COMPUTER ORGANIZATION

B.TECH III SEM

CSE-4

N SANTOSHI
ASSISTANT PROFESSOR
DEPARTMENT OF ECE

NUMBER SYSTEMS

- DECIMAL NUMBER SYSTEM
  - 0,1,2,3,4,5,6,7,8,9
- BINARY NUMBER SYSTEM
  - 0,1
- OCTAL NUMBER SYSTEM
  - 0,1,2,3,4,5,6,7
- HEXADECIMAL NUMBER SYSTEM
  - 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Complements

- There are two types of complements for each base r system: the r's complement and the (r - 1)'s complement.
- When the value of the base r is substituted in the name, the two types are referred to as the 2's and 1's complement for binary numbers and the 10's and 9's complement for decimal numbers.

Subtraction of Unsigned Numbers

- Unsigned 2's Complement Subtraction Example
- Find $01010100_2 - 01000011_2$

$$\begin{array}{r} 01010100 \\ -\ \underline{01000011} \end{array} \xrightarrow{\text{2's comp}} \begin{array}{r} {}^{1}\,01010100 \\ +\ \underline{10111101} \\ 00010001 \end{array}$$

The carry of 1 indicates that no correction of the result is required.

Fixed-Point Representation

- Positive integers, including zero, can be represented as unsigned numbers.
- To represent negative integers, we need a notation for negative values.
- In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign.
- Because of hardware limitations, computers must represent everything with 1's and 0's, including the sign of a number.
- sign bit equal to 0 for positive and to 1 for negative.

- In addition to the sign, a number may have a binary (or decimal) point.
- The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers.
- There are two ways of specifying the position of the binary point in a register:
  1. by giving it a fixed position or
  2. by employing a floating-point representation.

- The two positions most widely used are
  1. binary point in the extreme left of the register to make the stored number a fraction
  2. binary point in the extreme right of the register to make the stored number an integer.

Integer Representation

- When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:
  1. Signed-magnitude representation
  2. Signed-1' s complement representation
  3. Signed 2' s complement representation

Signed numbers

Signed Binary Numbers

Decimal	Signed 2's Complement	Signed 1's Complement	Signed Magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	—	1111	1000
−1	1111	1110	1001
−2	1110	1101	1010
−3	1101	1100	1011
−4	1100	1011	1100
−5	1011	1010	1101
−6	1010	1001	1110
−7	1001	1000	1111
−8	1000	—	—

- The signed-magnitude representation of a negative number consists of the magnitude and a negative sign.
- The negative number is represented in either the 1's or 2's complement of its positive value.
- For example, consider the signed number 14 stored in an 8-bit register.
- + 14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14: 00001110.

- Although there is only one way to represent + 14, there are three different ways to represent - 14 with eight bits.
- In signed-magnitude representation 1 0001110
- In signed-1's complement representation 1 1110001
- In signed-2's complement representation 1 11 10010

- The 1' s complement imposes difficulties because it has two representations of 0   (+ 0 and - 0).

Arithmetic Addition

- The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign.
- If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude.
- For example, $(+25) + (-37) = -(37 - 25) = -12$ and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result.
- This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction.

2's complement addition

- the rule for adding numbers in the signed-2's complement system does not require a comparison or subtraction, only addition and complementation.
- The procedure is very simple and can be stated as follows: Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position.
- Note that negative numbers must initially be in 2' s complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

Example

$$
\begin{array}{rr}
+6 & 00000110 \\
+13 & 00001101 \\
\hline
+19 & 00010011
\end{array}
\qquad
\begin{array}{rr}
-6 & 11111010 \\
+13 & 00001101 \\
\hline
+7 & 00000111
\end{array}
$$

$$
\begin{array}{rr}
+6 & 00000110 \\
-13 & 11110011 \\
\hline
-7 & 11111001
\end{array}
\qquad
\begin{array}{rr}
-6 & 11111010 \\
-13 & 11110011 \\
\hline
-19 & 11101101
\end{array}
$$

In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2' s complement form.

- To determine the value of a negative number when in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form.
- For example, the signed binary number 1111 1001 is negative because the leftmost bit is 1. Its 2' s complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to - 7 .

Arithmetic Subtraction

- Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.

Overflow

- When two numbers of n digits each are added and the sum occupies n+1 digits, we say that an overflow occurred.
- An overflow is a problem in digital computers because the width of registers is finite.
- A result that contains n+1 bits cannot be accommodated in a register with a standard length of n bits.
- For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

example

carries:	0 1	carries:	1 0
+70	0 1000110	−70	1 0111010
+80	0 1010000	−80	1 0110000
+150	1 0010110	−150	0 1101010

Note that the 8-bit result that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9-bit answer so obtained will be correct. Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred.

Decimal Fixed-Point Representation

- The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit.
- A 4-bit decimal code requires four flip-flops for each decimal digit.
- The representation of 4385 in BCD requires 16 flip-flops, four flip-flops for each digit. The number will be represented in a register with 16 flip-flops as follows:

0100 0011 1000 0101

- By representing numbers in decimal we are wasting a considerable amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its equivalent binary representation.
- However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system.
- However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system.

- For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion to binary and back to decimal.
- The sign of a decimal number is usually represented with four bits to conform with the 4-bit code of the decimal digits. It is customary to designate a plus with four 0' s and a minus with the BCD equivalent of 9, which is 1001 .

Consider the addition (+375) + (-240) = + 135 done in the signed- 10's complement system.

$$\begin{array}{rl} 0\ 375 & (0000\ 0011\ 0111\ 0101)_{BCD} \\ +\underline{9\ 760} & \underline{(1001\ 0111\ 0110\ 0000)_{BCD}} \\ 0\ 135 & (0000\ 0001\ 0011\ 0101)_{BCD} \end{array}$$

- The 9 in the leftmost position of the second number indicates that the number is negative. 9760 is the 10's complement of 0240.
- The two numbers are added and the end carry is discarded to obtain +135.

Floating-Point Representation

- The floating-point representation of a number has two parts.
- The first part represents a signed, fixed-point number called the mantissa. The second part designates the position of the decimal (or binary) point and is called the exponent.
- The fixed-point mantissa may be a fraction or an integer.
- For example, the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:
- Example

Fraction	*Exponent*
+0.6132789	+04

- This representation is equivalent to the scientific notation +0. 6132789 X 10^{+4}.
- Floating-point is always interpreted to represent a number in the following form:

$$m \times r^e$$

- Only the mantissa m and the exponent e are physically represented in the register (including their signs).

- A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number +1001. 11 is represented with a n 8-bit fraction and 6-bit exponent a s follows:

Fraction	*Exponent*
01001110	000100

- The fraction has a 0 in the leftmost position to denote positive. The binary point of the fraction follows the sign bit but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$m \times 2^e = +(.1001110)_2 \times 2^{+4}$$

Normalisation

- **Normalisation** is the process of moving the binary **point** so that the first digit after the **point** is a significant digit.
- A floating-point number is said to be normalized if the most significant digit of the mantissa is nonzero.
- For example, the decimal number 350 is normalized but 00035 is not.

- For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0' s. The number can be normalized by shifting it three positions to the left and discarding the leading O's to obtain 11010000.
- The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be subtracted by 3 .

Alphanumeric representation

- Many applications of digital computers require the handling of data that consist not only of numbers, but also of the letters of the alphabet and certain special characters.
- An alphanumeric character set is a set of elements that includes the 10 decimal digits, the 26 letters of the alphabet and a number of special characters, such as $, + , and = .

- The standard alphanumeric binary code is the ASCII (American Standard Code for Information Interchange), which uses seven bits to code 128 characters.
- Binary codes play an important part in digital computer operations.
- The codes must be in binary because registers can only hold binary information.

ASCII CODES

Character	Binary code	Character	Binary code
A	100 0001	0	011 0000
B	100 0010	1	011 0001
C	100 0011	2	011 0010
D	100 0100	3	011 0011
E	100 0101	4	011 0100
F	100 0110	5	011 0101
G	100 0111	6	011 0110
H	100 1000	7	011 0111
I	100 1001	8	011 1000
J	100 1010	9	011 1001
K	100 1011		
L	100 1100		
M	100 1101	space	010 0000
N	100 1110	.	010 1110
O	100 1111	(	010 1000
P	101 0000	+	010 1011
Q	101 0001	$	010 0100
R	101 0010	*	010 1010
S	101 0011	)	010 1001
T	101 0100	−	010 1101
U	101 0101	/	010 1111
V	101 0110	,	010 1100
W	101 0111	=	011 1101
X	101 1000		
Y	101 1001		
Z	101 1010		

- The ASCII code is the standard code commonly used for the transmission of binary information.
- Each character is represented by a 7-bit code and usually an eighth bit is inserted for parity.
- The code consists of 128 characters.

