Combinational and Sequential Circuits

Combinational circuits are defined as the time independent circuits which do not depends upon previous inputs to generate any output are termed as combinational circuits.

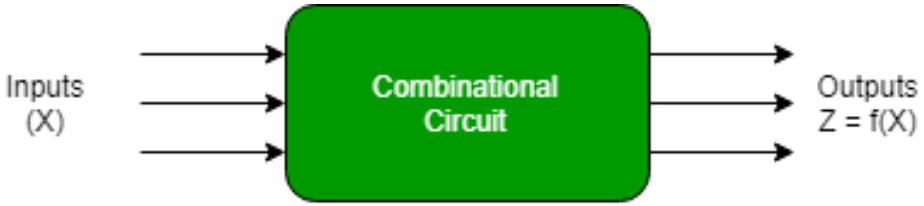


Figure: Combinational Circuits

Sequential circuits are those which are dependent on clock cycles and depends on present as well as past inputs to generate any output.

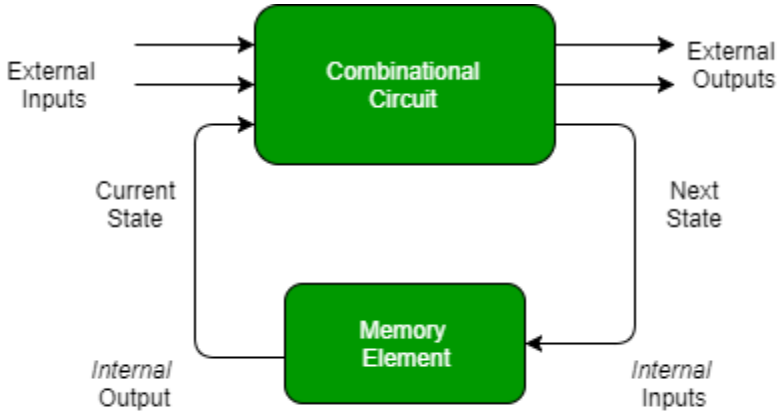


Figure: Sequential Circuit

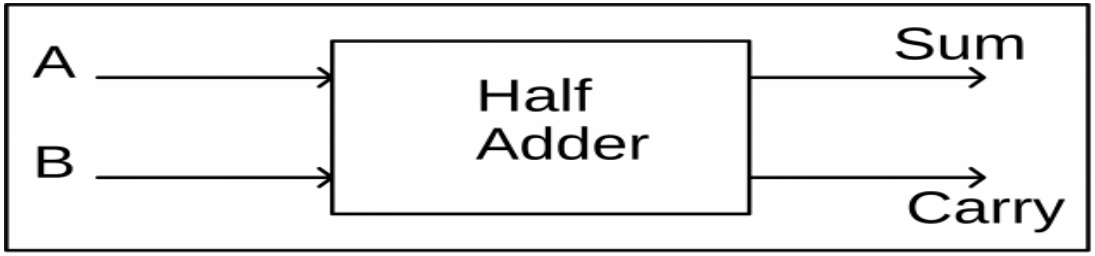
Adder

Adder :- A combinational circuit that performs the addition of bits is called an Adder. Each computer has an adder located in its CPU(ALU) that is responsible for the process of addition ,calculation of memory address and many other work. There are two types of Adder..

Half Adder

Full Adder

Half Adder:-A combinational circuit that performs the addition of two bits is called a Half Adder. It receives two inputs and produces two outputs Sum and Carry. The **block diagram** for a **half adder** is as follows.



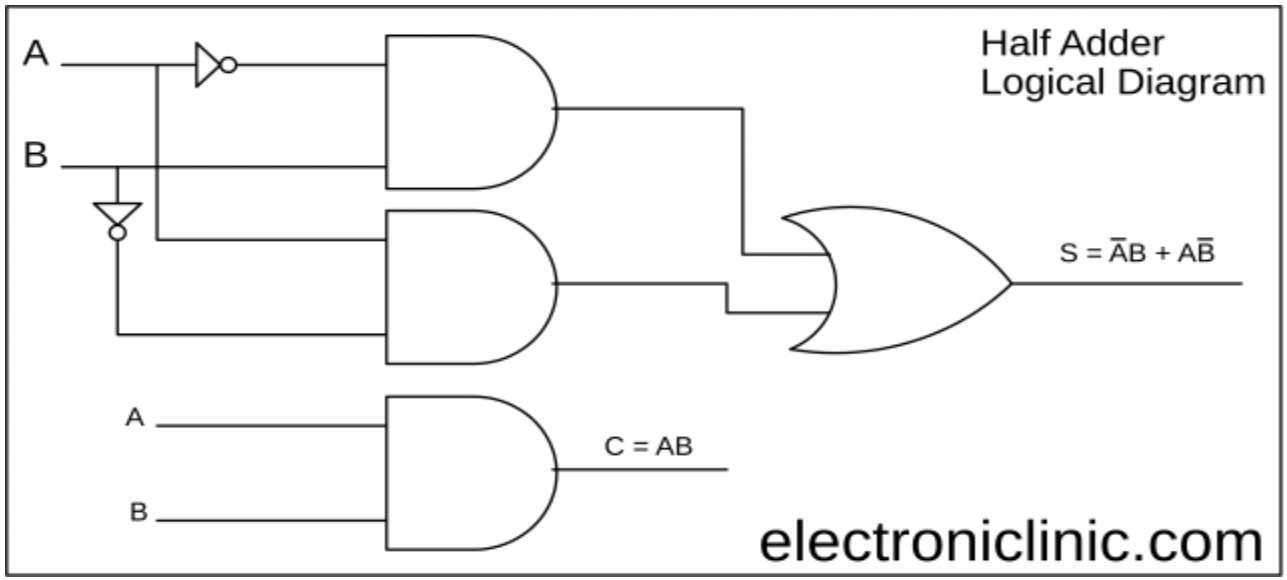
Input		Output	
A	B	Sum = S	Carry = C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

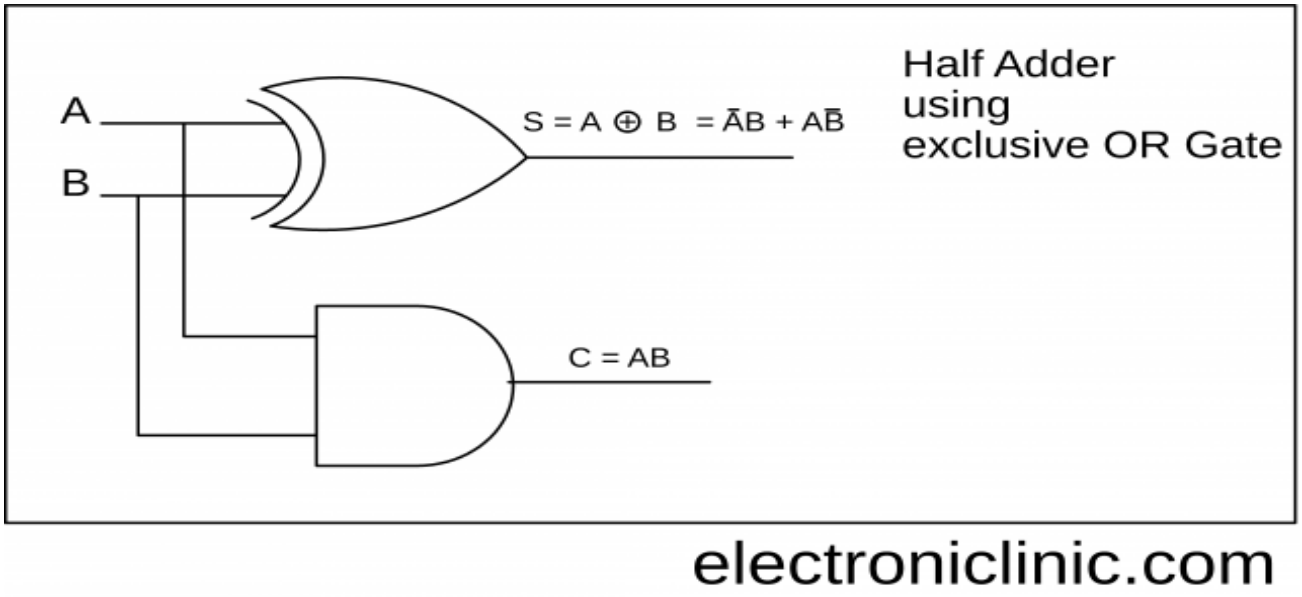
electroniclinic.com

$$Sum(A,B) = \bar{A}B + A\bar{B}$$

$$Carry(A,B) = AB$$

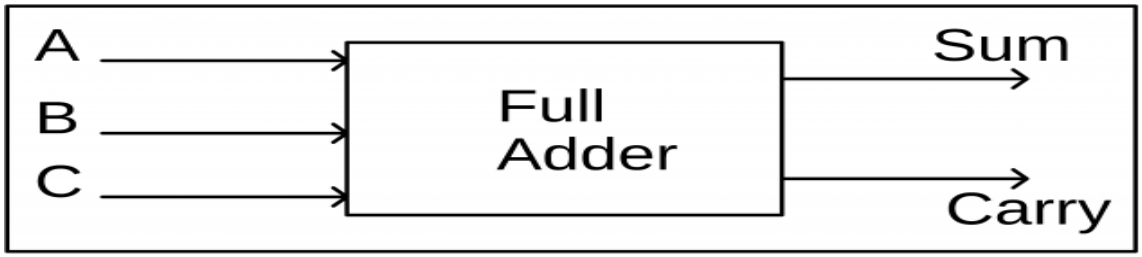
Adder


Half Adder
Logical Diagram
A
B
S = $\bar{A}$B + A$\bar{B}$
A
B
C = AB
electroniclinic.com


Half Adder
using
exclusive OR Gate
A
B
S = A ⊕ B = $\bar{A}$B + A$\bar{B}$
C = AB
electroniclinic.com

Adder

Full Adder:- A combinational circuit that performs the addition of three bits is called a Full Adder. It receives three inputs and produces two outputs Sum and Carry. The Block diagram for the Full Adder is shown below.

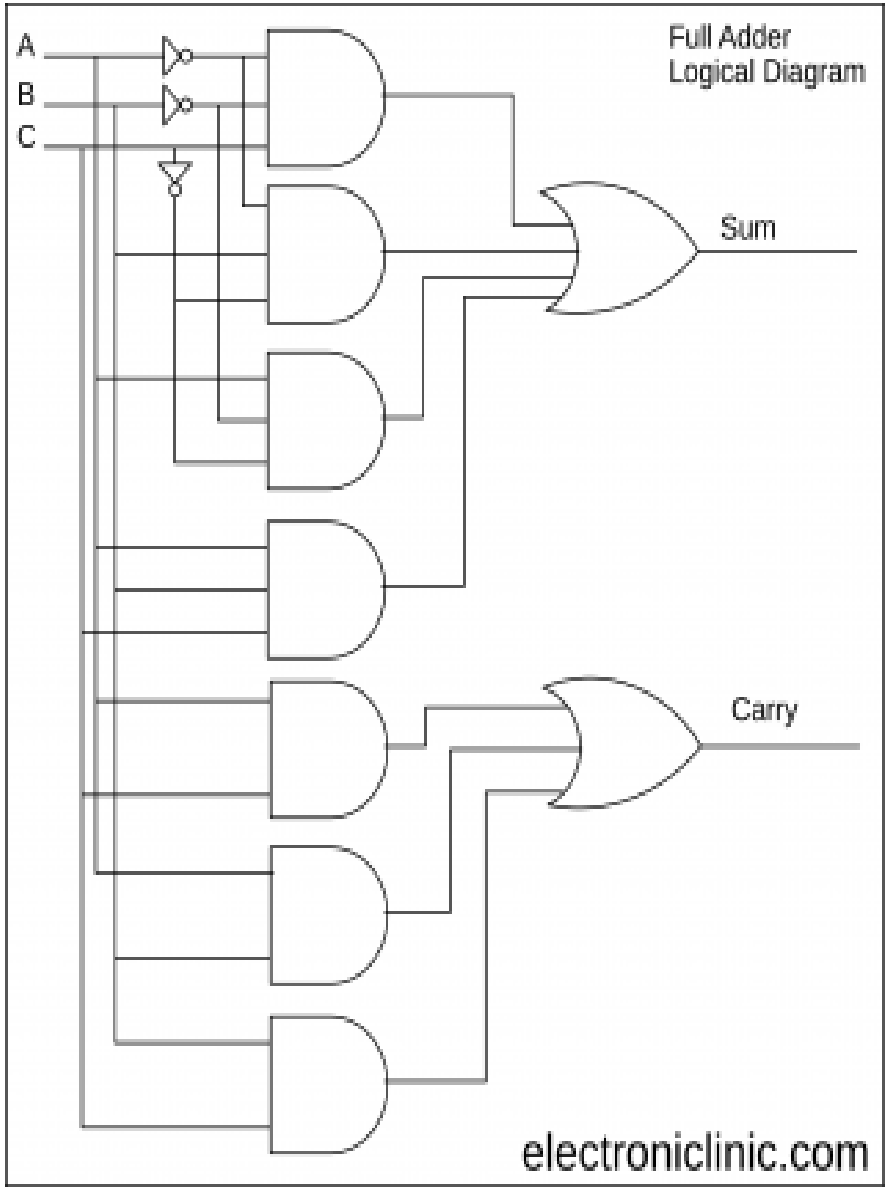


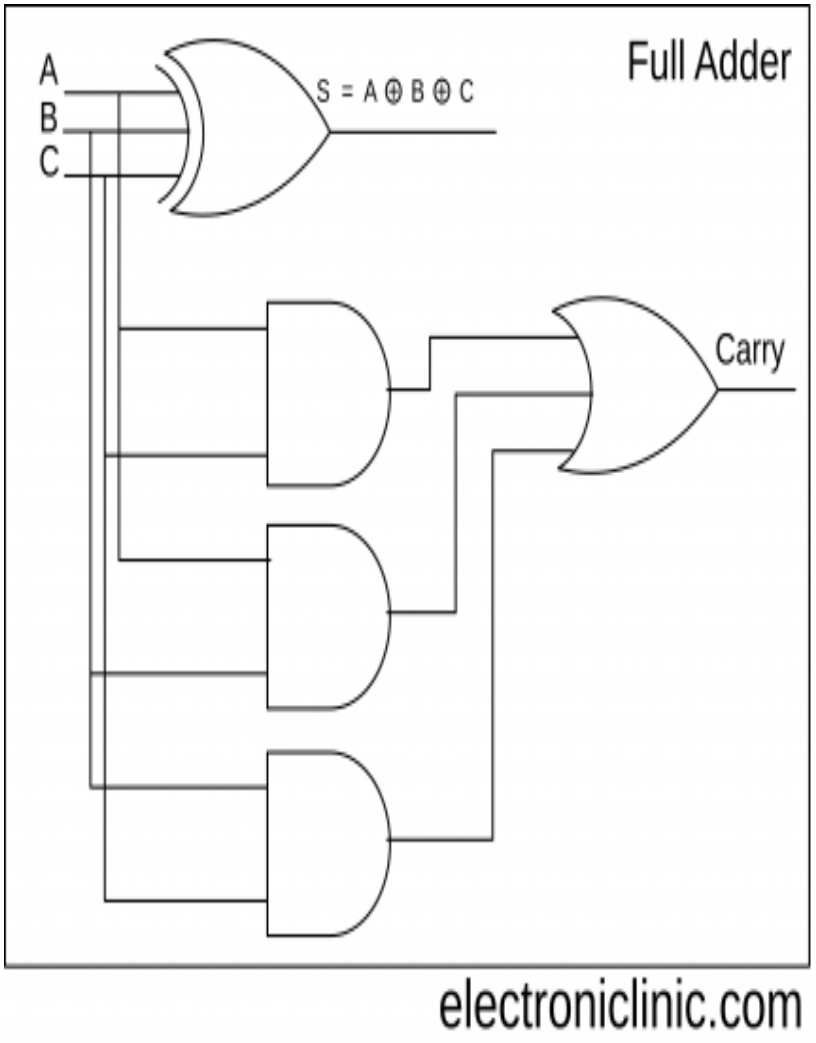
Input			Output	
A	B	C	Sum = S	Carry = C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\text{Sum}(A,B,C) = \bar{A}\bar{B}C+\bar{A}B\bar{C}+A\bar{B}\bar{C}+ABC$$

