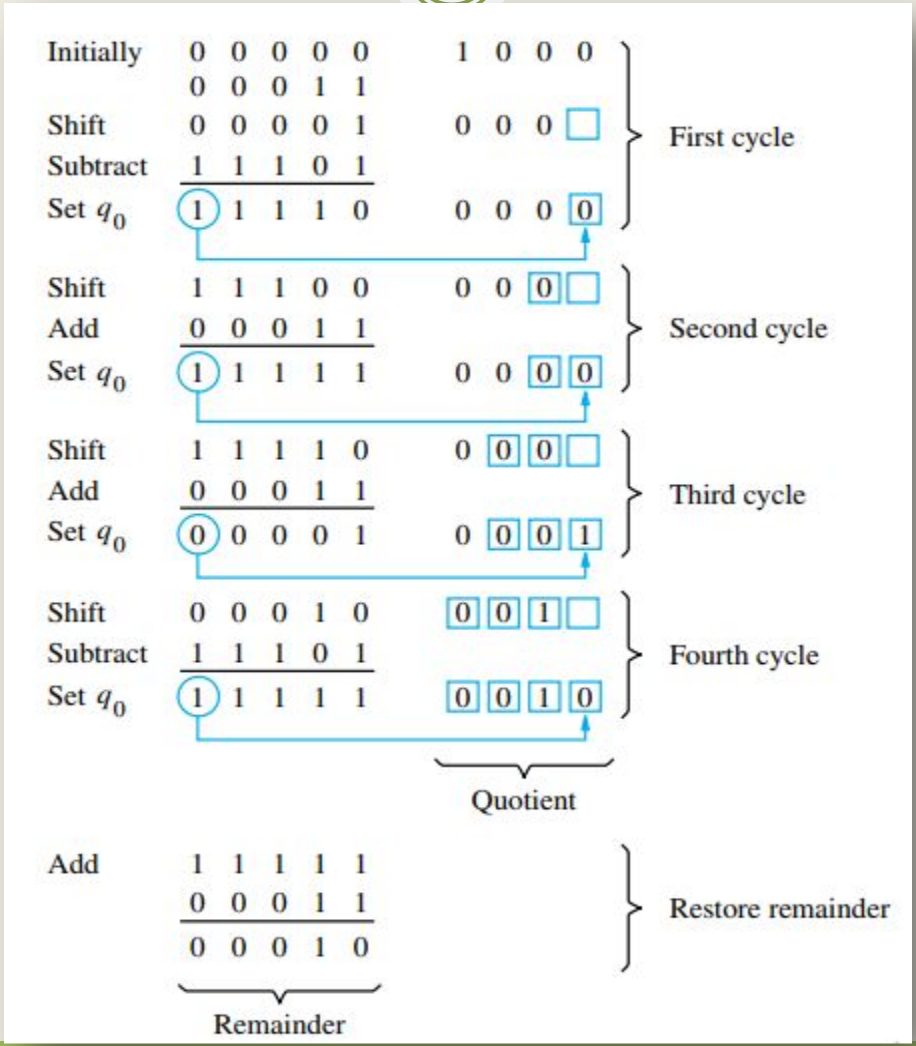
Non-restoring division

- The restoring division algorithm can be improved by avoiding the need for restoring A after an unsuccessful subtraction.
- Subtraction is said to be unsuccessful if the result is negative. Consider the sequence of operations that takes place after the subtraction operation in the preceding algorithm. If A is positive, we shift left and subtract M, that is, we perform 2A - M.
- If A is negative, we restore it by performing A + M, and then we shift it left and subtract M. This is equivalent to performing 2A + M. The q_0 bit is appropriately set to 0 or 1 after the correct operation has been performed.

- Do the following two steps *n* times:

  1. If the sign of A is 0, shift A and Q left one bit position and subtract M from A;
  otherwise, shift A and Q left and add M to A.
  2. Now, if the sign of A is 0, set q_o to 1; otherwise, set q_o to 0.