EBCDIC

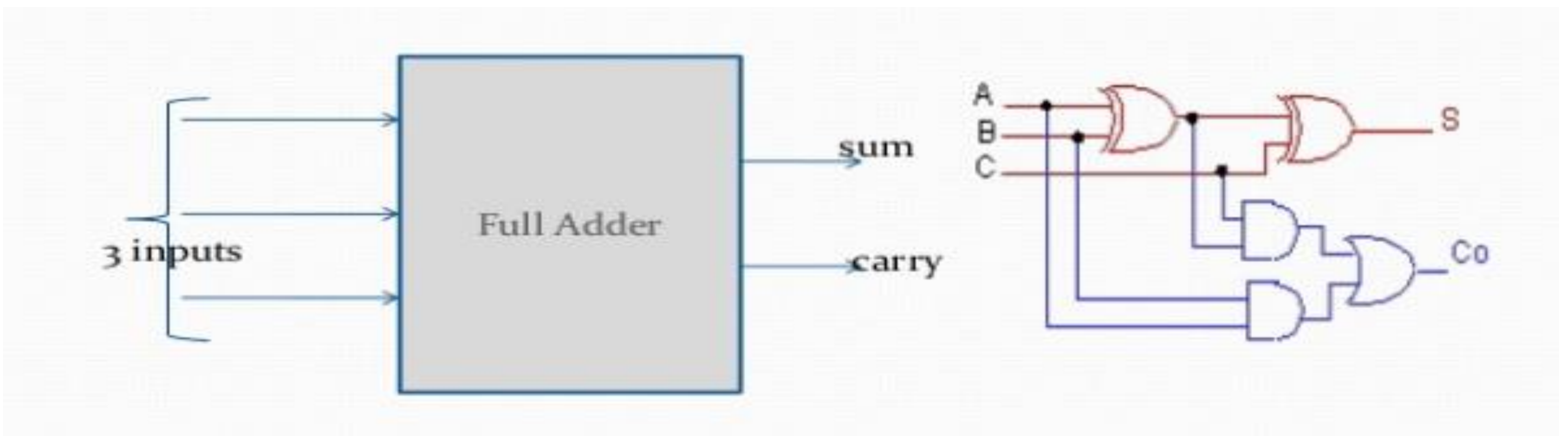
- Another alphanumeric (sometimes called alphameric) code used in IBM equipment is the EBCDIC (Extended BCD Interchange Code).
- It uses eight bits for each character (and a ninth bit for parity). EBCDIC has the same character symbols as ASCII but the bit assignment to characters is different.

Adders

- Binary addition is a fundamental operation in most digital circuits.
- There are a variety of adders, each has certain performance.
- Each type of adder is selected depending on where the adder is to be used.
- Ripple carry adder is suitable for small bit applications.

Full adder

- combinational circuit that adds two bits is called a half adder.
- A full adder is one that adds three bits Full Adder sum 3 inputs carry.



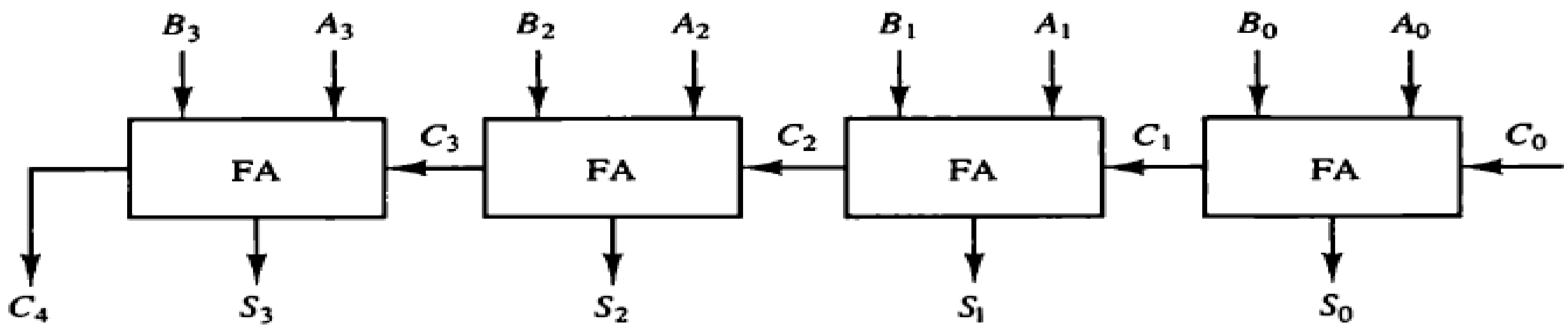
Truth table

Input A	Input B	Input C	sum	carry
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

N-bit binary adder/ripple carry adder

- The ripple carry adder is constructed by cascading full adder blocks in series.
- The carryout of one stage is fed directly to the carry-in of the next stage .
- For an n-bit ripple adder, it requires n full adders/n-1 full adders and 1 half adder.

Block diagram of ripple carry adder



- To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder.
- The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a binary adder.

- The binary adder is constructed with full-adder circuits connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder.
- The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C_0 and the output carry is C_4. The S outputs of the full-adders generate the required sum bits.

- An n-bit binary adder requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order full-adder.

Advantages

- We can add two n-bit numbers easily.
- It is advantageous for less number of bit operations.

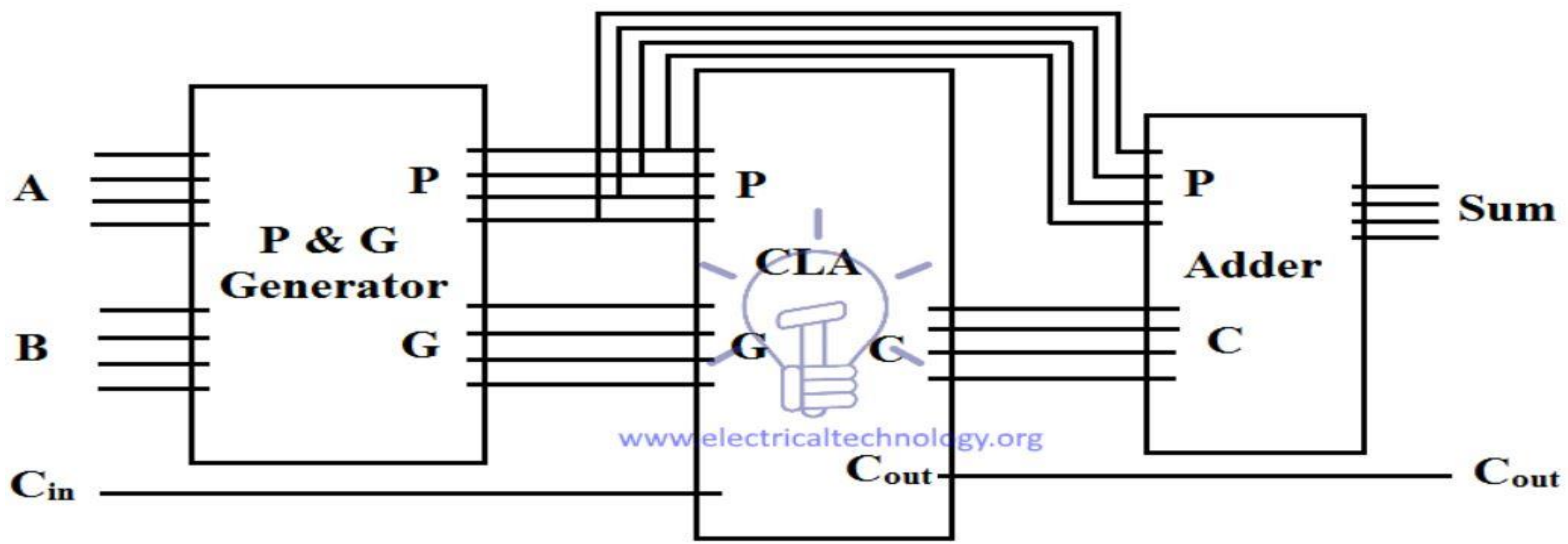
Disadvantages

- **Ripple-carry adder**, illustrating the delay of the **carry** bit.
- The **disadvantage** of the **ripple-carry adder** is that it can get very slow when one needs to add many bits.
- To reduce the computation time, there are faster ways to add two binary numbers by using **carry** look ahead **adders**.