$$\text{Carry}(A,B,C) = \bar{A}BC+A\bar{B}C+AB\bar{C}+ABC$$

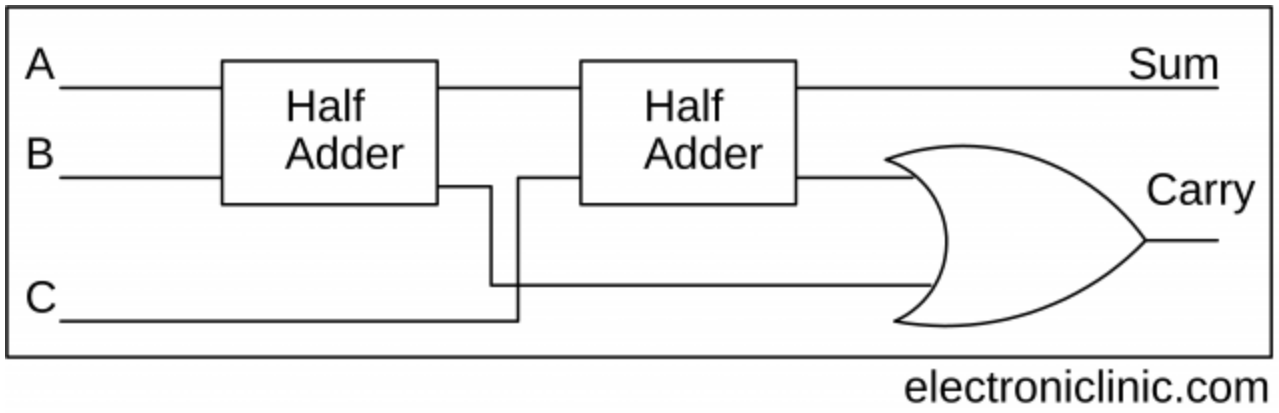
Adder

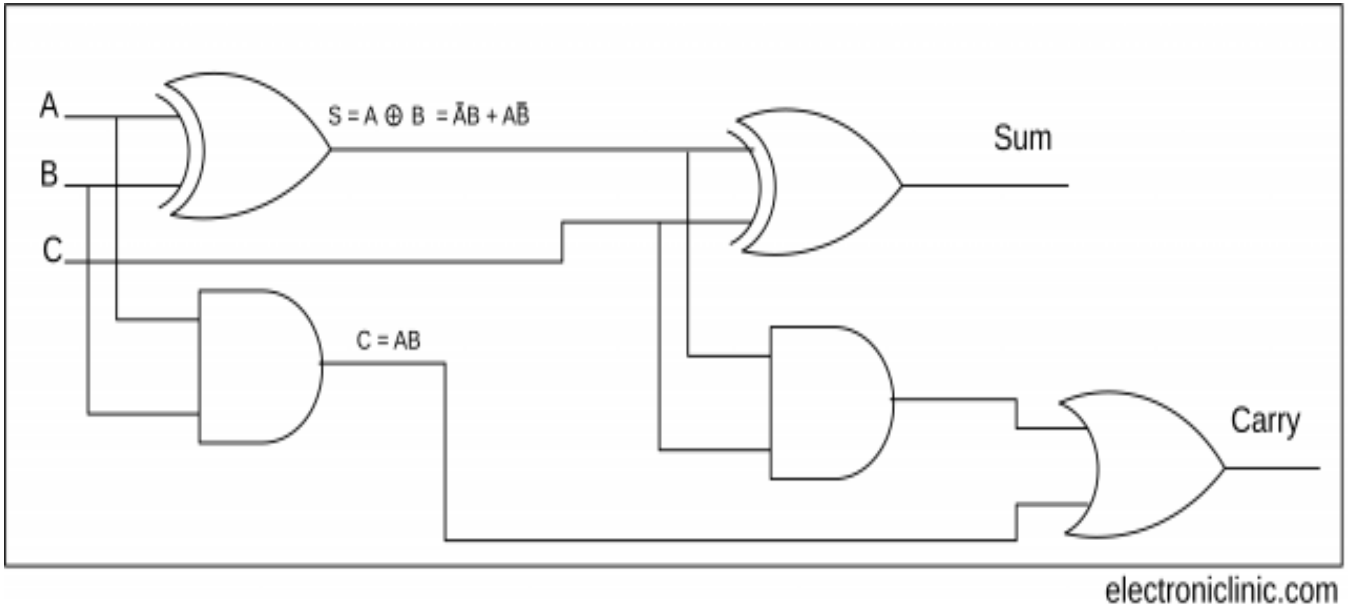

Full Adder
Logical Diagram
A
B
C
Sum
Carry
electroniclinic.com


Full Adder
A
B
C
S = A⊕B⊕C
Carry
electroniclinic.com

Adder

Full Adder Using 2 half Adder



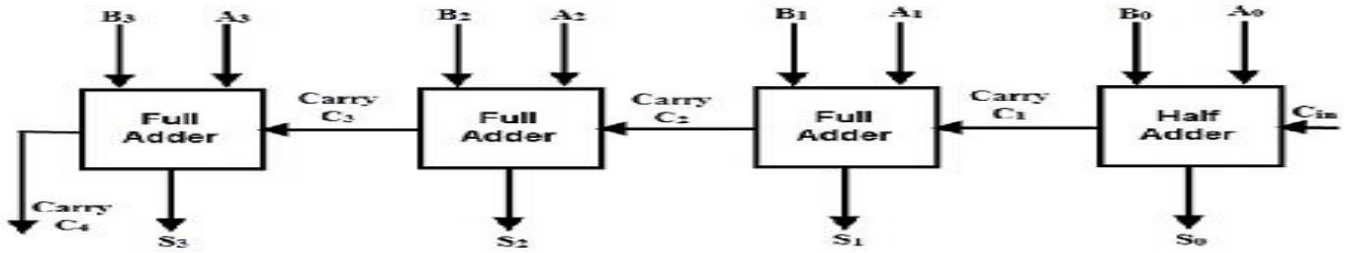


Adder

Disadvantage of Full Adder:

In parallel adders, carry output of each full adder is given as a carry input to the next higher-order state. Hence, these adders it is not possible to produce carry and sum outputs of any state unless a carry input is available for that state.

So, for computation to occur, the circuit has to wait until the carry bit propagated to all states. This induces carry propagation delay in the circuit.



Consider the 4-bit ripple carry adder circuit above. Here the sum S3 can be produced as soon as the inputs A3 and B3 are given. But carry C3 cannot be computed until the carry bit C2 is applied whereas C2 depends on C1. Therefore to produce final steady-state results, carry must propagate through all the states. This increases the carry propagation delay of the circuit.

The propagation delay of the adder is calculated as **"the propagation delay of each gate times the number of stages in the circuit".** For the computation of a large number of bits, more stages have to be added, which makes the delay much worse. Hence, to solve this situation, **Carry Look-ahead Adder was introduced**.

Carry Look-ahead Adder

Carry Look-ahead Adder :- is the faster adder circuit. It reduces the propagation delay, which occurs during addition, by using more complex hardware circuitry. It is designed by transforming the ripple-carry Adder circuit such that the carry logic of the adder is changed into two-level logic**. (Predict the Carry)**

A	B	Ci	C i +1	Condition
0	0	0	0	No carry generate
0	0	1	0	
0	1	0	0	
0	1	1	1	No carry propagate
1	0	0	0	
1	0	1	1	
1	1	0	1	Carry generate
1	1	1	1	

Carry Look-ahead Adder

Carry Generate Gi =1. It depends on Ai and Bi inputs. Gi is 1 when both Ai and Bi are 1. Hence, Gi is calculated as Gi = Ai. Bi.

Carry propagated Pi is associated with the propagation of carry from Ci to Ci+1. It is calculated as Pi = Ai $\oplus$ Bi

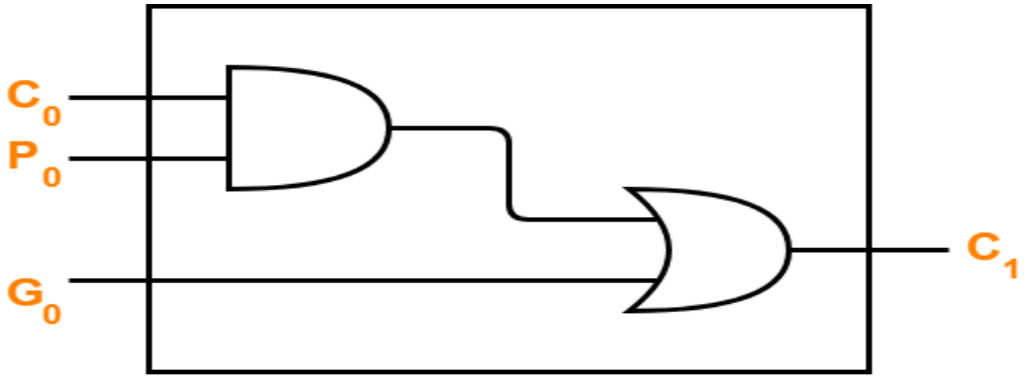
Using the Gi and Pi terms the Sum Si and Carry Ci+1 are given as below –

- Si = Pi $\oplus$ Ci.
- Ci+1 = Ci.Pi +Gi.

Therefore, the carry bits C1, C2, C3, and C4 can be calculated as

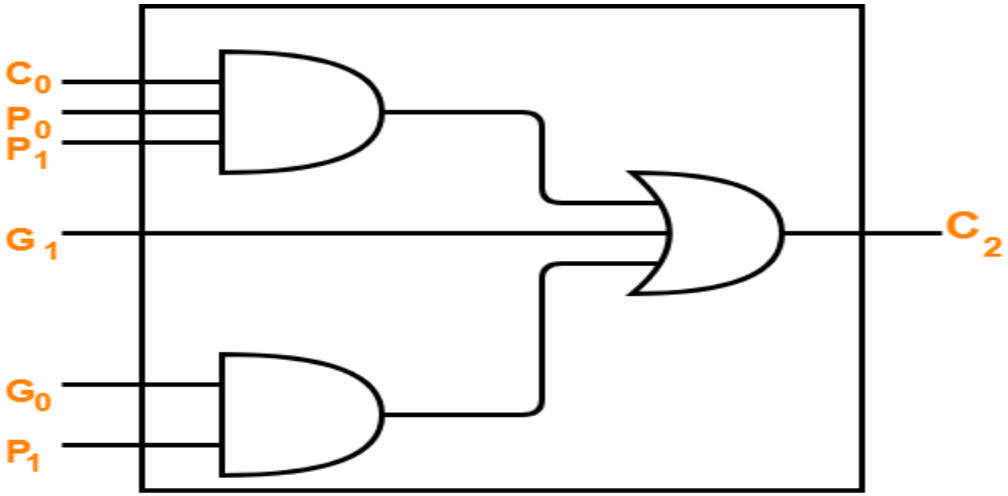
- C1 = C0.P0+G0.
- C2 = C1.P1+G1 = (C0.P0+G0).P1+G1.
- C3 = C2.P2+G2 = (C1.P1+G1).P2+G2.
- C4 = C3.P3+G3 = C0.P0.P1.P2.P3 + P3.P2.P1.G0 + P3.P2.G1 + G2.P3 + G3.