A non-restoring division example



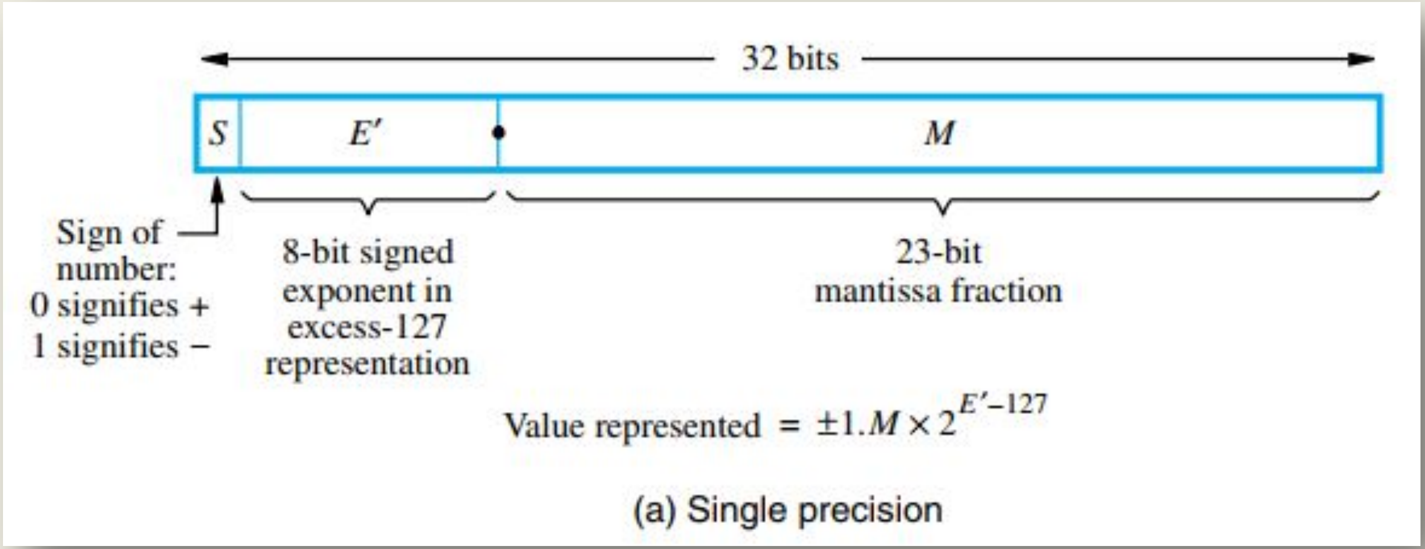
```
Initially   0 0 0 0 0     1 0 0 0   }
            0 0 0 1 1               }
Shift       0 0 0 0 1     0 0 0 _   }  First cycle
Subtract    1 1 1 0 1               }
Set q0      1 1 1 1 0     0 0 0 0   }

Shift       1 1 1 0 0     0 0 0 _   }
Add         0 0 0 1 1               }  Second cycle
Set q0      1 1 1 1 1     0 0 0 0   }

Shift       1 1 1 1 0     0 0 0 _   }
Add         0 0 0 1 1               }  Third cycle
Set q0      0 0 0 0 1     0 0 0 1   }

Shift       0 0 0 1 0     0 0 1 _   }
Subtract    1 1 1 0 1               }  Fourth cycle
Set q0      1 1 1 1 1     0 0 1 0   }
                          Quotient

Add         1 1 1 1 1               }
            0 0 0 1 1               }  Restore remainder
            0 0 0 1 0               }
            Remainder
```

Floating-Point Numbers

- a binary floating-point number can be represented by
  A sign for the number
  • Some significant bits
  • A signed scale factor exponent for an implied base of 2