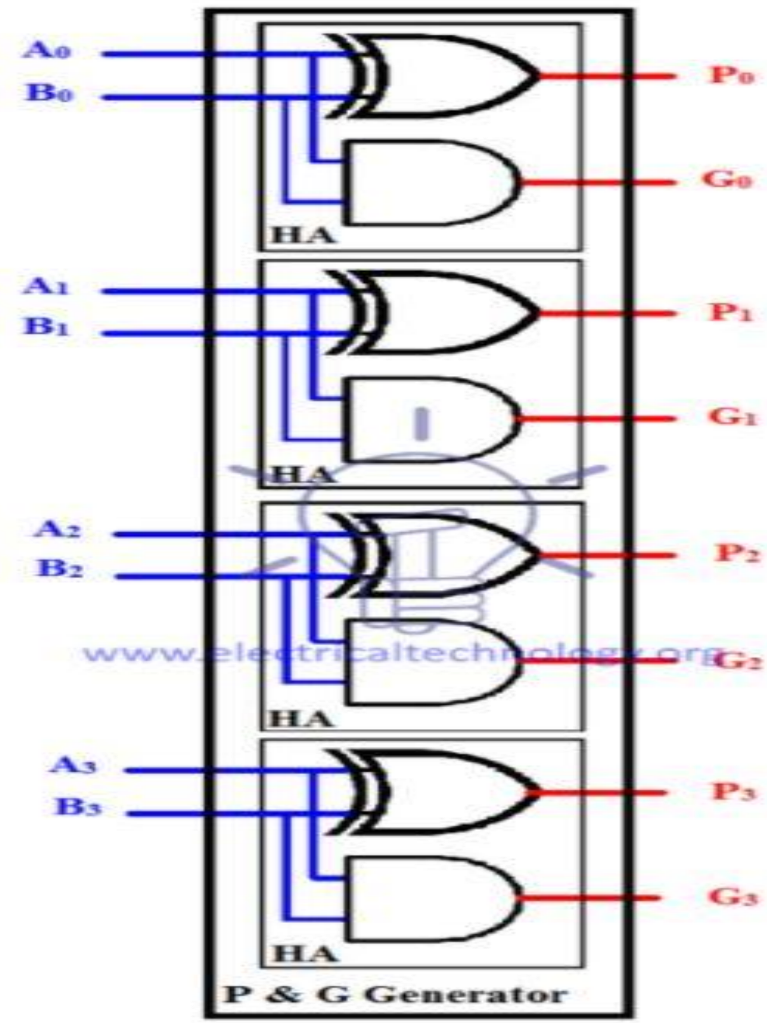
Carry Look Ahead Adder (CLA)

- **Carry look ahead adder’s (CLA) logic diagram** is given below. It contains **3 blocks**; “**P and G generator**”, “**Carrylook ahead**” block and “**adder block**
- Input “**Augend**”, “**Addend**” is provided to the “P and G generator” block whose output is connected with **CLA** and the **adder block**.

- **Carry = AB + C_{in} (A XOR B)**
- **P = (A XOR B)**: **P** is known as **Carry propagate**, because it propagates the **C_{in}** from previous stage to the next stage.
- **G = AB: G** is known as **Carry Generate**, because it can directly generate carry bit without any C_{in}.



Carry Lookahead (CLA) Block Diagram



- Let us consider a full adder. We have the inputs signals A, B, and Cin. If we consider the addition of these three variables in every possible case, we get a truth table like the one below.

A	B	C_{in}	Sum	Carry	Condition
0	0	0	0	0	No Carry Generated
0	0	1	1	0	
0	1	0	1	0	
0	1	1	0	1	Carry Propagated
1	0	0	1	0	No Carry
1	0	1	0	1	Carry Propagated
1	1	0	0	1	Carry Generated
1	1	1	1	1	

- On analyzing the truth table, we see that the Carry is 1 when
  1. Either the value of A or B is one, as well as Cin, is 1, or
  2. Both A and B have the value 1.

- Let us now consider two new variables, **Carry Generate (Gi)** and **Carry Propagate (Pi)**.
- **Case1:** we see that an output carry is propagated, when we give an input carry. We will refer to this with Pi. So, the mathematical expression of Pi can we represented as :

$$Pi = Ai \oplus Bi$$

- **case 2:** we see that an output carry is generated when both inputs, A and B, are high, regardless of the value of the input carry. We will refer to this output carry as Gi. Thus, we can mathematically express Gi as :

$$G_i = A_i \cdot B_i$$

- Originally, for a full adder we have the following equations:

$$Sum = A \oplus B \oplus Ci$$

$$Carry = Ci(A+B) + AB$$

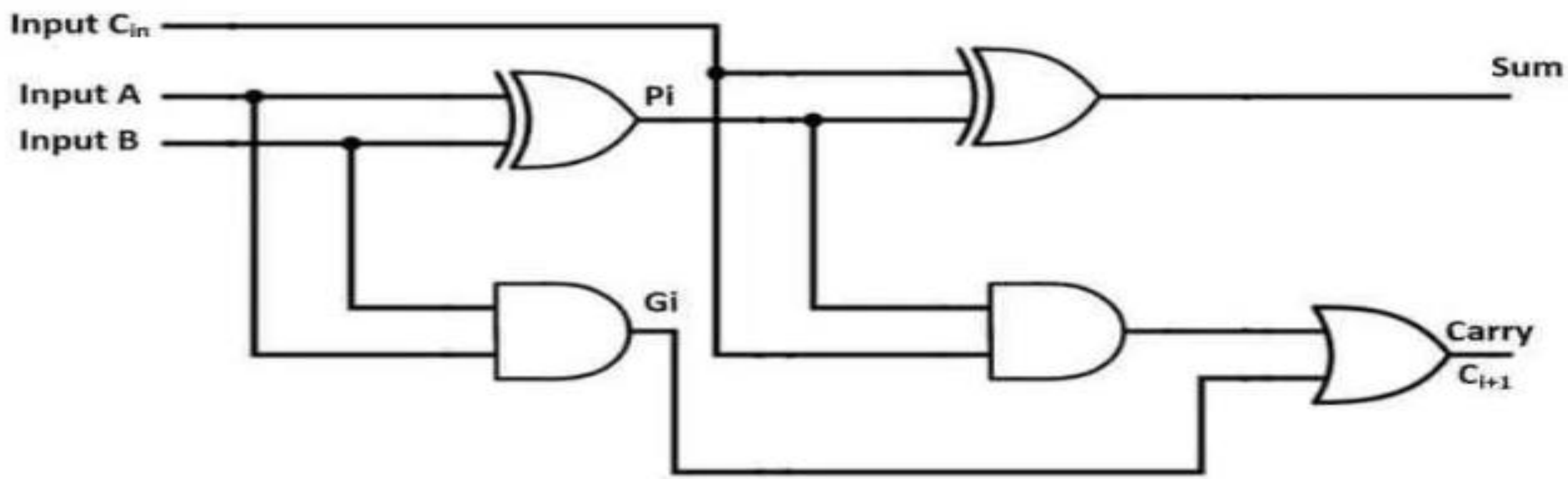
- Or $carry_{i+1}=AB+BC_i+AC_i$

- Thus, we can rewrite the equations of the full adder in terms of Carry Propagate (Pi) and Carry Generate (Gi) as :

$$Sum = Pi \oplus Ci$$

$$Carry = Gi + Pi \cdot Ci$$

The equations of Sum and Carry can be represented by a logic circuit given below.



Logic Circuit

We can calculate the output carry C1, C2, C3, and C4 using the above derived equations as:

$C1 = \underline{(C_{in} . P0) + G0}$

$C2 = (C1 . P1) + G1 = (((C_{in} . P0) + G0) . P1) + G1$

$= \underline{(C_{in} . P0 . P1) + (G0 . P1) + G1}$

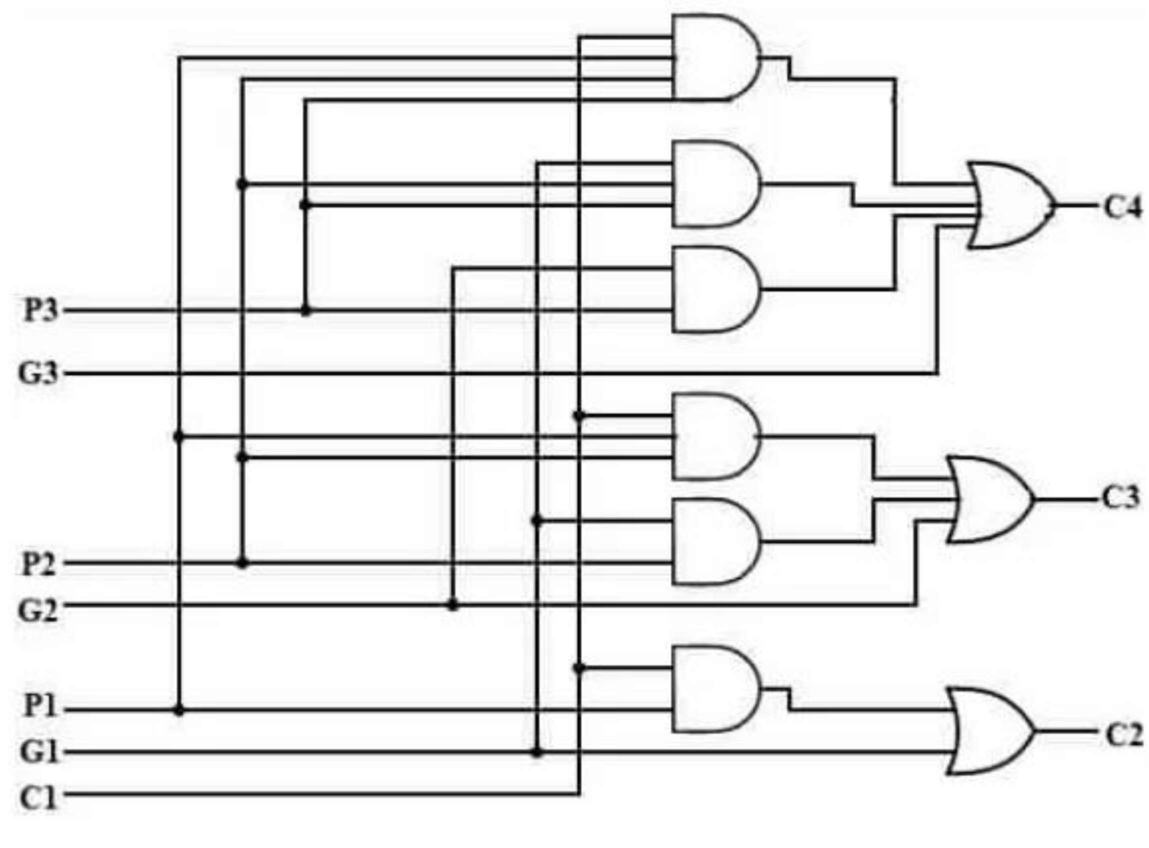
$C3 = (C2 . P2) + G2 = (((C1 . P1) + G1) . P2) + G2$