Carry Look-ahead Adder



Implementation of C1

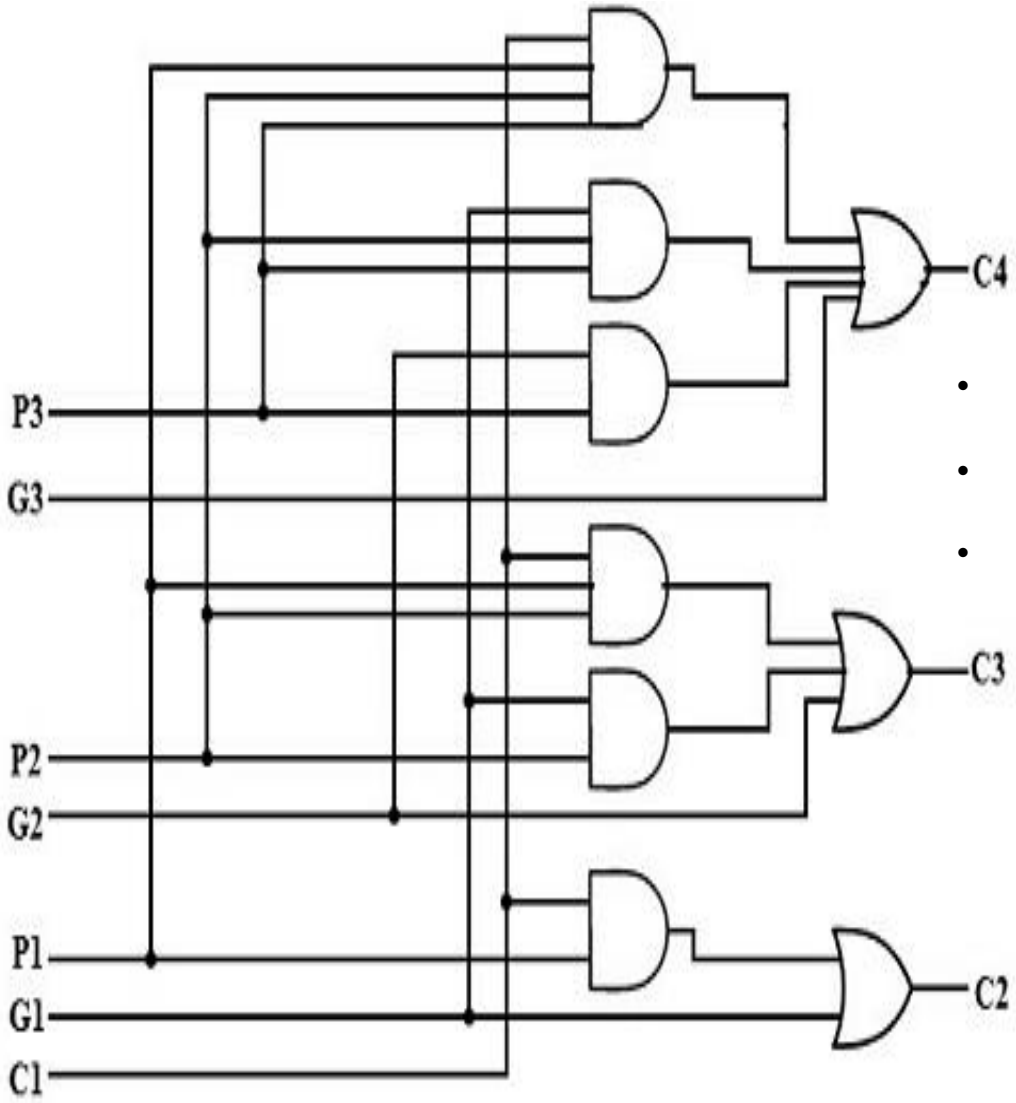
$C_1 = C_0P_0 + G_0$



Implementation of C2

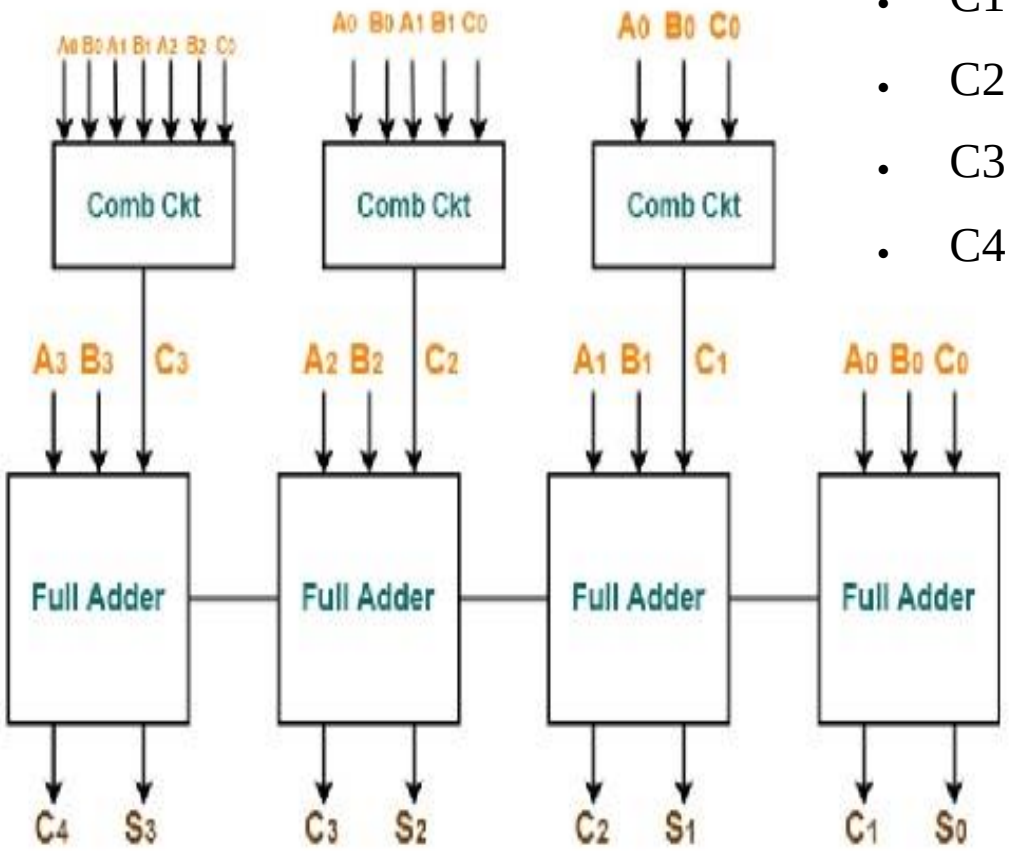
$C_2 = C_0P_0P_1 + G_0P_1 + G_1$

Carry Look-ahead Adder



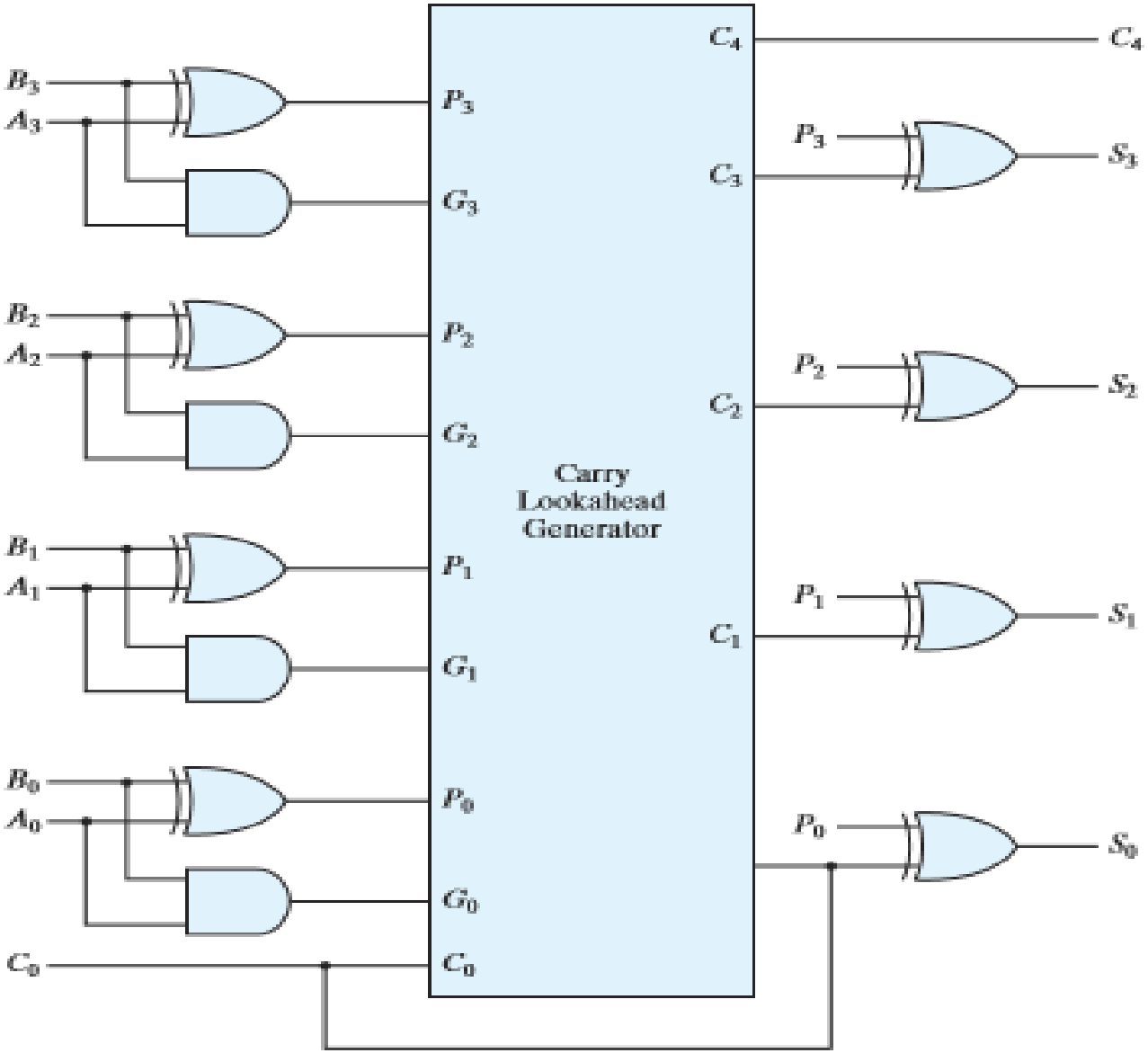
- C2 = C1.P1+G1 = (C0.P0+G0).P1+G1.
- C3 = C2.P2+G2 = (C1.P1+G1).P2+G2.
- C4 = C3.P3+G3 = C0.P0.P1.P2.P3 + P3.P2.P1.G0 + P3.P2.G1 + G2.P3 + G3

Carry Look-ahead Adder



- C1 = C0P0 + G0
- C2 = C1.P1+G1 = (C0.P0+G0).P1+G1.
- C3 = C2.P2+G2 = (C1.P1+G1).P2+G2.
- C4 = C3.P3+G3 = C0.P0.P1.P2.P3 + P3.P2.P1.G0 + P3.P2.G1 + G2.P3 + G3

Carry Look-ahead Adder



MULTIPLICATION OF TWO NUMBERS

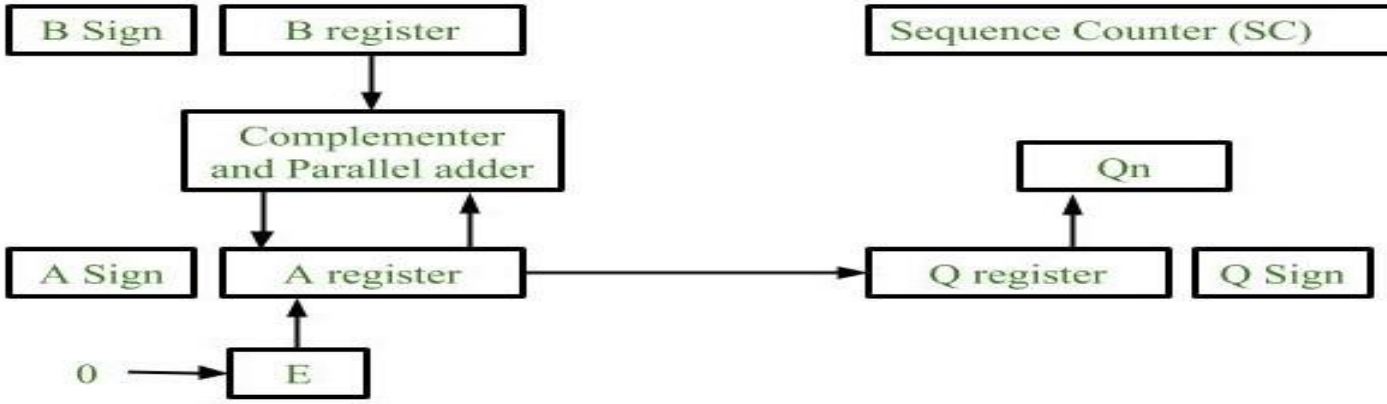
Multiplication of two fixed point binary number in *signed magnitude representation* is done with process of *successive shift* and *add operation*.

```
   10111 (Multiplicand)
x  10011 (Multiplier)
  ------
   10111
  10111
 00000
00000
10111
---------
011011010    (Product)
---------
```

In the multiplication process we are considering successive bits of the multiplier, least significant bit first.
If the multiplier bit is 1, the multiplicand is copied down else 0's are copied down.
The numbers copied down in successive lines are shifted one position to the left from the previous number.
Finally numbers are added and their sum form the product.
The sign of the product is determined from the sign of the multiplicand and multiplier.
If they are alike, sign of the product is positive else negative.

MULTIPLICATION OF TWO NUMBERS

Hardware Implementation for Multiplying Binary Numbers



1.Initially multiplicand is stored in B register and multiplier is stored in Q register.

2.Sign of registers B (Bs) and Q (Qs) are compared using **XOR** functionality (i.e., if both the signs are alike, output of XOR operation is 0 unless 1) and output stored in As (sign of A register).**Note:** Initially 0 is assigned to register A and E flip flop. Sequence counter is initialized with value n, n is the number of bits in the Multiplier.

3. Now least significant bit of multiplier is checked. If it is 1 add the content of register A with Multiplicand (register B) and result is assigned in A register with carry bit in flip flop E. Content of E A Q is shifted to right by one position, i.e., content of E is shifted to most significant bit (MSB) of A and least significant bit of A is shifted to most significant bit of Q.

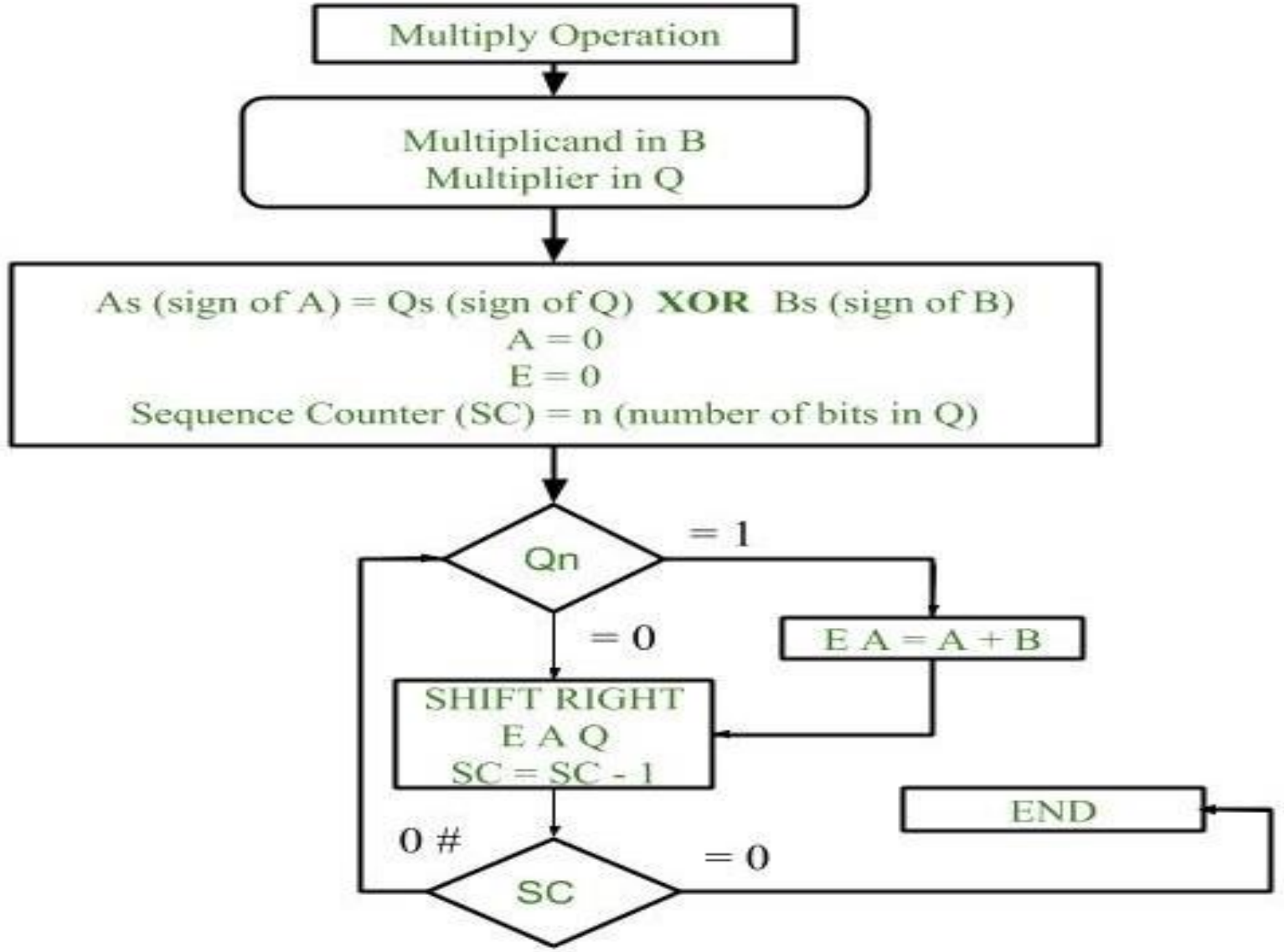
4. If Qn = 0, only shift right operation on content of E A Q is performed in a similar fashion.