- IEEE (Institute of Electrical and Electronics Engineers) Standard 754,

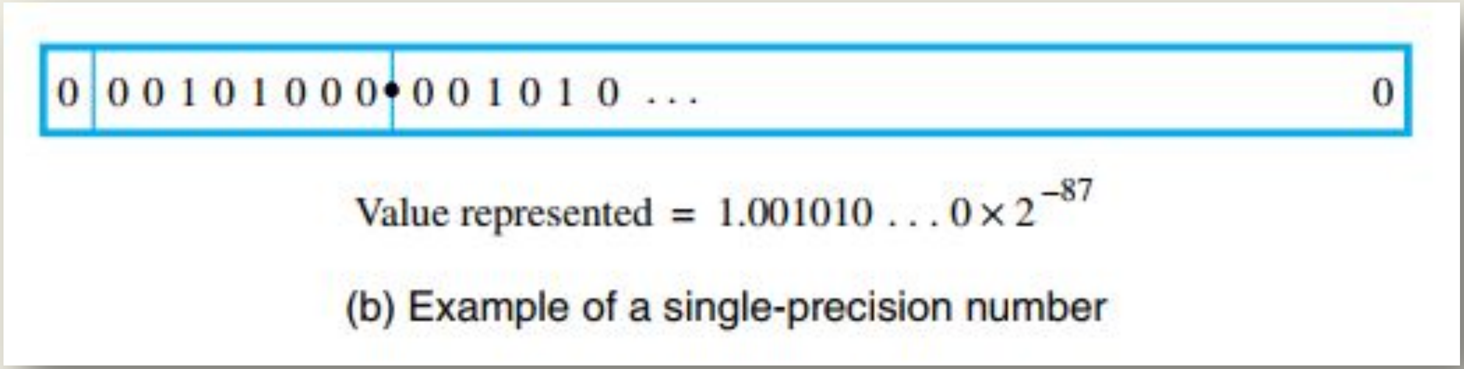
IEEE standard floating-point formats.



32 bits
S
E′
M
Sign of number: 0 signifies + 1 signifies −
8-bit signed exponent in excess-127 representation
23-bit mantissa fraction