$= \underline{G2 + (P2 . G1) + (P2 . P1 . G0) + (P2 . P1 . P0 . C_{in})}$

$C4 = (C3 . P3) + G3$

$= \underline{(C_{in} . P0 . P1 . P2 . P3) + (P3 . P2 . P1 . G0) + (P3 . P2 . G1) + (G2 . P3) + G3}$

Circuit Diagram of 4-bit Carry-Lookahead Adder



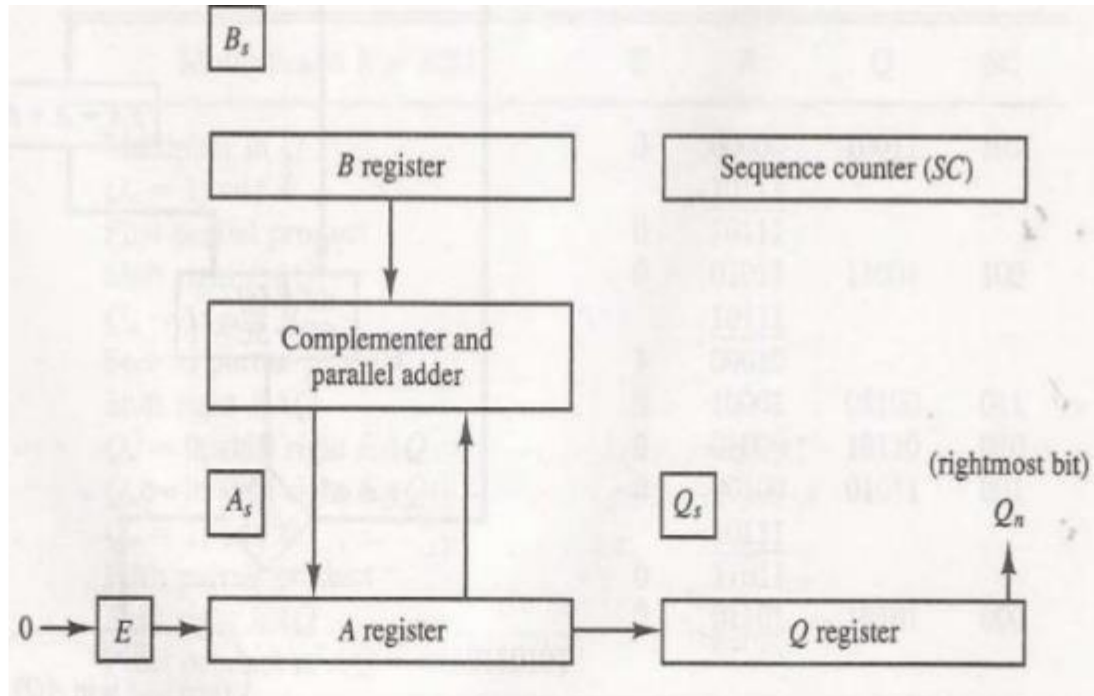
Multiplication – shift-and add

- A multiplication algorithm is an algorithm (or method)to multiply two numbers. Depending on the size of the numbers,different algorithms are in use. Efficient multiplicationalgorithms have existed since the advent of the decimal system.

23	10111		Multiplicand
19	× 10011		Multiplier
	10111		
	10111		
	00000	+	
	00000		
	10111		
437	110110101		Product

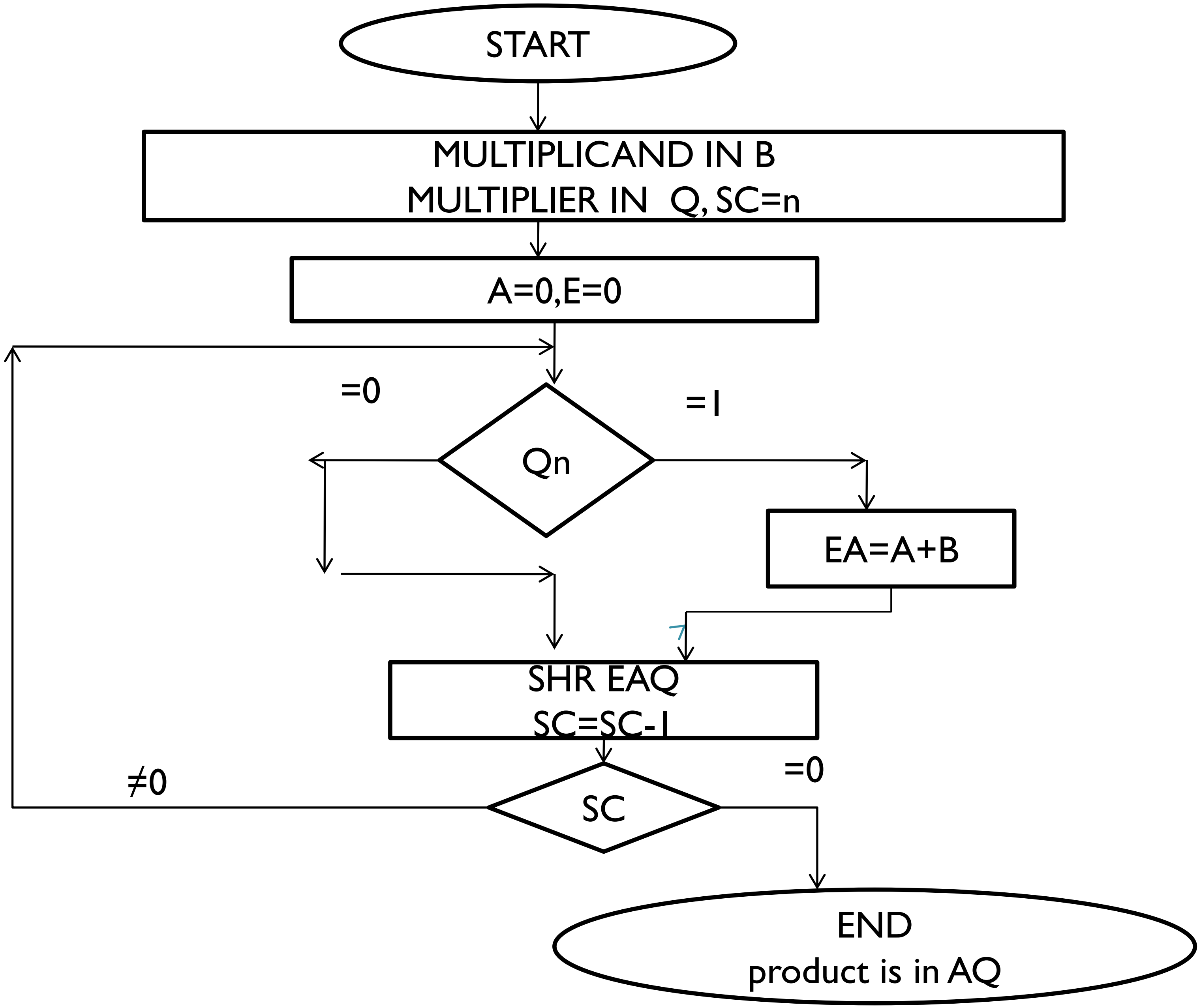
- Instead of as many number of registers as there are bits in multiplier, it is convenient to provide an adder for the summation of only two successive binary numbers.
- Instead of shifting the multiplicand to the left , the partial product will be shifted to the right.
- when the corresponding bit of multiplier is 0, there is no need to add all zeros to the partial product.
- Eg: 10011
- X 11
- ------------------------------
- 111001

Hardware implementation



- The multiplicand is in register B and multiplier is in Q. The SC is initially set a number equal to the number of bits in multiplier.
- The counter is decremented by 1 after forming each partial product.
- The sum of A and B forms a partial product which is transferred to the EA register.
- Both the partial product and multiplier are shifted to the right. shrEAQ.
- The LSB of A is shifted into MSB of Q, The bit from E is shifted into MSB of A, and 0 is shifted into E.
- In this manner the right most bit of the multiplier will be the one which must be inspected next.