5. Content of Sequence counter is decremented by 1.

6. Check the content of Sequence counter (SC), if it is 0, end the process and the final product is present in register A and Q, else repeat the process.

MULTIPLICATION OF TWO NUMBERS

Flowchart for Multiplying Binary Numbers



MULTIPLICATION OF TWO NUMBERS

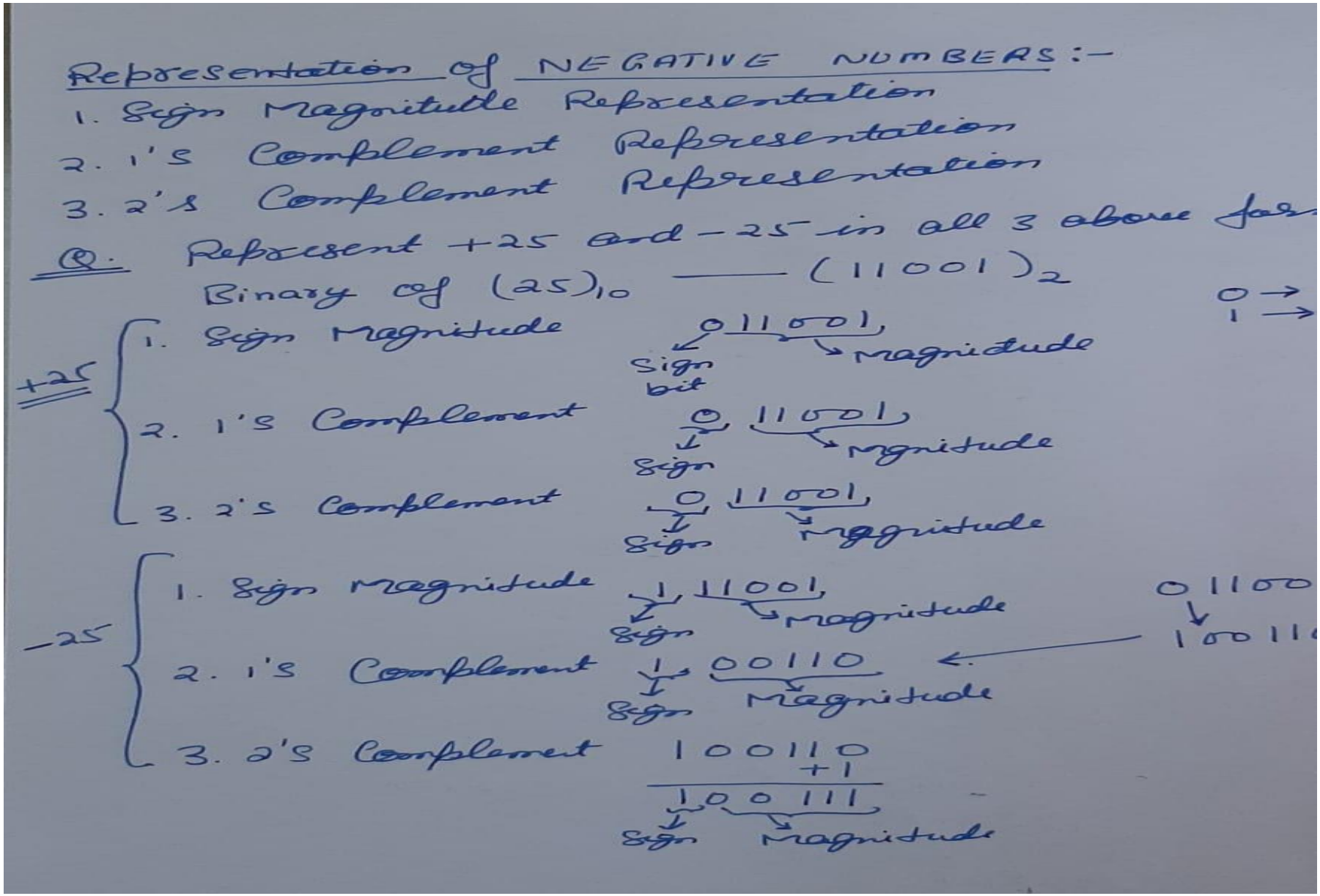
Multiplicand = 10111

Multiplier = 10011

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

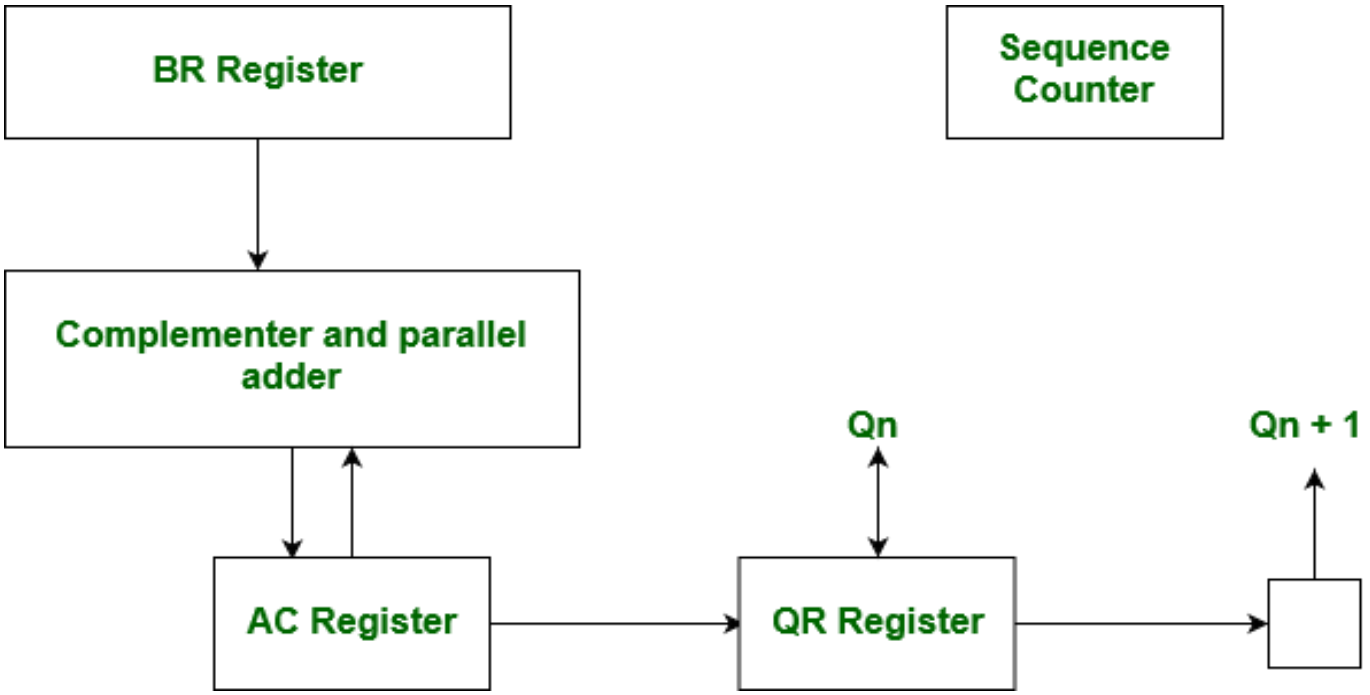
Final product in AQ
0110110101

2's Compliment Representation



MULTIPLICATION OF TWO NUMBERS

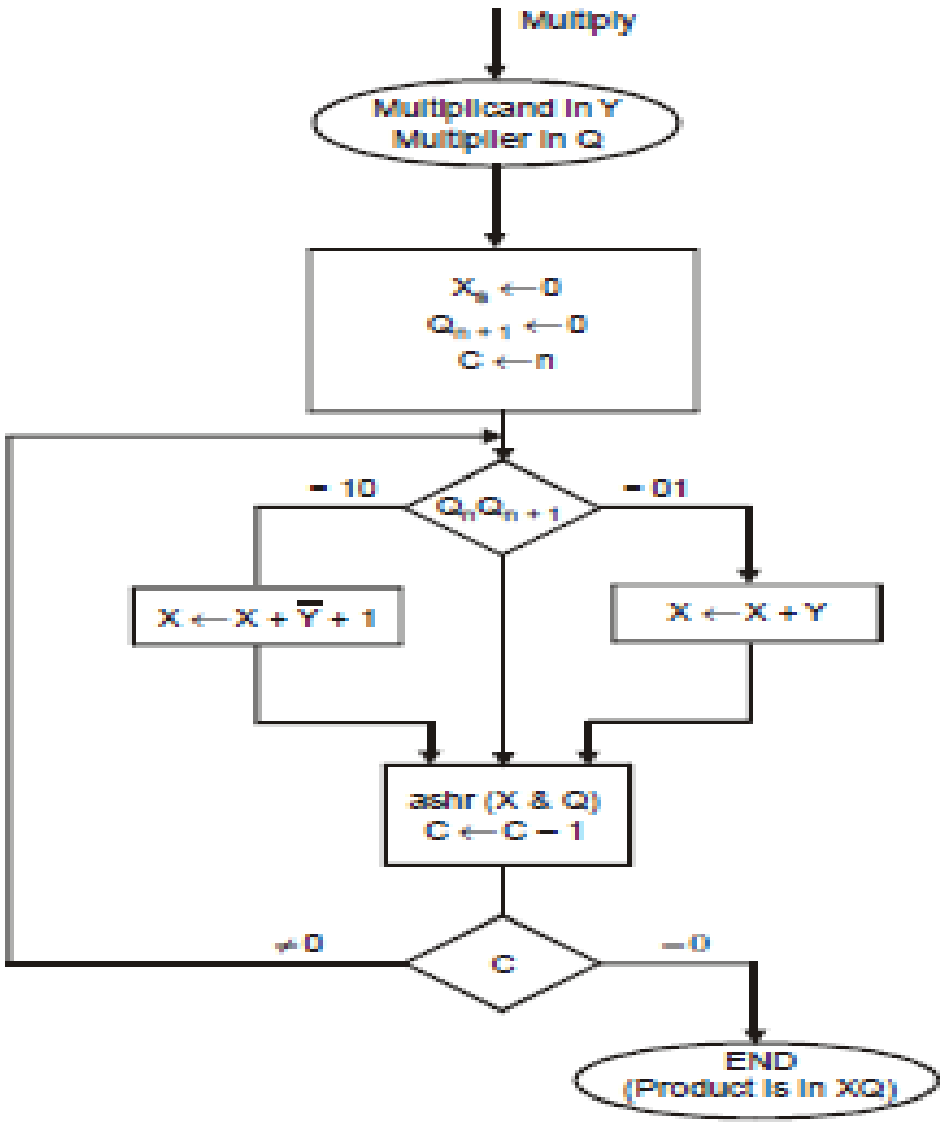
Booth algorithm gives a procedure for **multiplying binary integers** in signed 2's complement representation **in efficient way**, i.e., less number of additions/subtractions required.