Value represented $= \pm 1.M \times 2^{E'-127}$

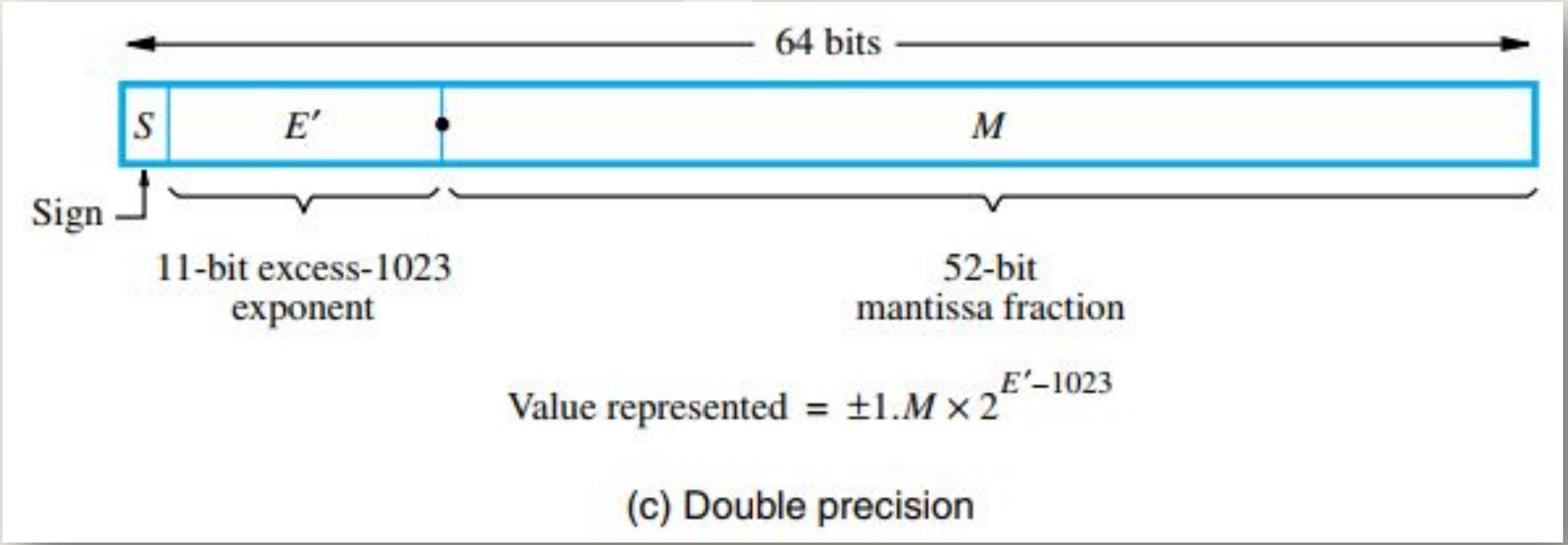
(a) Single precision



Value represented $= 1.001010 \ldots 0 \times 2^{-87}$

(b) Example of a single-precision number

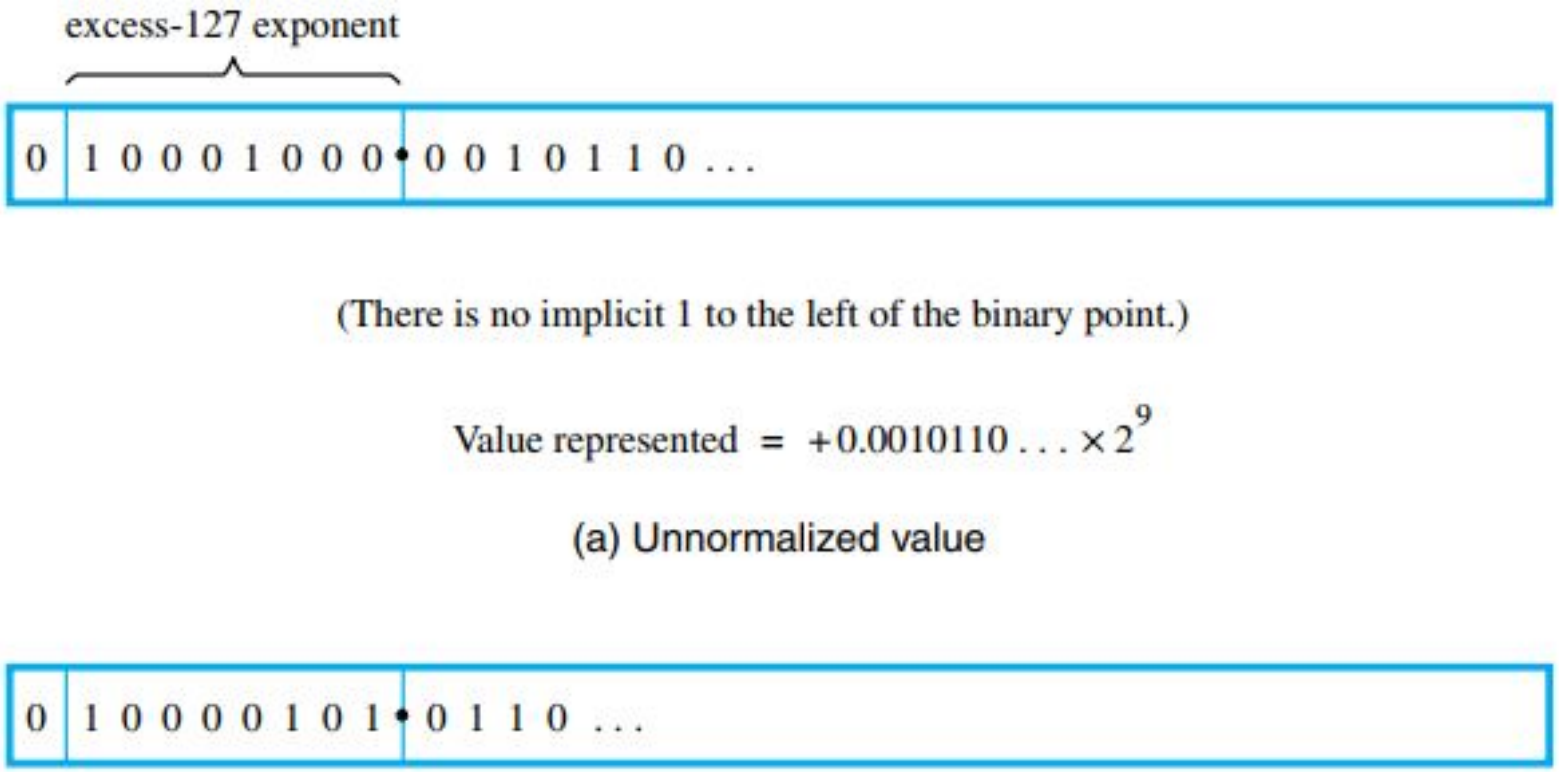
IEEE standard floating-point formats.