FLOW CHART


START
MULTIPLICAND IN B
MULTIPLIER IN Q, SC=n
A=0,E=0
=0
=1
Qn
EA=A+B
SHR EAQ
SC=SC-1
≠0
=0
SC
END
product is in AQ

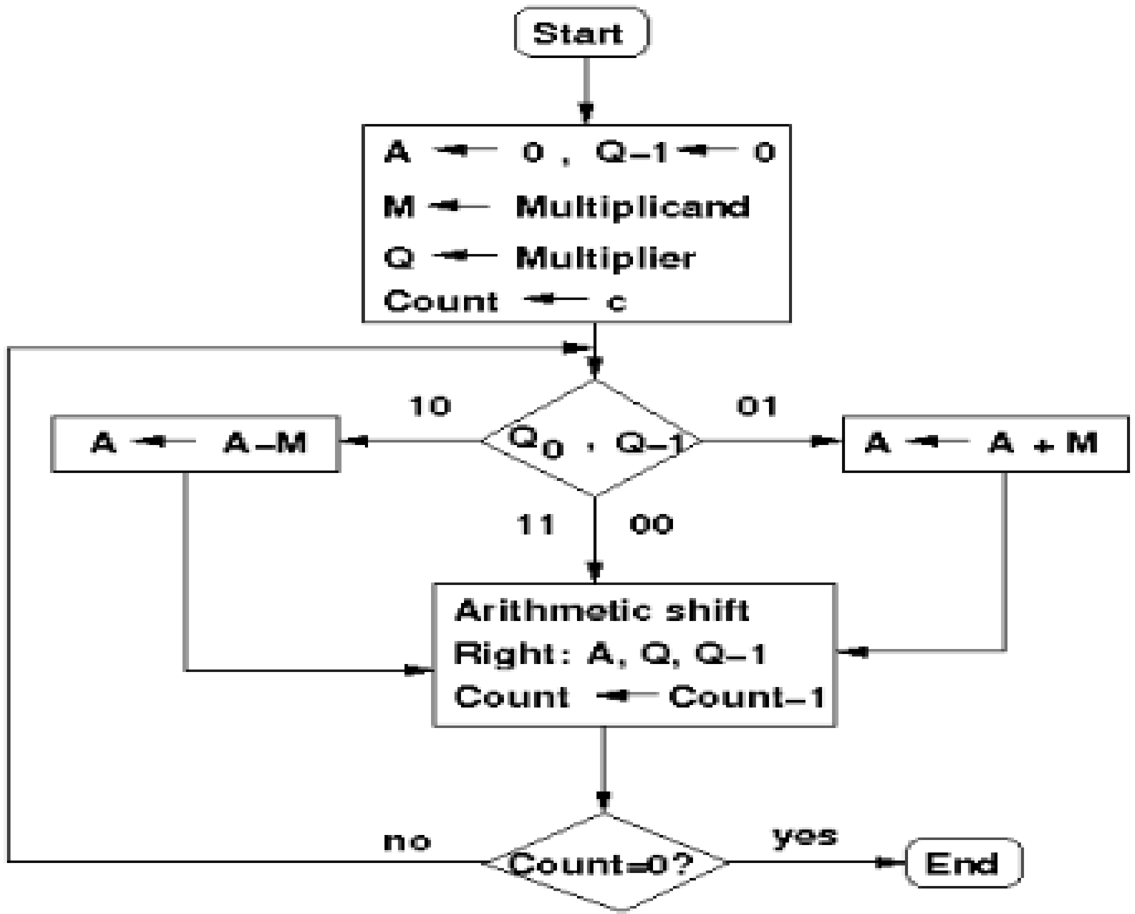
Numeric example for binary multiplier

Multiplicand $B = 10111$	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in $AQ = 0110110101$				

Booth’s multiplier

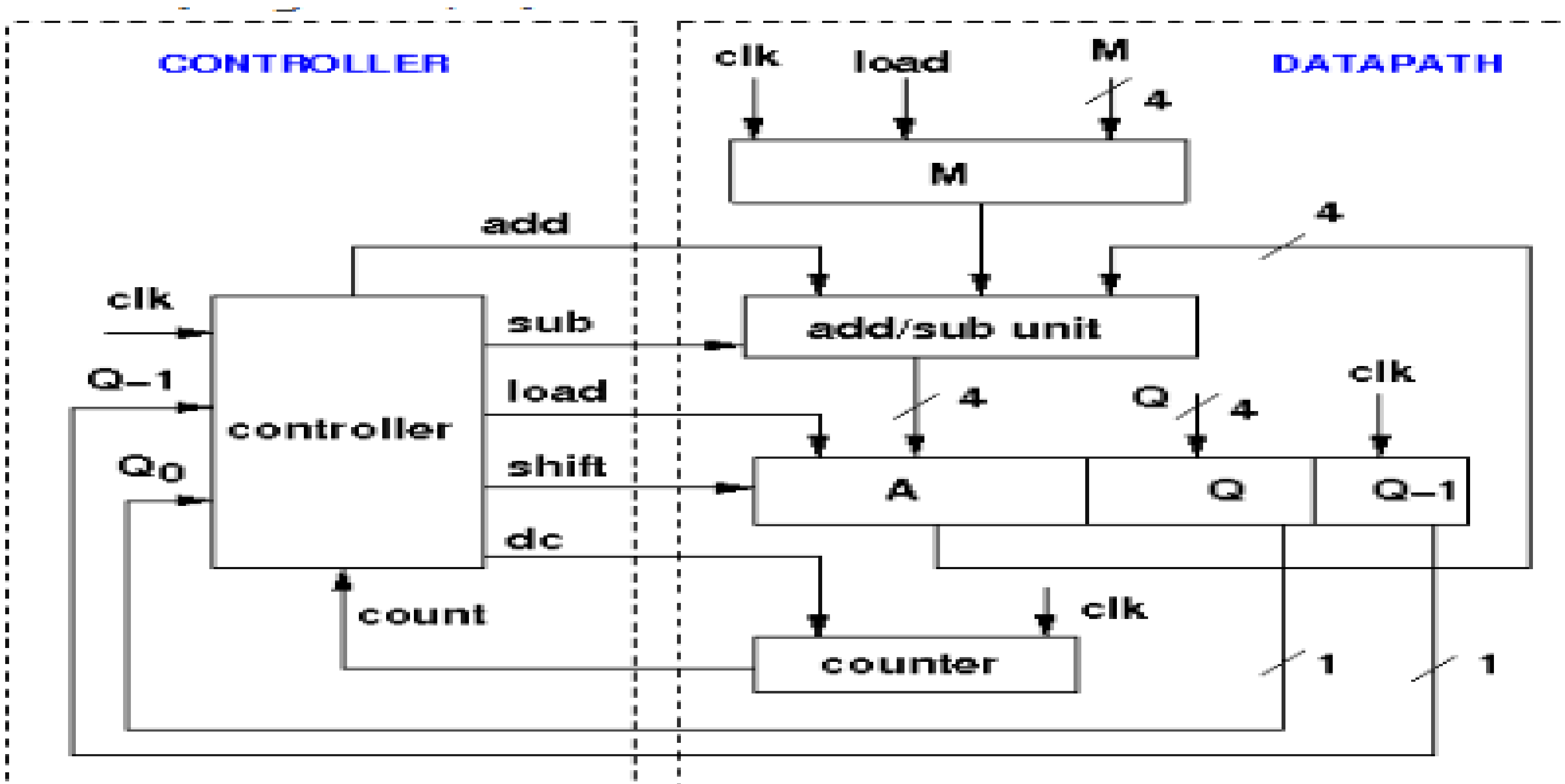
- Booth's multiplication algorithm is an algorithm which multiplies 2 signed integers in 2's complement.
- The multiplicand and multiplier are placed in the m and Q registers respectively.
- A 1 bit register is placed logically to the right of the LSB (least significant bit) Q0 of Q register. This is denoted by Q-1.

Flow chart



Start
A ← 0 , Q–1 ← 0
M ← Multiplicand
Q ← Multiplier
Count ← c
10
01
Q0 , Q–1
A ← A–M
A ← A + M
11
00
Arithmetic shift
Right: A, Q, Q–1
Count ← Count–1
no
yes
Count=0?
End

Hardware implementation



Set M as 0111 which is 7 and Q as 0101 which is 5. Now start the multiplication operation and observe the results including the intermediate results.