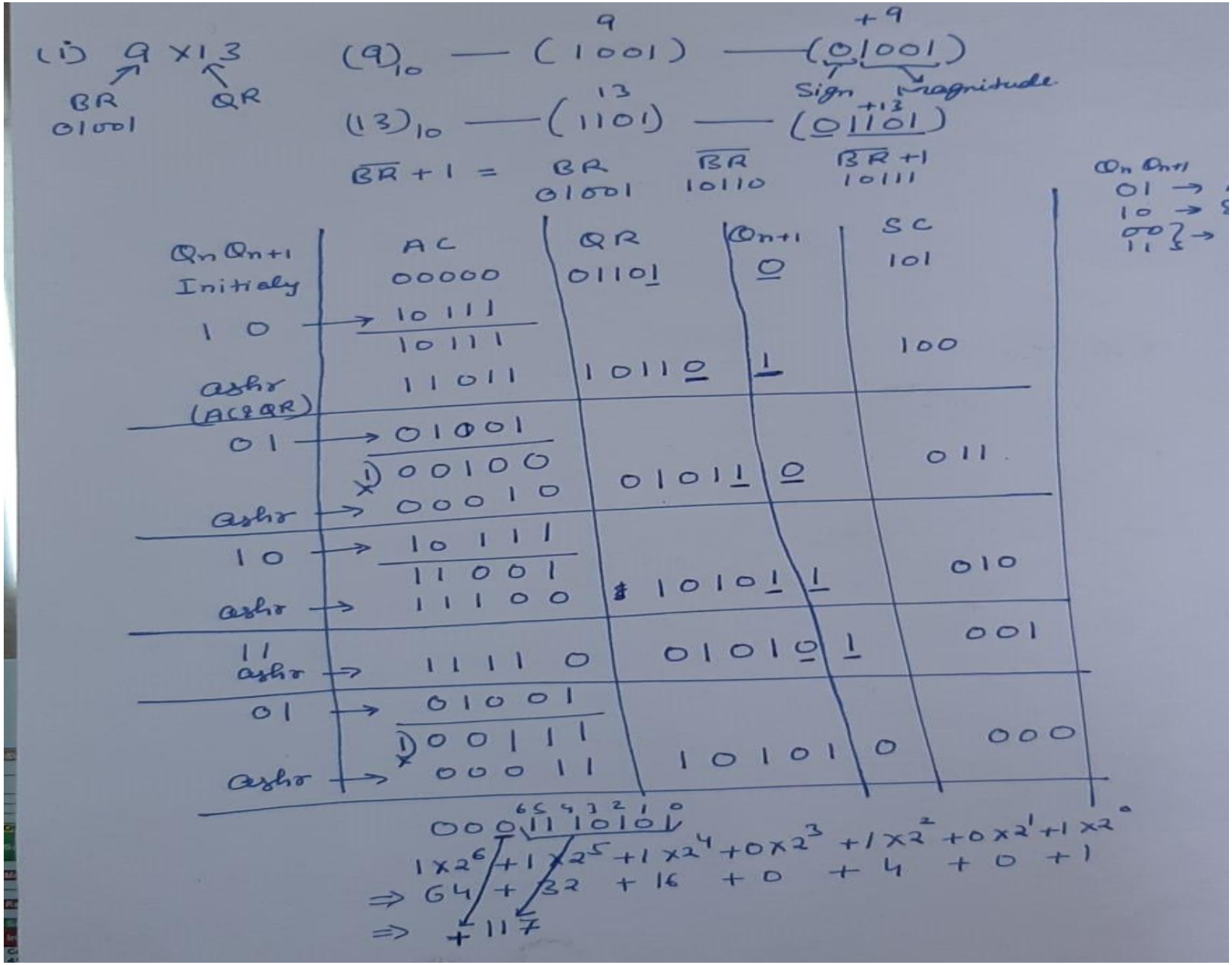
Hardware Implementation

MULTIPLICATION OF TWO NUMBERS

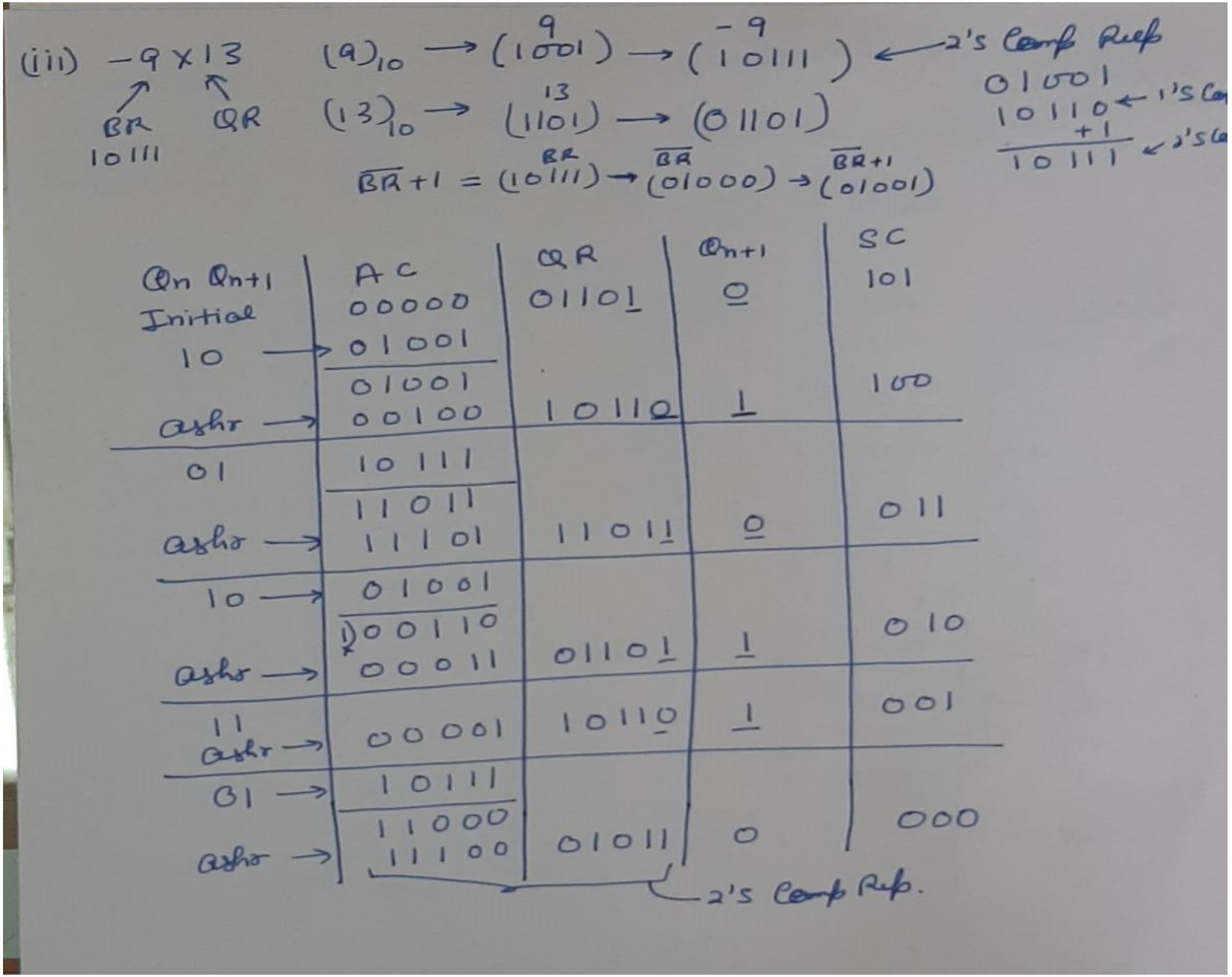
BOOTH.S MULTIPLICATION ALGORITHM



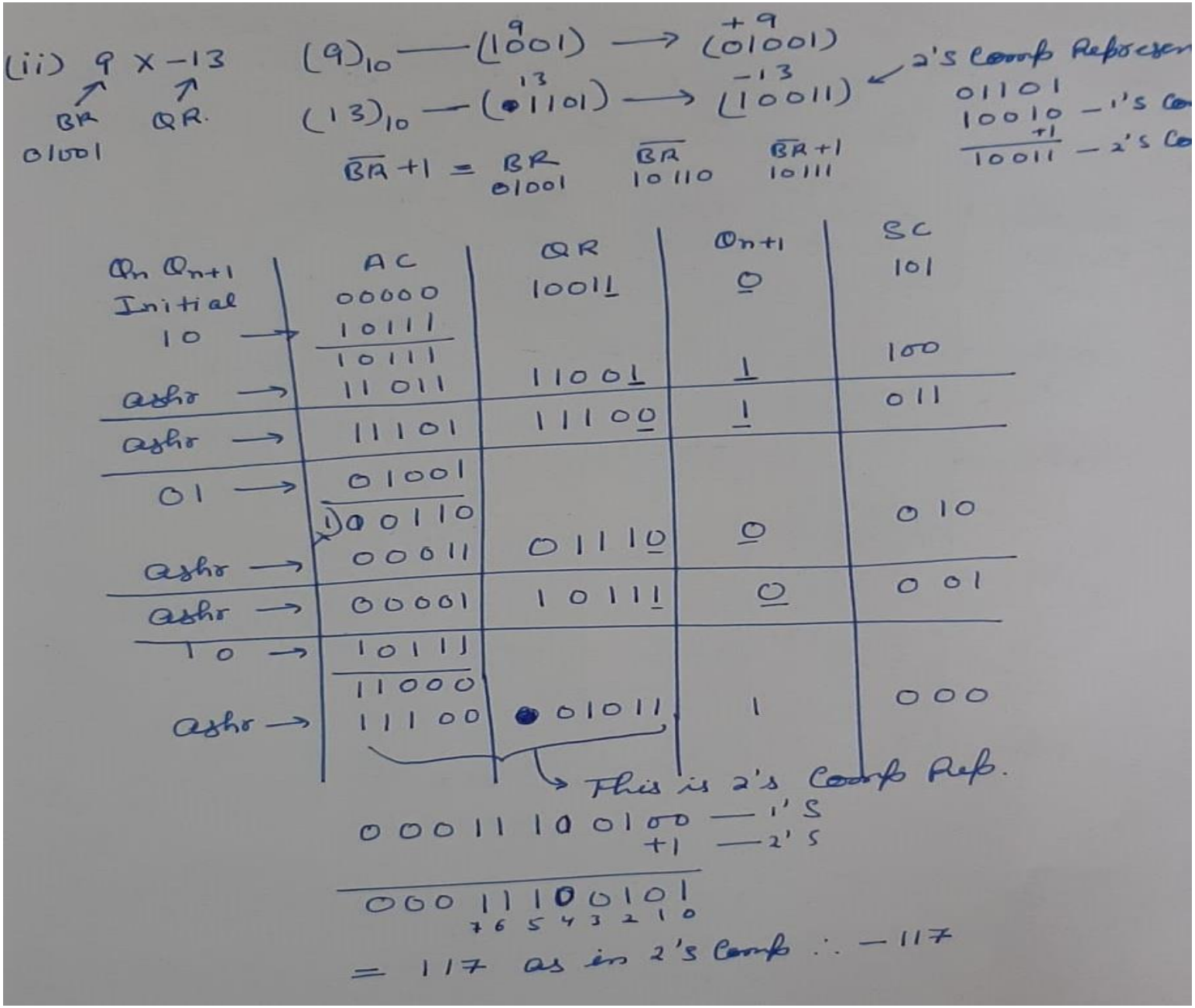
MULTIPLICATION OF TWO NUMBERS



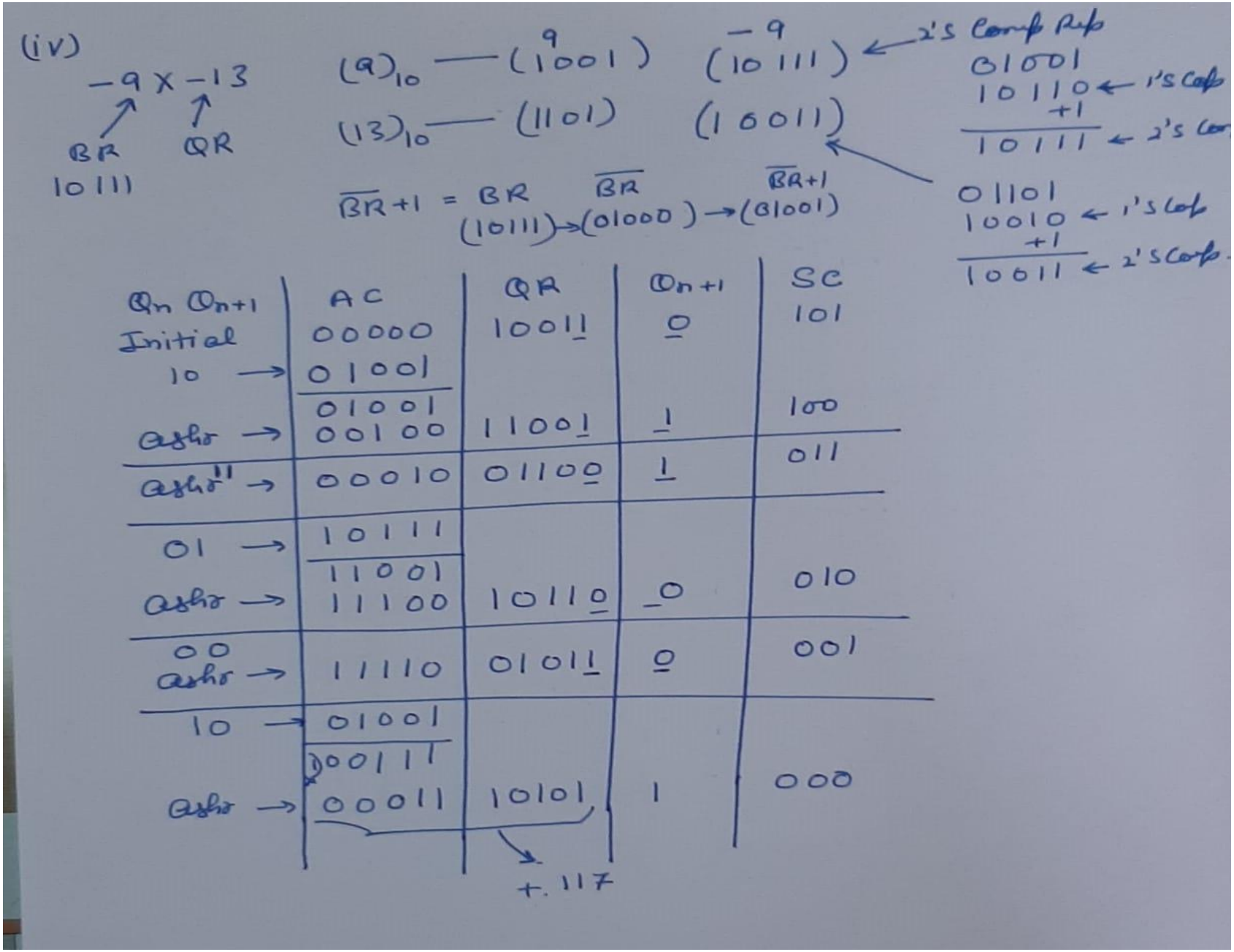
MULTIPLICATION OF TWO NUMBERS



MULTIPLICATION OF TWO NUMBERS



MULTIPLICATION OF TWO NUMBERS



Array Multiplier

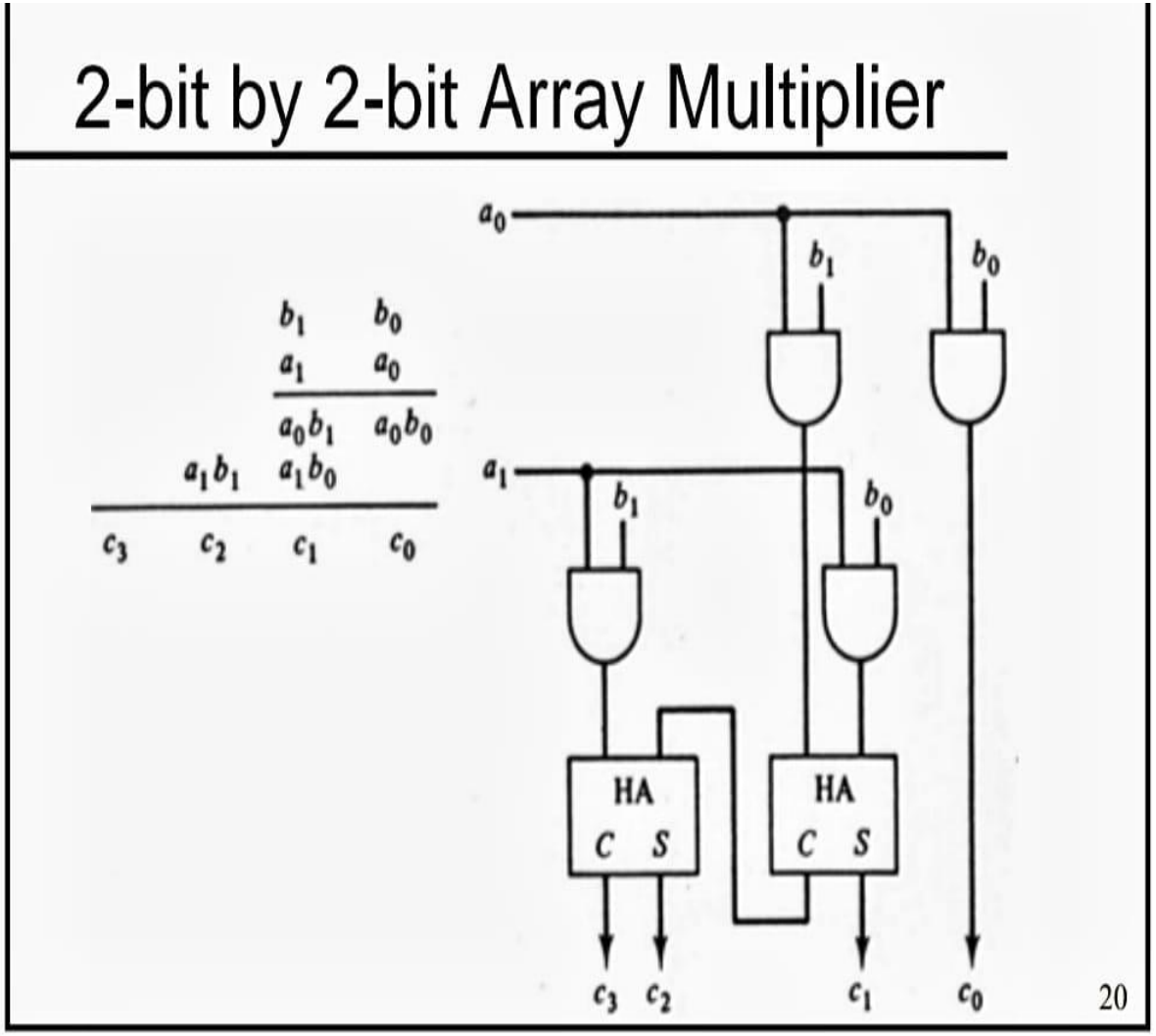
- The multiplication of two binary numbers can be done with one micro operation by means of a combinational circuit (**called Array Multiplier)** that forms the product bits all at once.
- This is a fast way of multiplying two numbers since all it takes is the time for the signals to propagate through the gates that form the multiplication array.
- However, an array multiplier requires a large number of gates, and for this reason it was not economic.
- For Multiplier j bits

  Multiplicand k bits

  j x k AND gates are required

  (j - 1) k-bit adders to produce a product of j + k bits.