Value represented $= \pm 1.M \times 2^{E'-1023}$

(c) Double precision

Floating-point normalization in IEEE single-precision format



(There is no implicit 1 to the left of the binary point.)

$$\text{Value represented} = +0.0010110\ldots \times 2^{9}$$

(a) Unnormalized value

$$\text{Value represented} = +1.0110\ldots \times 2^{6}$$

(b) Normalized version

Objective Type

- The carry generation function: $c_i + 1 = y_i c_i + x_i c_i + x_i y_i$, is implemented in ______________
- a) Half adders
- b) **Full adders**
- c) Ripple adders
- d) Fast adders
-
- Which option is true regarding the carry in the ripple adders? CO 2, PO1
- a) Are generated at the beginning only
- b) **Must travel through the configuration**
- c) Is generated at the end of each operation
- d) None of the mentioned

- In a normal adder circuit, the delay obtained in a generation of the output is ________
- a) **2n + 2**
- b) 2n
- c) n + 2
- d) None of the mentioned

The method used to reduce the maximum number of summands by half is ________
CO 2, PO1

a) Fast multiplication
b) **Bit-pair recording**
c) Quick multiplication
d) None of the mentioned

CSA stands for? CO 2, PO1
a) Computer Speed Addition
b) Carry Save Addition
c) Computer Service Architecture
d) None of the mentioned

In IEEE 32-bit representations, the mantissa of the fraction is said to occupy _______ bits.
CO 2, PO1

a) 24
b) **23**
c) 20
d) 16

In double precision format, the size of the mantissa is _______ CO 2, PO1
a) 32 bit
b) **52 bit**
c) 64 bit
d) 72 bit

The product of 1101 & 1011 is _______ CO 2, PO1

a) **10001111**
b) 10101010
c) 11110000
d) 11001100

We make use of _______ circuits to implement multiplication. CO 2, PO1

a) Flip flops
b) Combinatorial
c) **Fast adders**
d) None of the mentioned

In full adders the sum circuit is implemented using _________ CO 2, PO1

a) And & or gates
b) NAND gate
c) **XOR**
d) XNOR

Short Type

a) Write the equation for sum and carry-out in full adder.

b) Write the corresponding sign-and-magnitude, 1’s complement and 2’s complement representation of following numbers: (a) -10 (b) 8

c) Illustrate the difference between booth’s bit-recording and booth’s bit-pair recording.

d) Find the quotient and remainder using restoring division of the following numbers. 16 ÷ 3

e) Find the quotient and remainder using non-restoring division of the following numbers. 25 ÷ 4

f) Represent the following numbers in single precision of IEEE format. -3.65

g) Represent the following numbers in double precision of IEEE format. +8.25

h) Perform the fast multiplication on the following 2’s complement numbers. 13 X -9

i) Explain with an example the addition/subtraction rule of floating point number.

j) Write the difference between restoring and non-restoring binary division.

Long Type

a) Solve the following operations by using 2's complement. 27-17=?
b) Draw a n-bit ripple adder circuit.
c) What is the number of gate delay for 16- bit carry look ahead adder by using 4-bit adders?
d) Represent -16.25 in single precision and double precision IEEE format.
e) If a 32-bit IEEE format for a floating point number is B0050000 then find its decimal equivalent.
f) Write the steps for multiplication of two floating point number.
g) What are the advantages of Booth algorithm?
h) How does a fast multiplication method reduce number of gate delay?

i) Between restoring and non-restoring division algorithm which perform faster and why?
j) Find the product of the following 2's complement numbers using booth's multiplication.
   (a) -15 X 25
   (b) 32 X -16

Long Type

a) Draw the flowchart for Booth's algorithm for multiplication of signed 2's complement numbers and explain with an example.

b) Explain with a diagram the designs of a fast multiplier using carry save Adder circuit. Give the block diagram for a floating-point adder / subtractor unit and discuss its operation.

c) a) Explain the floating point addition and subtraction
b) State the Non – restoring division technique

d) Explain Division techniques with an example

e) Design an arithmetic circuit with one selection variable S and two n-bit data inputs A & B. The circuit generates the following four arithmetic operations in conjunction with the input carry Cin. Draw the logic diagram for the first two stages.

f) Multiple (-7)10 with (3)10 by using Booth's multiplication. Give the flow table of the multiplication.

g) Explain the Hardware implementation of multiplication in signed-magnitude data? Using this multiply two numbers (Multiplicand is 23 and Multiplier is 19)?