A	Q	Q-1	M		
0000	0101	0	0111	Initial value	
1001	0101	0	0111	A ← A-M	First cycle
1100	1010	1	0111	shift	
0011	1010	1	0111	A ← A+M	Second cycle
0001	1101	0	0111	shift	
1010	1101	0	0111	A ← A-M	Third cycle
1101	0110	1	0111	shift	
0100	0110	1	0111	A ← A+M	Fourth cycle
0010	0011	0	0111	shift	

Multiplier Q=7=0111 and multiplicand M= -6 = 1010(as negetive results are automatically in 2's complement form). [A flip-flop (a fictitious bit position)is used to the right of lsb of the multiplier and it is initialized to 0]

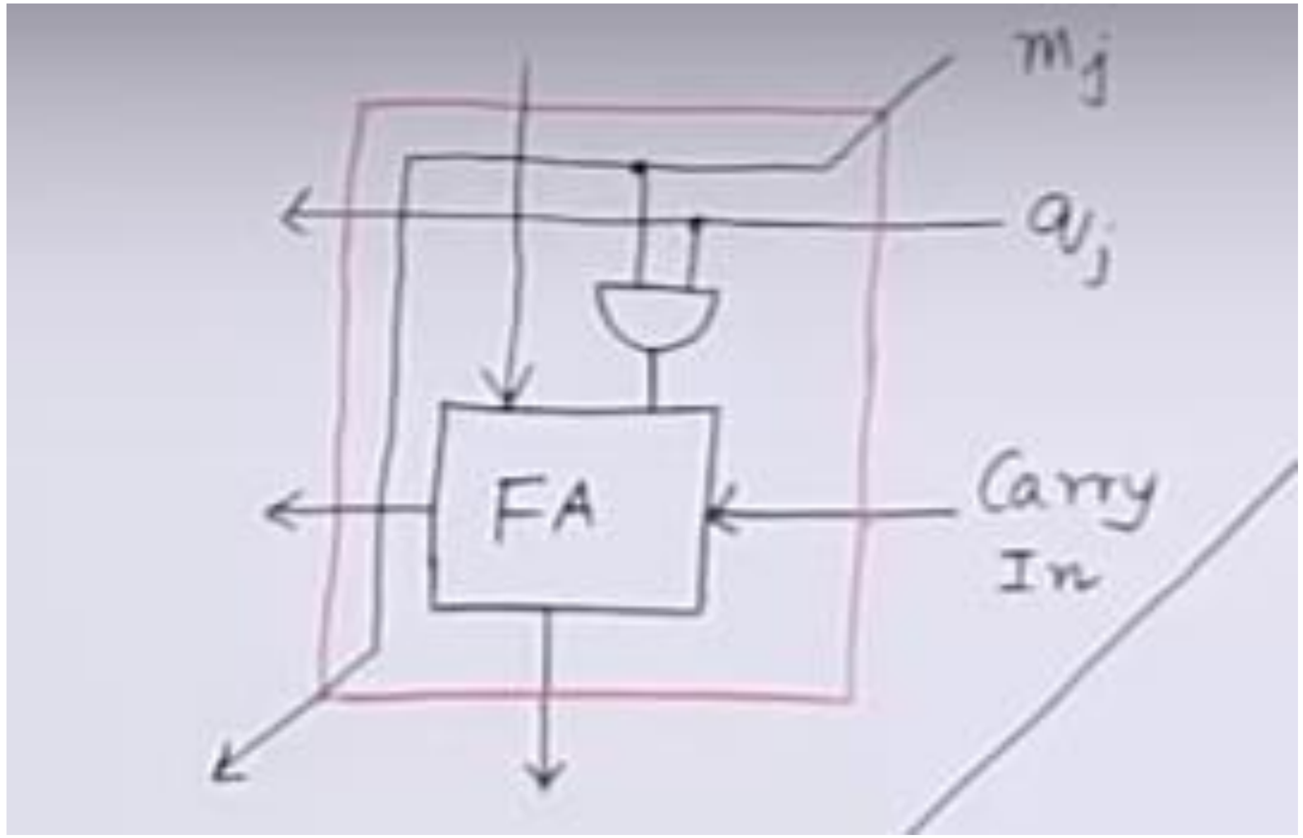
INTIAL VALUES	M 1010	A 0000	Q 0111	Q-1 0
A=A-M SHIFT	1010 1010	0110 0011	0111 0011	0 1
SHIFT	1010	0001	1001	1
SHIFT	1010	0000	1100	1
A=A+M SHIFT	1010 1010	1010 1101	1100 0110	1 0

- ... the number in familiar form we take 2's complement of magnitude. **The result is -42 .**

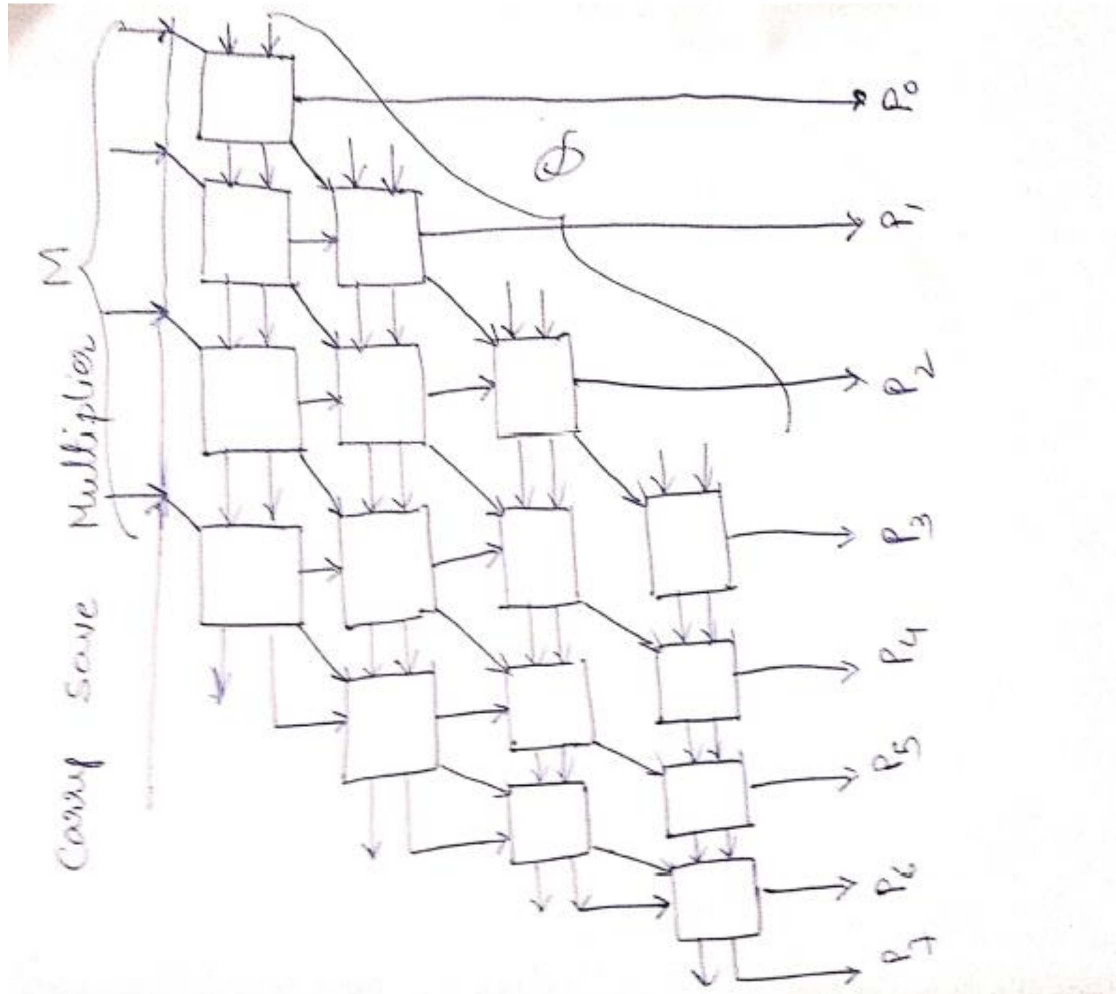
- It can handle signed integers in 2's complement notion
- It decreases the number of addition and subtraction
- It requires less hardware than combinational multiplier
- It is faster than straightforward sequential multiplier