Array Multiplier



Multiplier j bits

Multiplicand k bits

j x k AND gates

(j - 1) k-bit adders to produce a product of j + k bits.

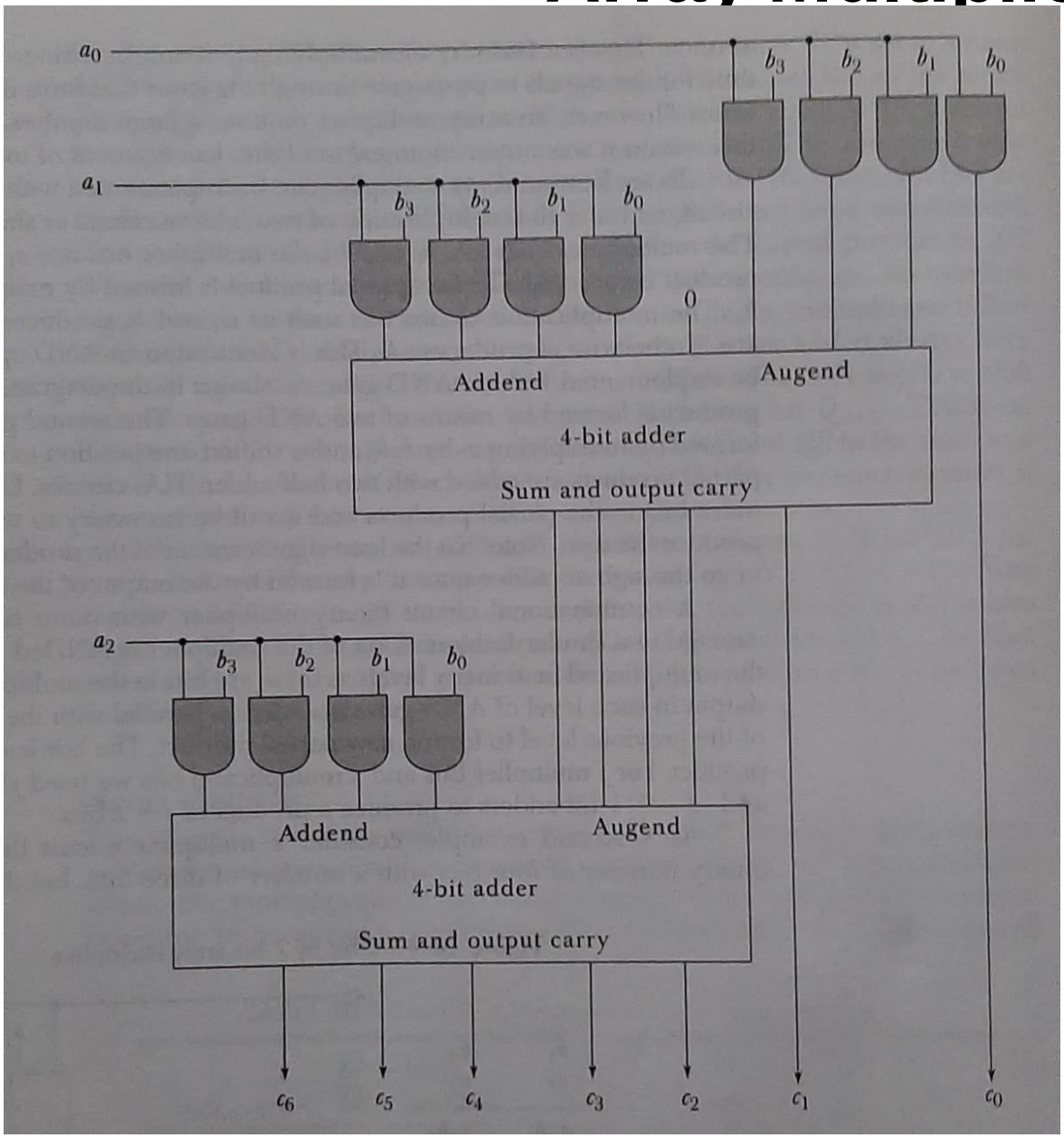
j=2,k=2

AND Gates 2*2=4

1,2 bit adders

2+2 =4 bit product

Array Multiplier



Multiplier j bits

Multiplicand k bits

j x k AND gates

(j - 1) k-bit adders to produce a product of j + k bits.

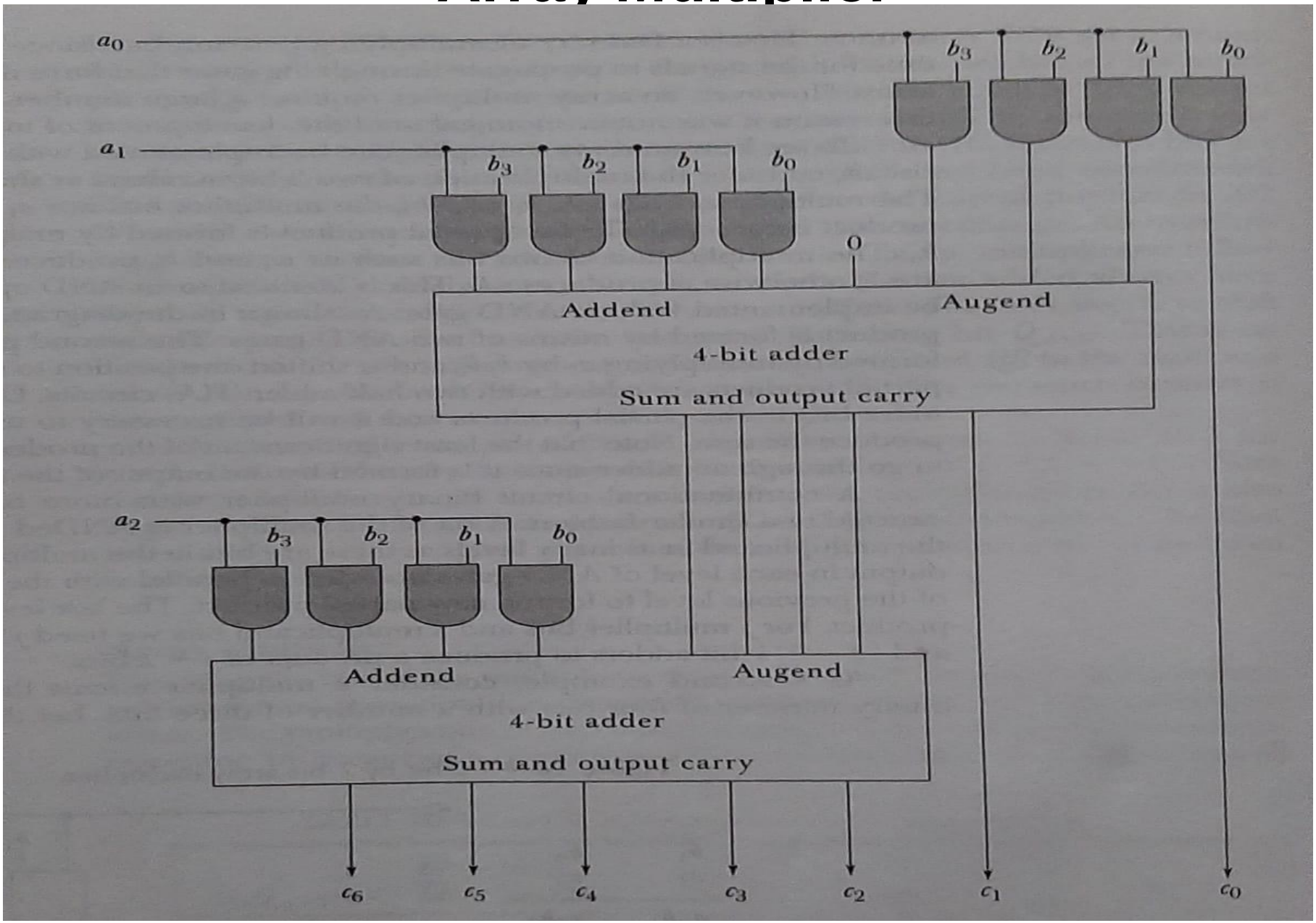
j=3,k=4

AND Gates 3*4=12

2,4bit adders

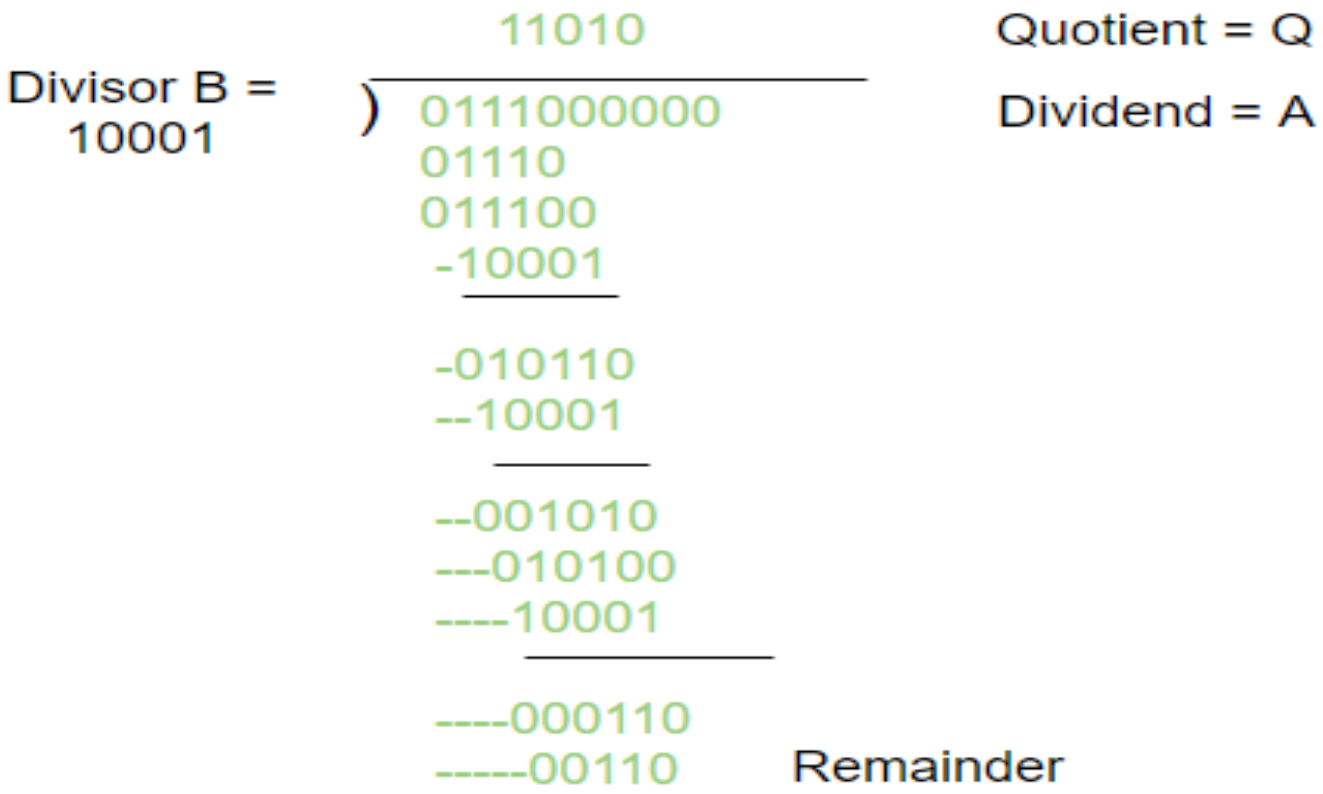
3+4=7bit product

Array Multiplier



Division Algorithm

The Division of two fixed-point binary numbers in the signed-magnitude representation is done by the cycle of successive compare, shift, and subtract operations.