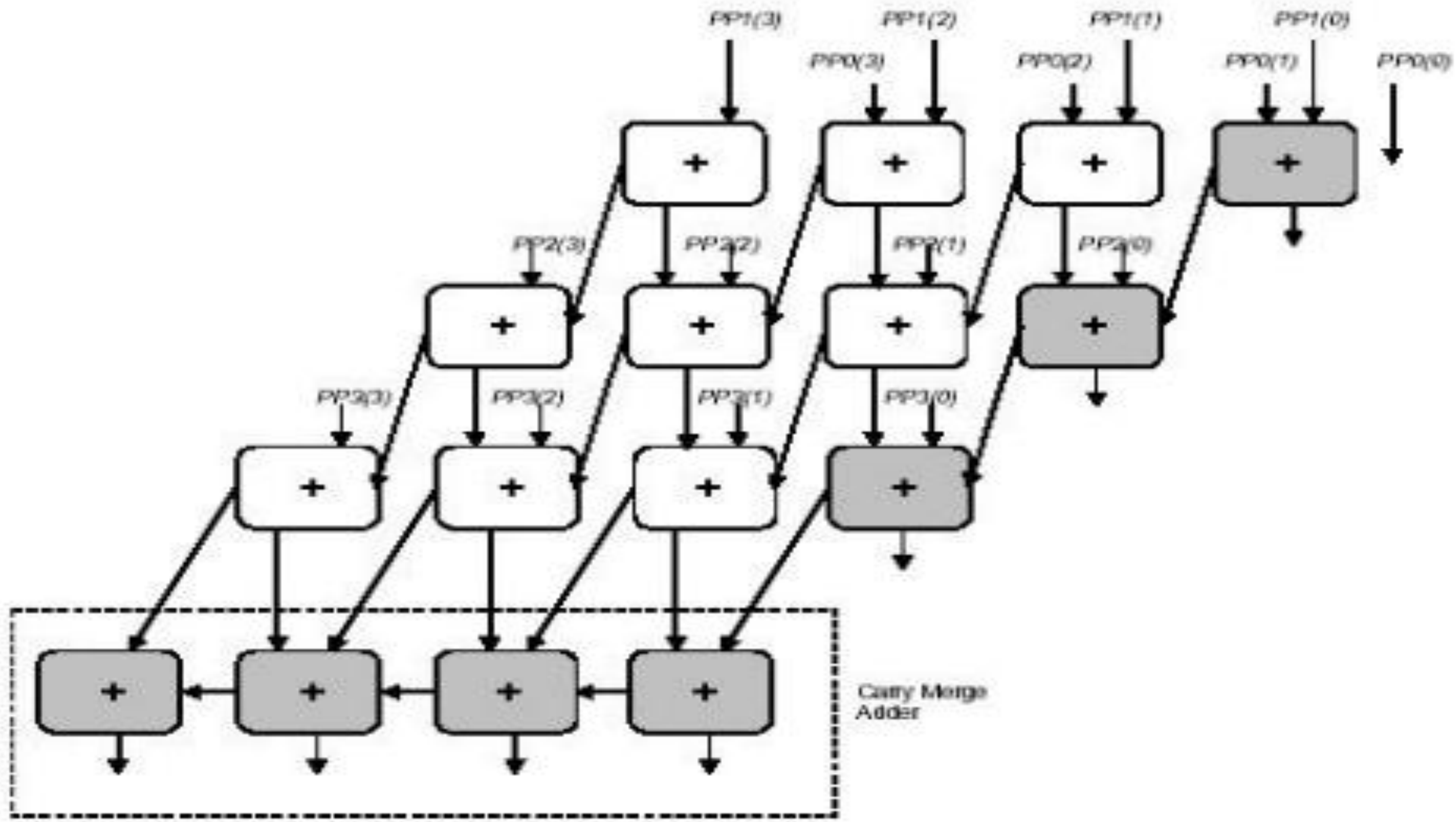
Carry save multiplier

1110	Multiplicand
1010	Multiplier
0000	Partial product
11110	Partial product
10000	Partial product
1110	Partial product
10001100	Final product



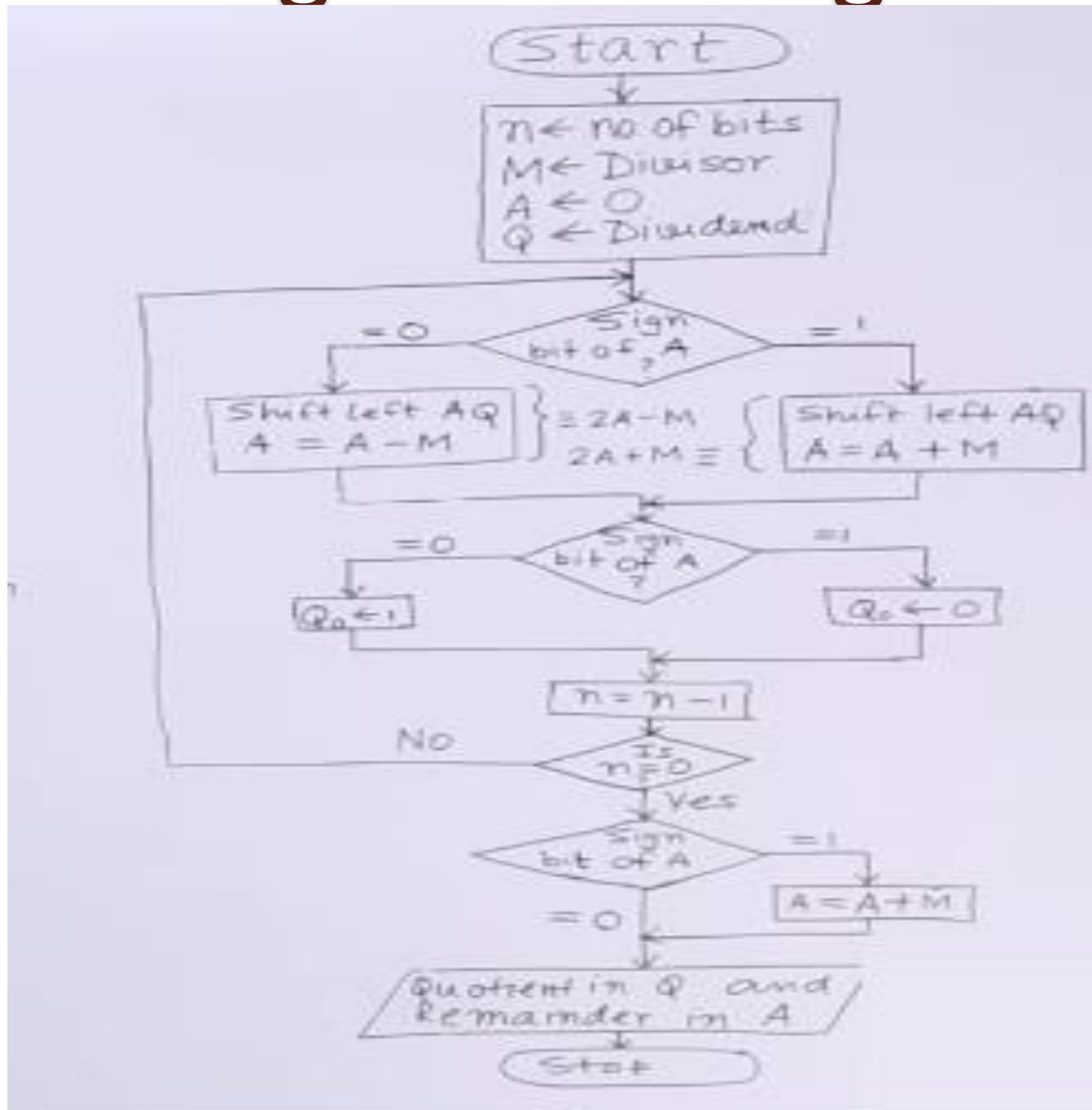
m_j
a_j
FA
Carry In


Carry Save Multiplier
M
P0
P1
P2
P3
P4
P5
P6
P7



PP1(3)
PP1(2)
PP1(1)
PP1(0)
PP0(3)
PP0(2)
PP0(1)
PP0(0)
PP2(3)
PP2(2)
PP2(1)
PP2(0)
PP3(3)
PP3(2)
PP3(1)
PP3(0)
Carry Merge Adder

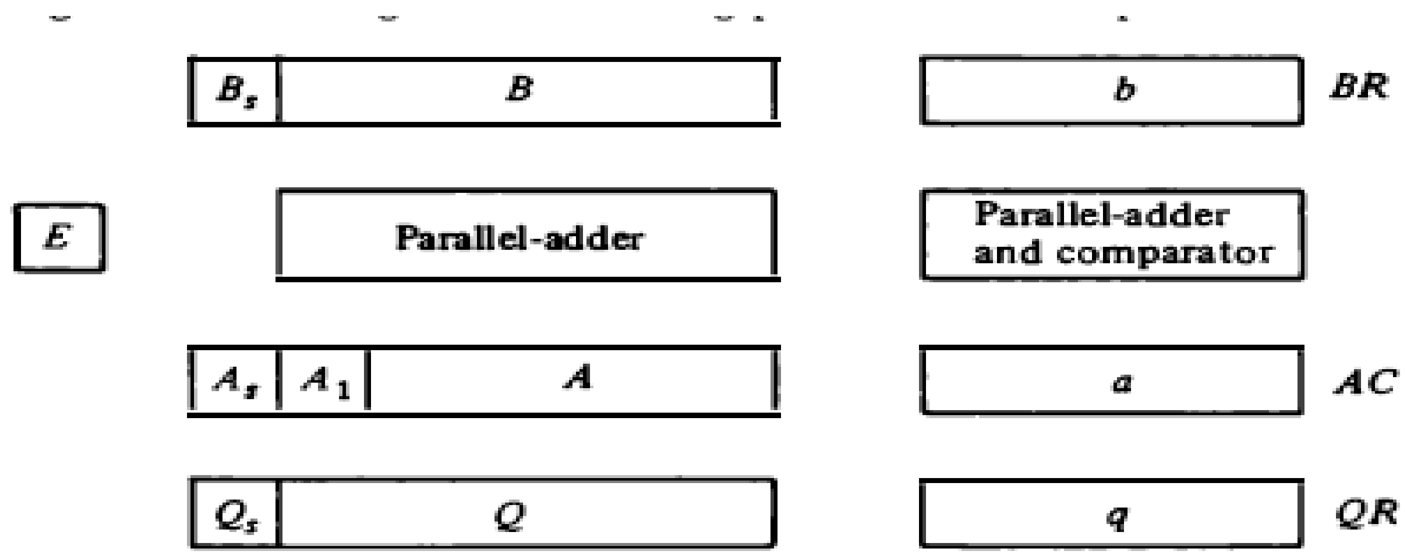
Non restoring division Algorithm



11(dividend)/3(divisor)=3(quotient) and 2(remainder)
-M=11101

N	M	A	Q	Operation
4	00011	00000	1011	intialization
		00001	011?	SL AQ
		11110	011?	A=A-M
3		11110	0110	$Q_0 \leftarrow 0$
		11100	110?	SL AQ
		11111	110?	A=A+M
2		11111	1100	$Q_0 \leftarrow 0$
		11111	100?	SL AQ
		00010	100?	A=A+M
1		00010	1001	$Q_0 \leftarrow 1$
		00101	001?	SL AQ
		00010	001?	A=A-M
0		00010	0011	$Q_0 \leftarrow 1$