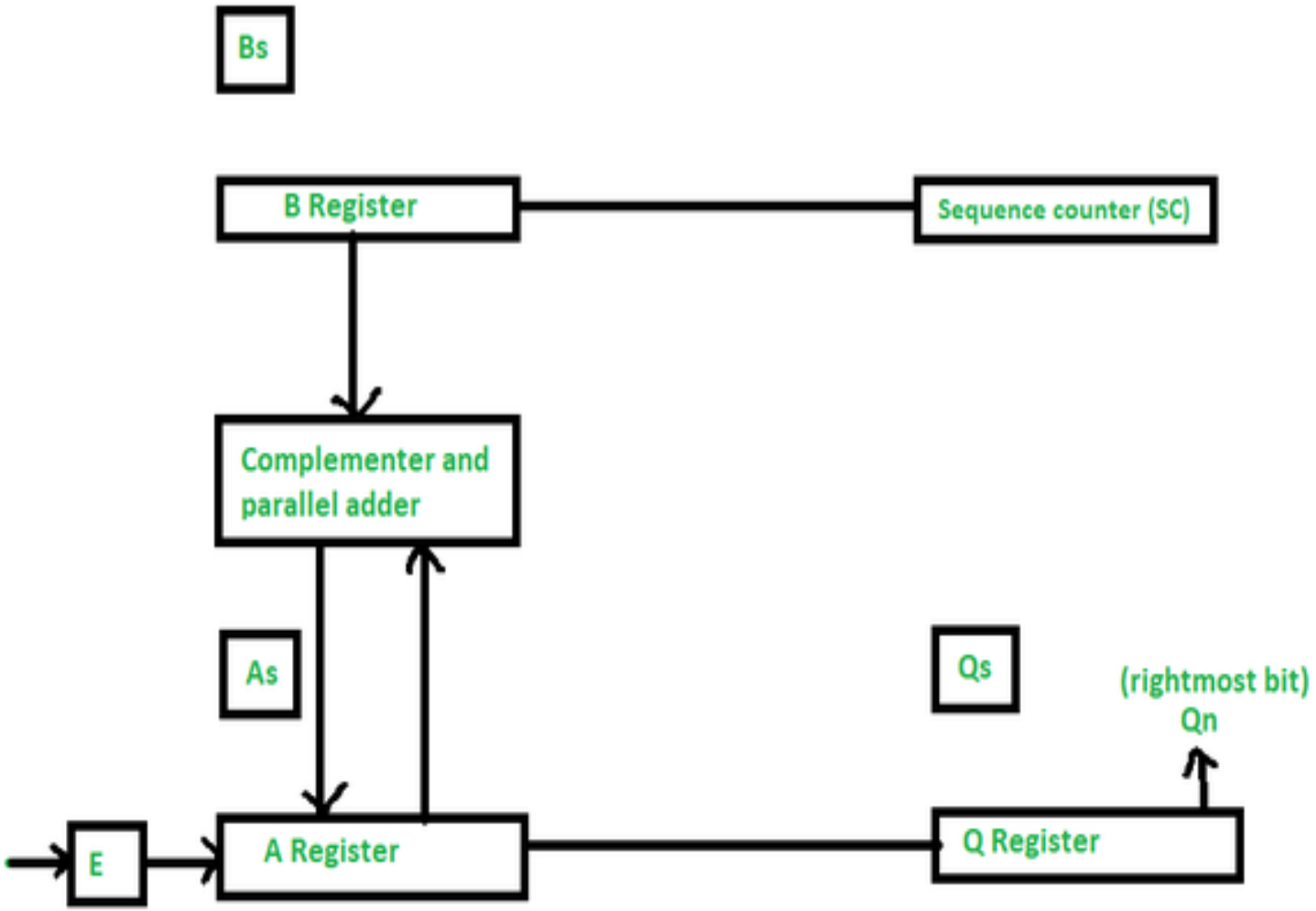


Division Algorithm

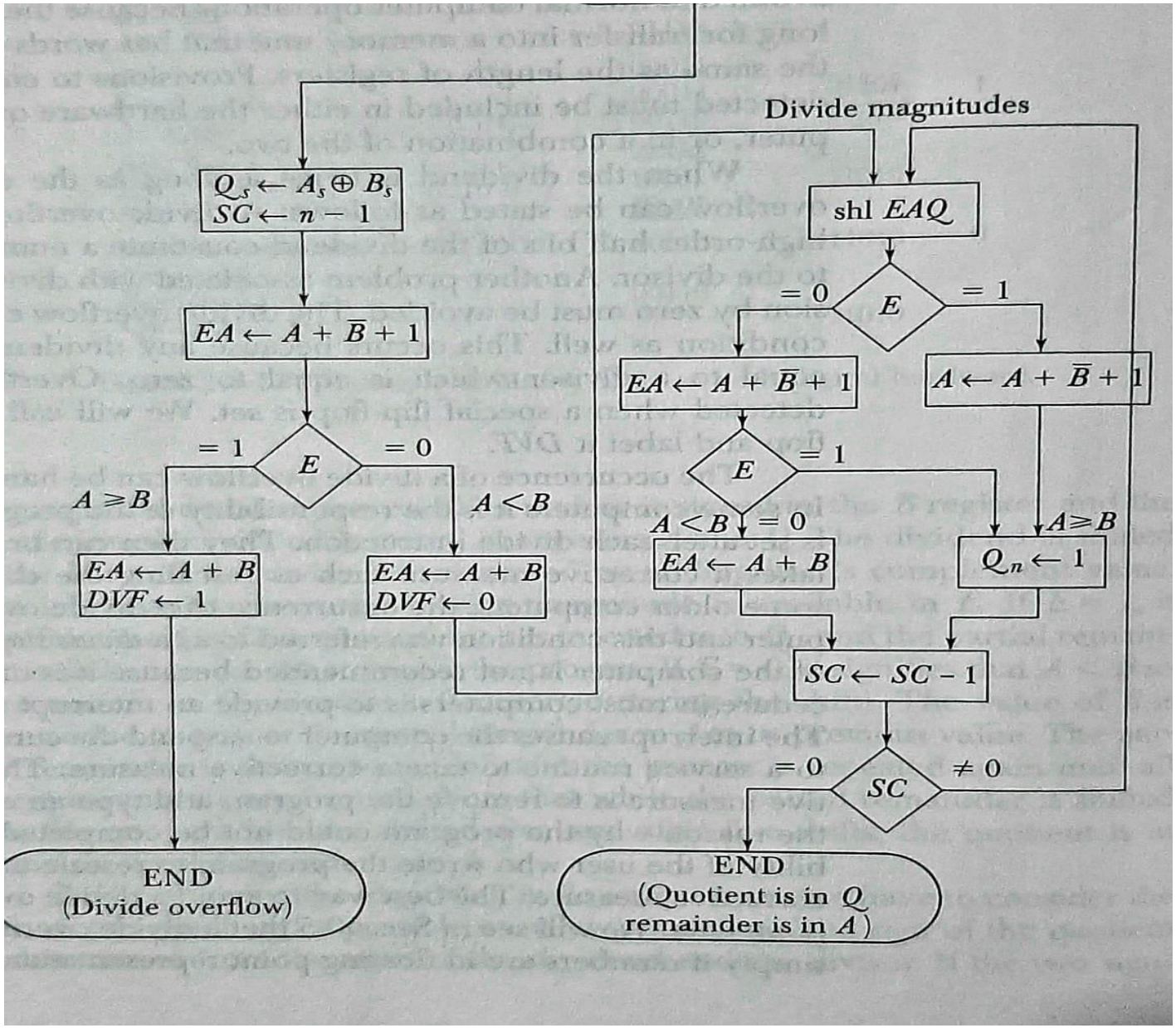
Divisor:
$B = 10001$

11010	Quotient = Q
0111000000	Dividend = A
01110	5 bits of $A < B$, quotient has 5 bits
011100	6 bits of $A \geq B$
- 10001	Shift right B and subtract; enter 1 in Q
-010110	7 bits of remainder $\geq B$
--10001	Shift right B and subtract; enter 1 in Q
--001010	Remainder $< B$; enter 0 in Q; shift right B
---010100	Remainder $\geq B$
----10001	Shift right B and subtract; enter 1 in Q
----000110	Remainder $< B$; enter 0 in Q
-----00110	Final remainder

Division Algorithm



Division Algorithm



Division Algorithm

Divisor B = 10001	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B}$ + 1		01111		
E = 1	1	01011		
Set Q_n = 1	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B}$ + 1		01111		
E = 1	1	00101		
Set Q_n = 1	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B}$ + 1		01111		
E = 0; leave Q_n = 0	0	11001	00110	
Add B		10001		2
Restore remainder	1	01010		
shl EAQ	0	10100	01100	
Add $\overline{B}$ + 1		01111		
E = 1	1	00011		
Set Q_n = 1	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B}$ + 1		01111		
E = 0; leave Q_n = 0	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

Final Remainder: 00110
Final Quotient: 11010

Floating Point Arithmetic operation

A floating point number in computer registers consists of two parts a Mantissa **m** and Exponent **e**. The two parts represent a number obtained from multiplying m times a radix **r** raised to the value of e

$$m \times r^{e}$$

For example a decimal number 537.25 is represented in a register with m=53725 and e=3 and radix(r) =10

$$.53735 \times 10^{3}$$

Consider the addition of two real decimal numbers as fixed point numbers:

```
  1234.00
+   56.78
--------
  1290.78
```

Now if we try to add the same numbers written in floating point notation, we see that simply adding the mantissas will not make sense unless the exponents are equal:

$0.1234 * 10^{4}$

$+ \quad 0.5678 * 10^{2}$

Floating Point Arithmetic operation

Thus, the following steps must be carried out before adding / subtracting two floating point numbers:

1. Make the exponents of the two numbers equal by making the smaller exponent equal to the larger and dividing the mantissa of the smaller number by the same factor by which its exponent was increased, in order to preserve the actual value of the number.
2. Add / subtract the mantissas.
3. If necessary, re-normalize the result (this is called *post-normalization*).

We apply those steps to the example above:

1. $0.1234 * 10^4$

$+ \ 0.5678 * 10^2 \ = \ + 0.005678 * 10^4$

2. $0.1234 * 10^4$

$+ \ 0.005678 * 10^4$

$0.129078 * 10^4$

3. The result is already normalized.

Floating Point Arithmetic operation

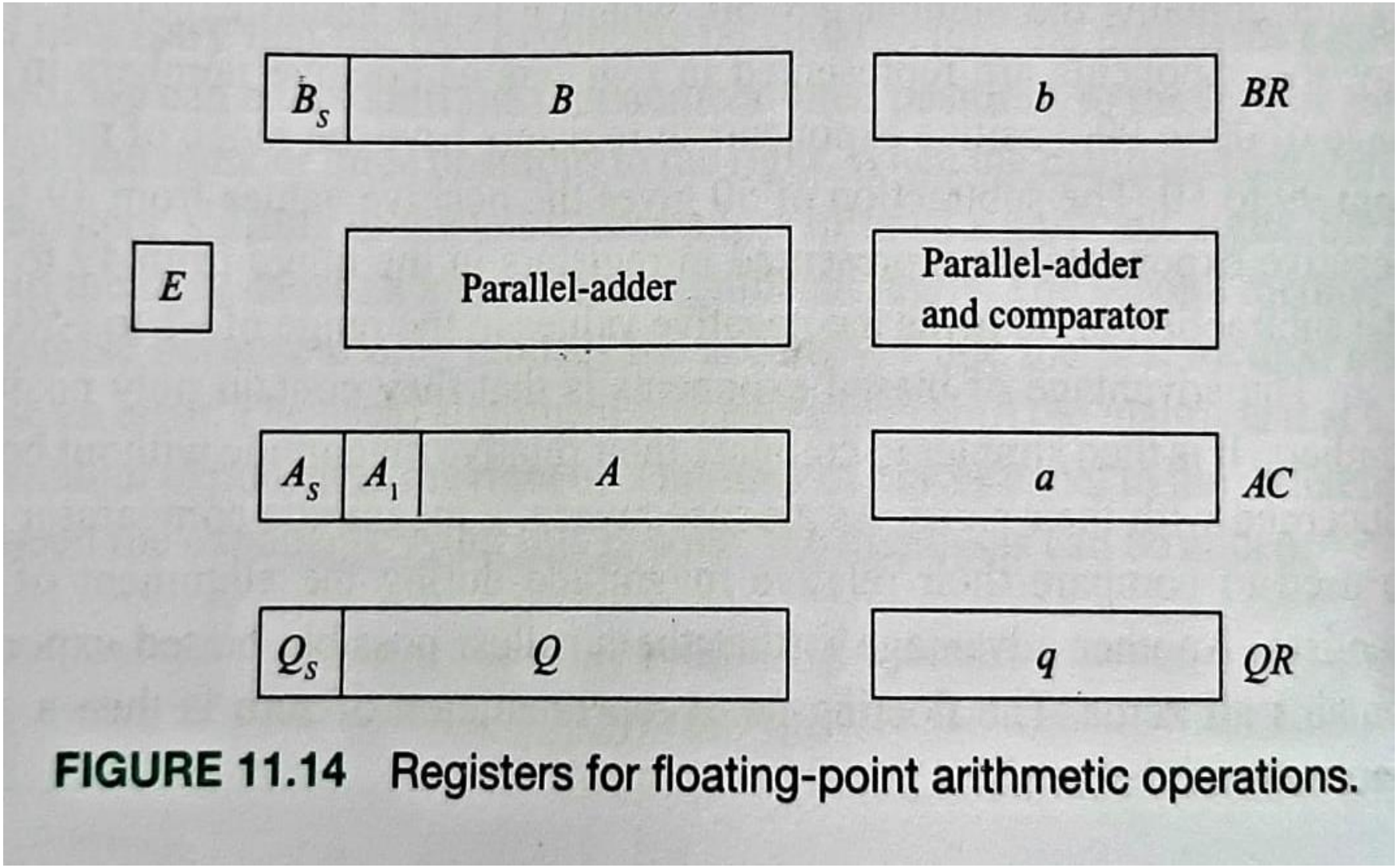


FIGURE 11.14 Registers for floating-point arithmetic operations.

Floating Point Arithmetic operation

Addition and Subtraction : Algorithm for Addition and Subtraction of two numbers is divided into 4 parts:-

1. Check for Zeros
2. Align the mantissa
3. Add or Subtract the mantissas
4. Normalize the result

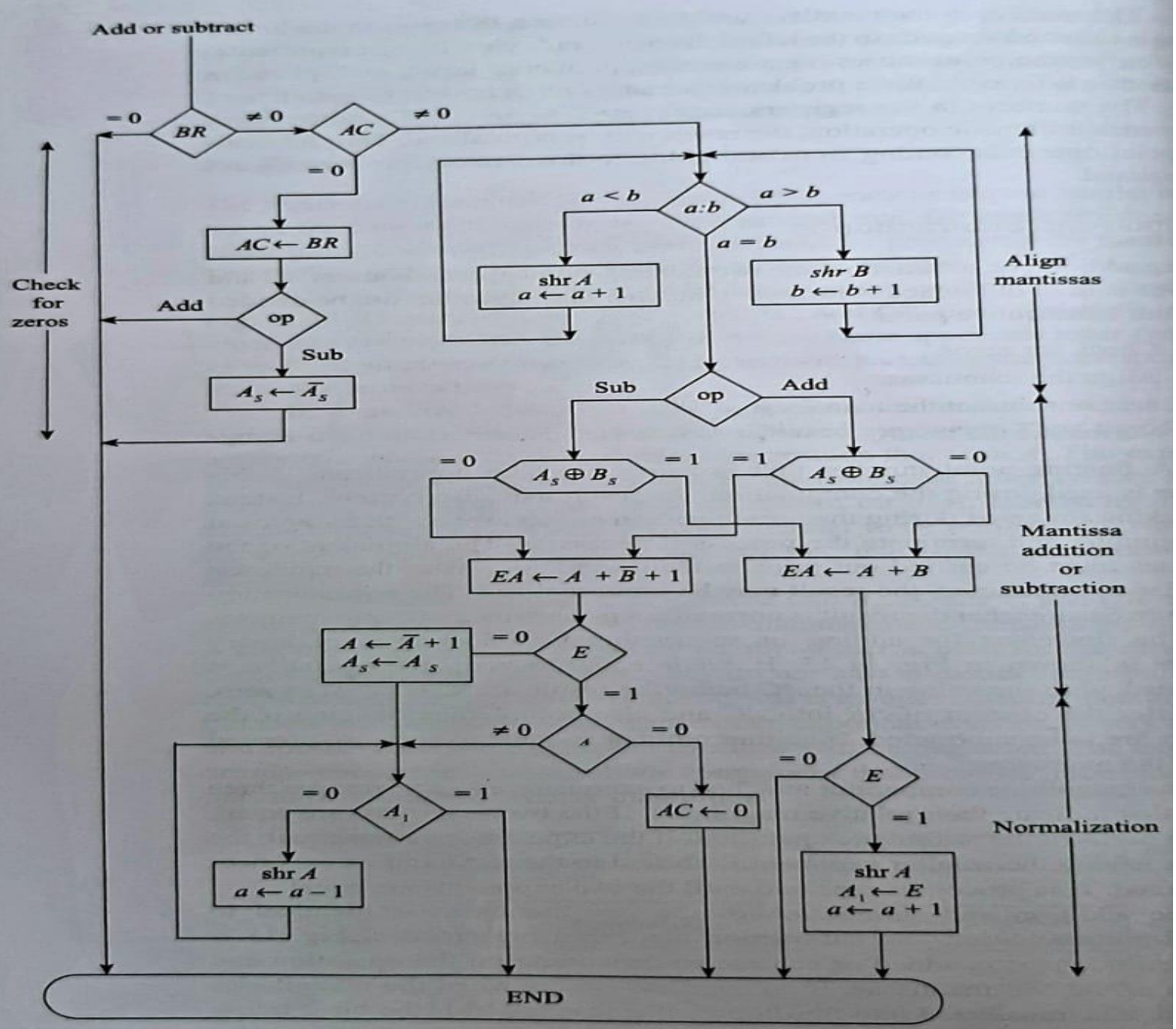
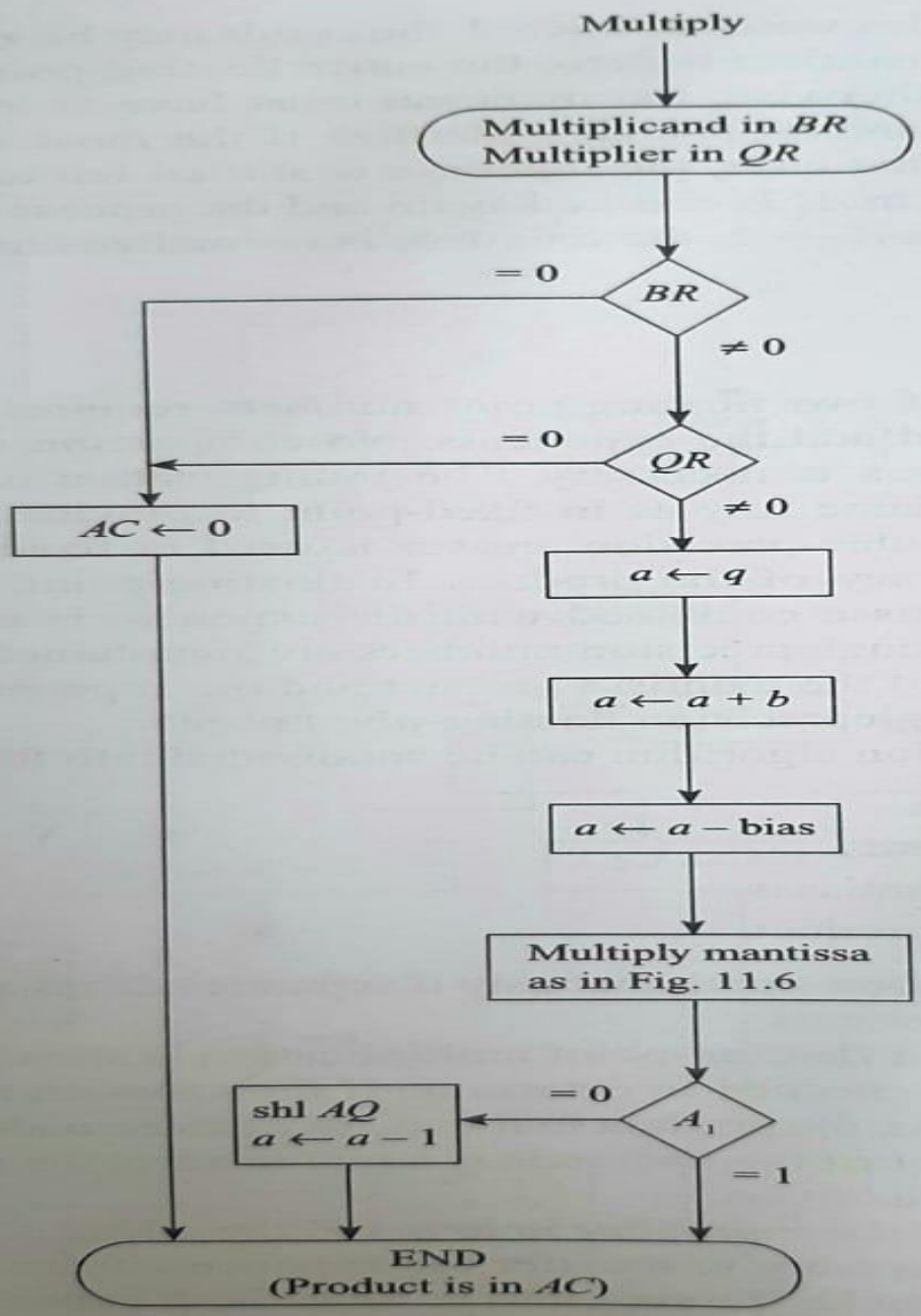