Floating-Point Arithmetic Operations



- There are three registers, BR, AC , and QR.
- Each register is subdivided into two parts.
- The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part uses the corresponding lowercase letter symbol.
- each floating-point number has a mantissa in signed magnitude representation and a biased exponent.

- Thus the AC has a mantissa whose sign is in A, and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol a. most significant bit of A, labeled by A1 •
- Similarly, register BR i s subdivided into B_S, B, and b, and QR into
- A parallel-adder adds the two mantissas and transfers the sum into A and the carry into E. A separate parallel-adder is used for the exponents. Q_S, Q , and q.

Floating point Addition and Subtraction

- In floating-point arithmetic:
- two floating-point operands are in AC and BR .
- The sum or difference is formed in the AC .
- There are four basic phases of the algorithm for addition and subtraction:
    1. Check for zeros
    2. Align the mantissas
    3. Add or subtract the mantissas
    4. Normalize the result.

Flow chart

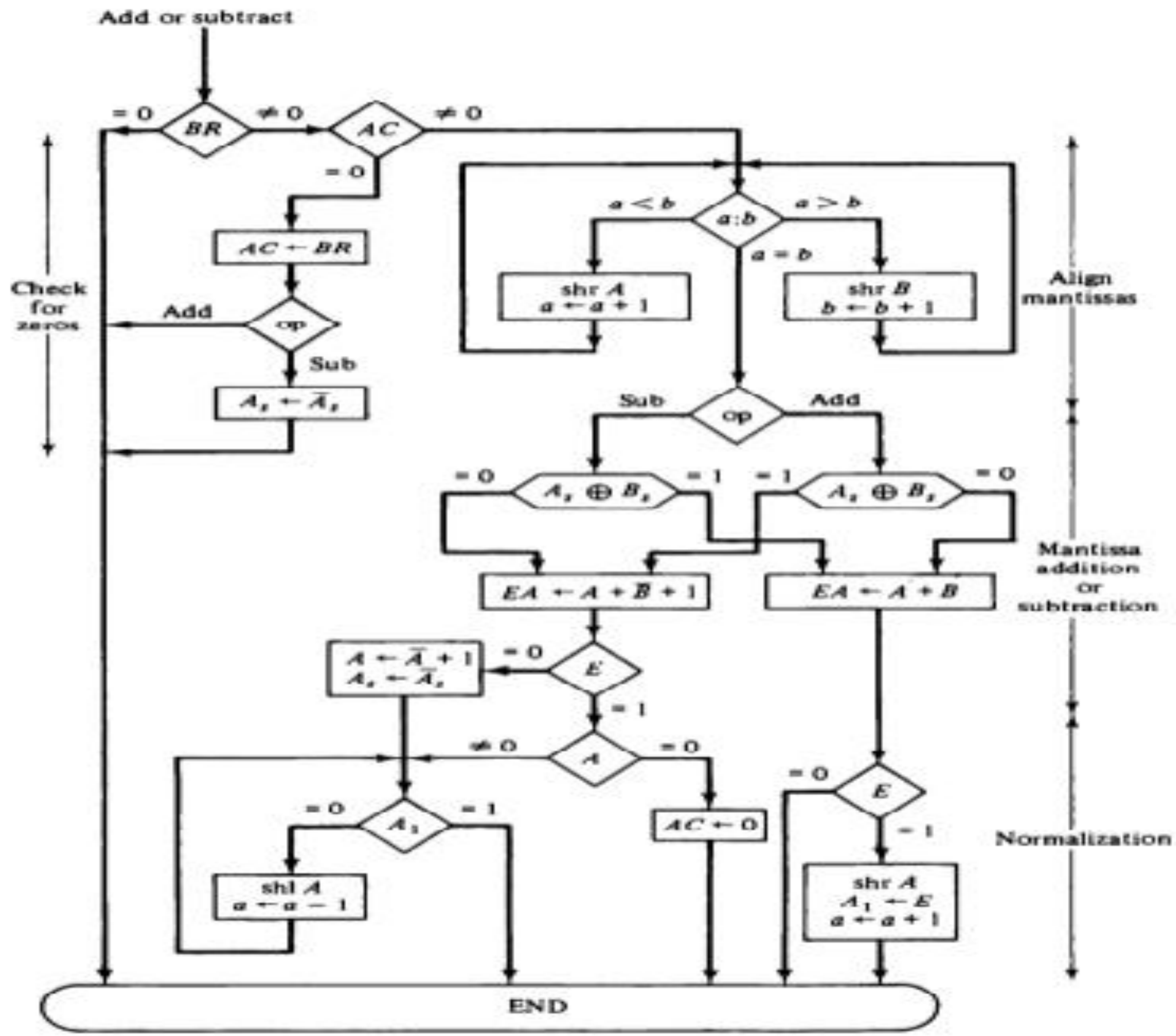
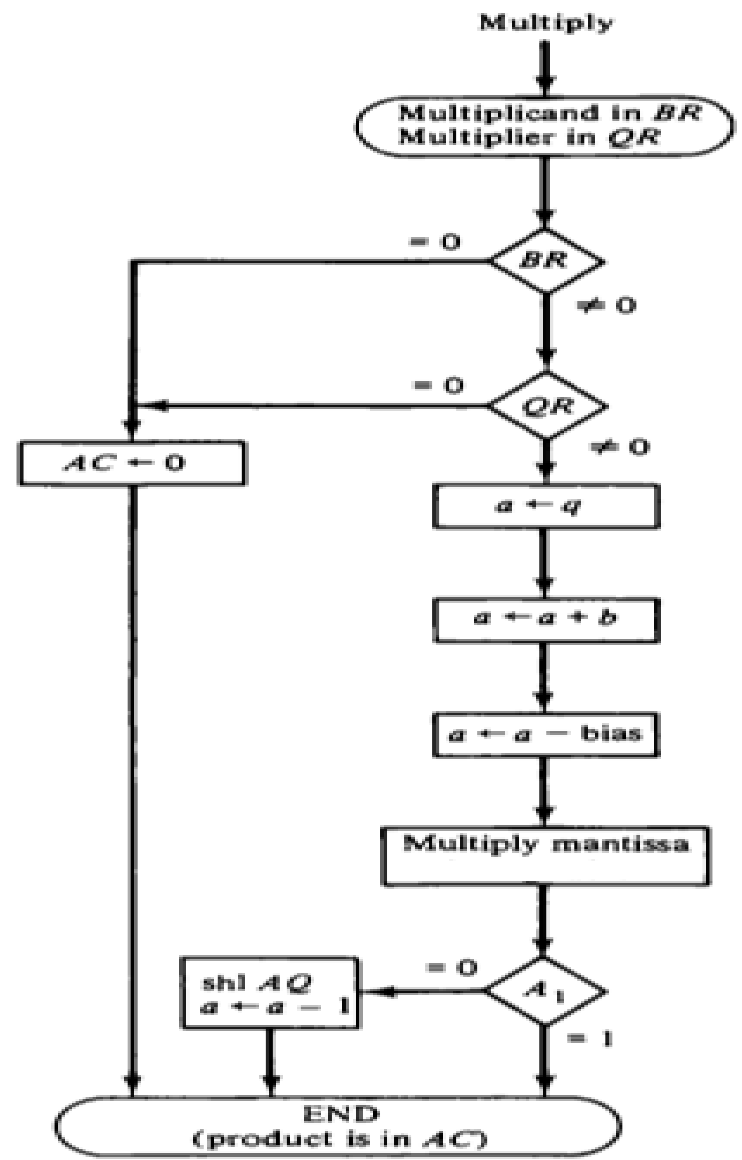


Figure 10-15 Addition and subtraction of floating-point numbers.

FLOATING POINT MULTIPLICATION

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product.



Floating point Division

1. Check for zeros.
2. Initialize registers and evaluate the sign.
3. Align the dividend.
4. Subtract the exponents.
5. Divide the mantissas.