FIGURE 11.15 Addition and subtraction of floating-point numbers.

Floating Point Arithmetic operation

Multiplication : Multiplication of two numbers is divided into 4 parts:-

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



RE 11.16 Multiplication of floating-point numbers.

Floating Point Arithmetic operation

Division : Division of two numbers is divided into 5 parts:-

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

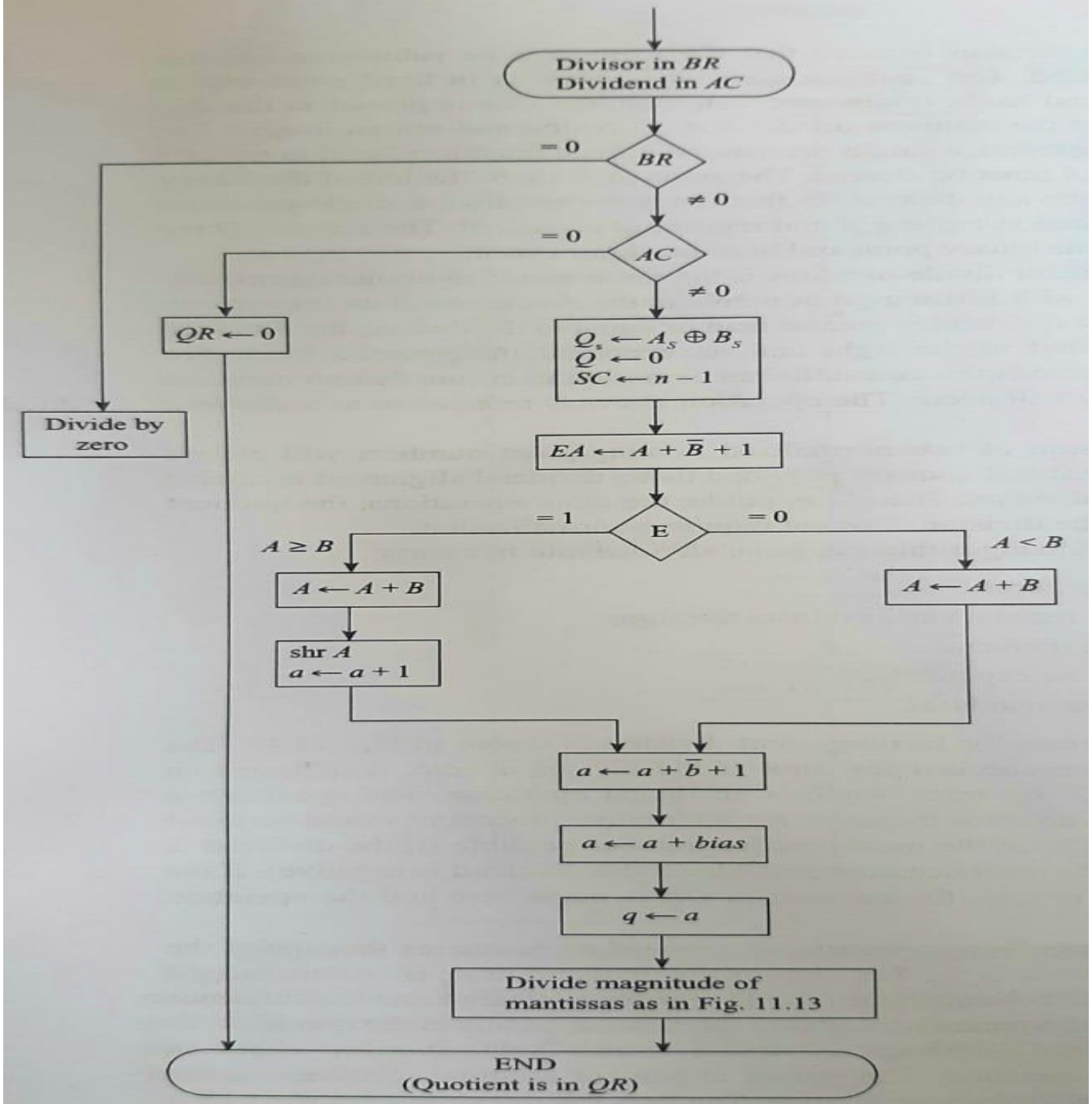


FIGURE 11.17 Division of floating-point numbers.

Arithmetic operation

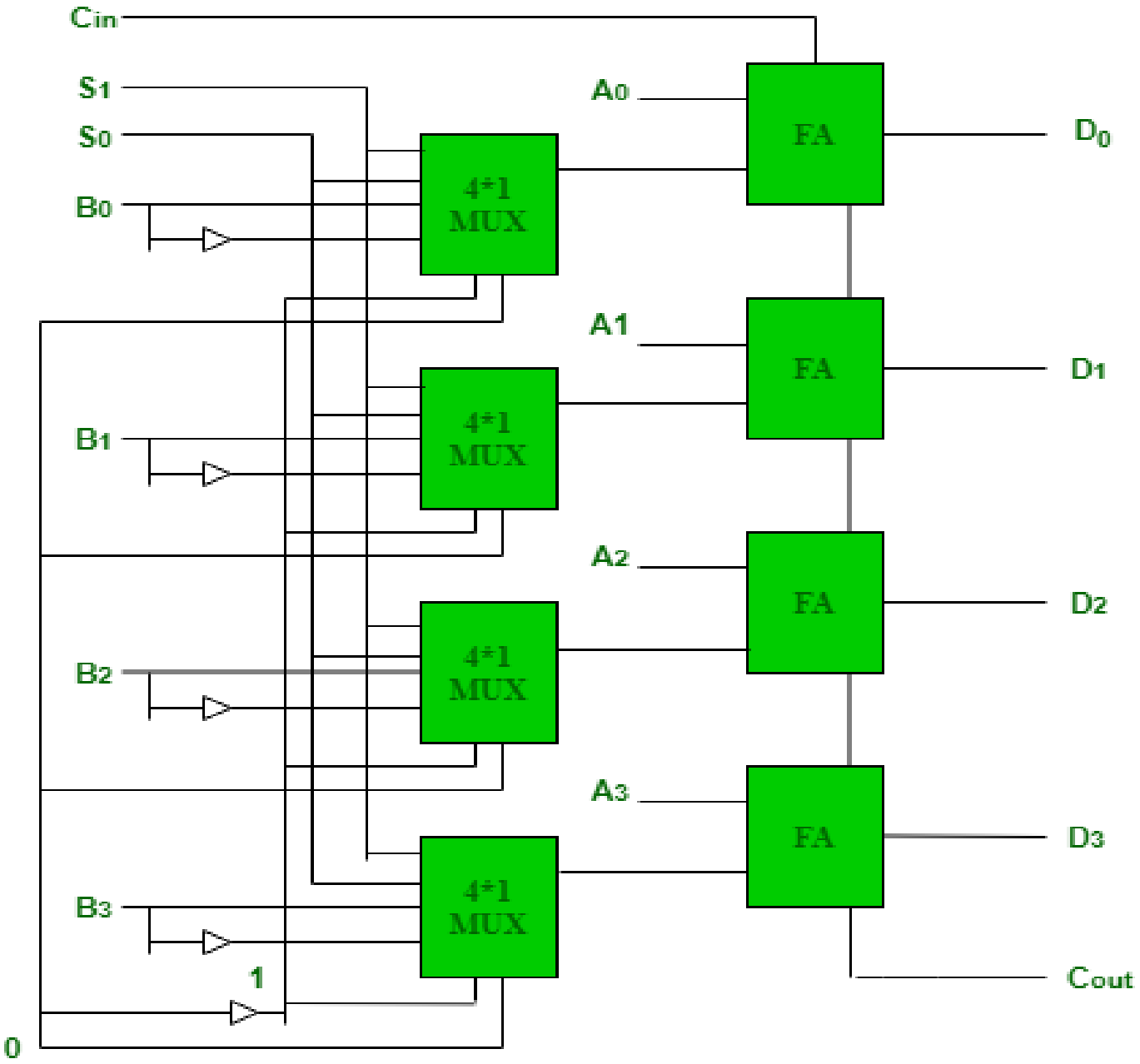
The arithmetic microoperations can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the **parallel adder.** By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations. The output of the binary adder is calculated from the following arithmetic sum:

D =A+ Y + C

TABLE 5.4 Arithmetic Circuit Function Table

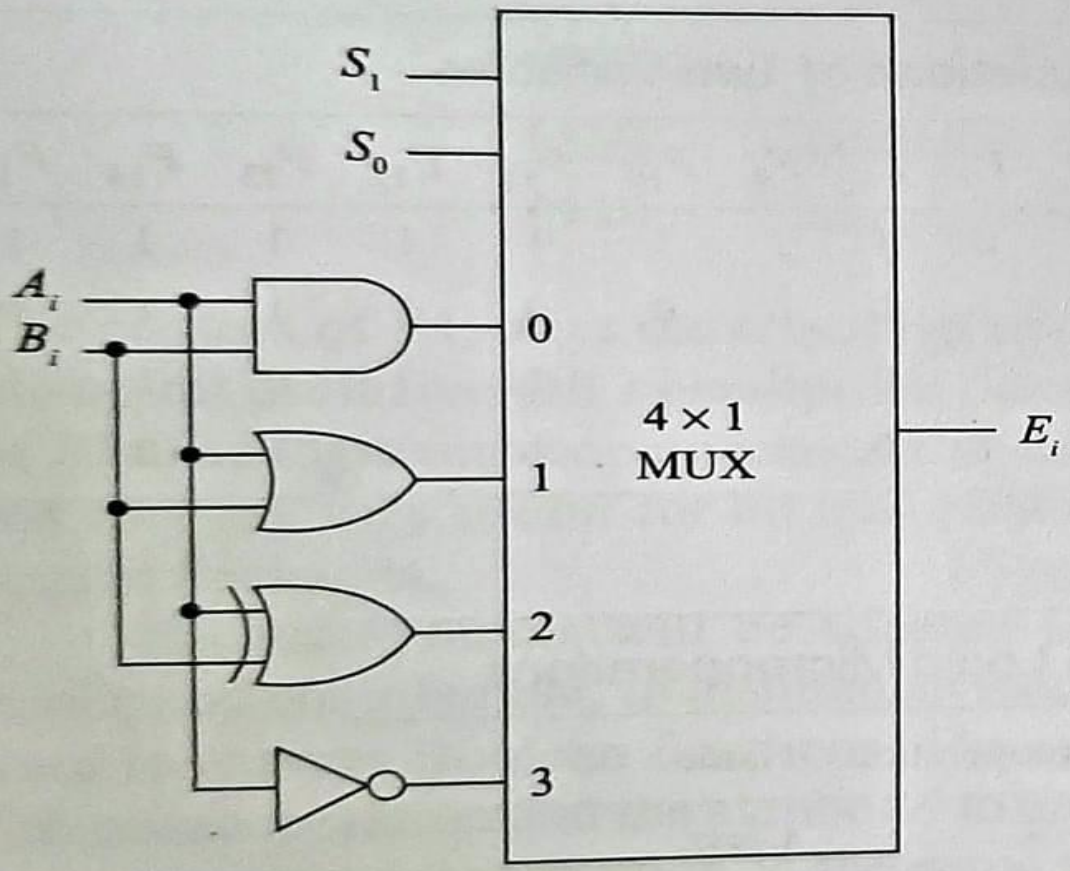
Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

Arithmetic operation



Arithmetic operation

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

FIGURE 5.10 One stage of logic circuit.

Arithmetic operation

Applications of Logic microoperations: - They very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B). In a typical application, register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B

selective-set

1010 A before

1100 B (logic operand)

1110 A after

the exclusive-OR microoperation can be used to selectively complement bits of a register.

selective-clear

1010 A before

1100 B (logic operand)

0010 A after

The corresponding logic microoperation is $A + A /\backslash \overline{B}$

Arithmetic operation

Mask operation

1010 A before

1100 B (logic operand)

1000 A after masking

The mask operation is an AND microoperation

Insert operation

0110 1010 A before

0000 1111 B (mask)

0000 1010 A after masking

and then insert the new value:

0000 1010 A before

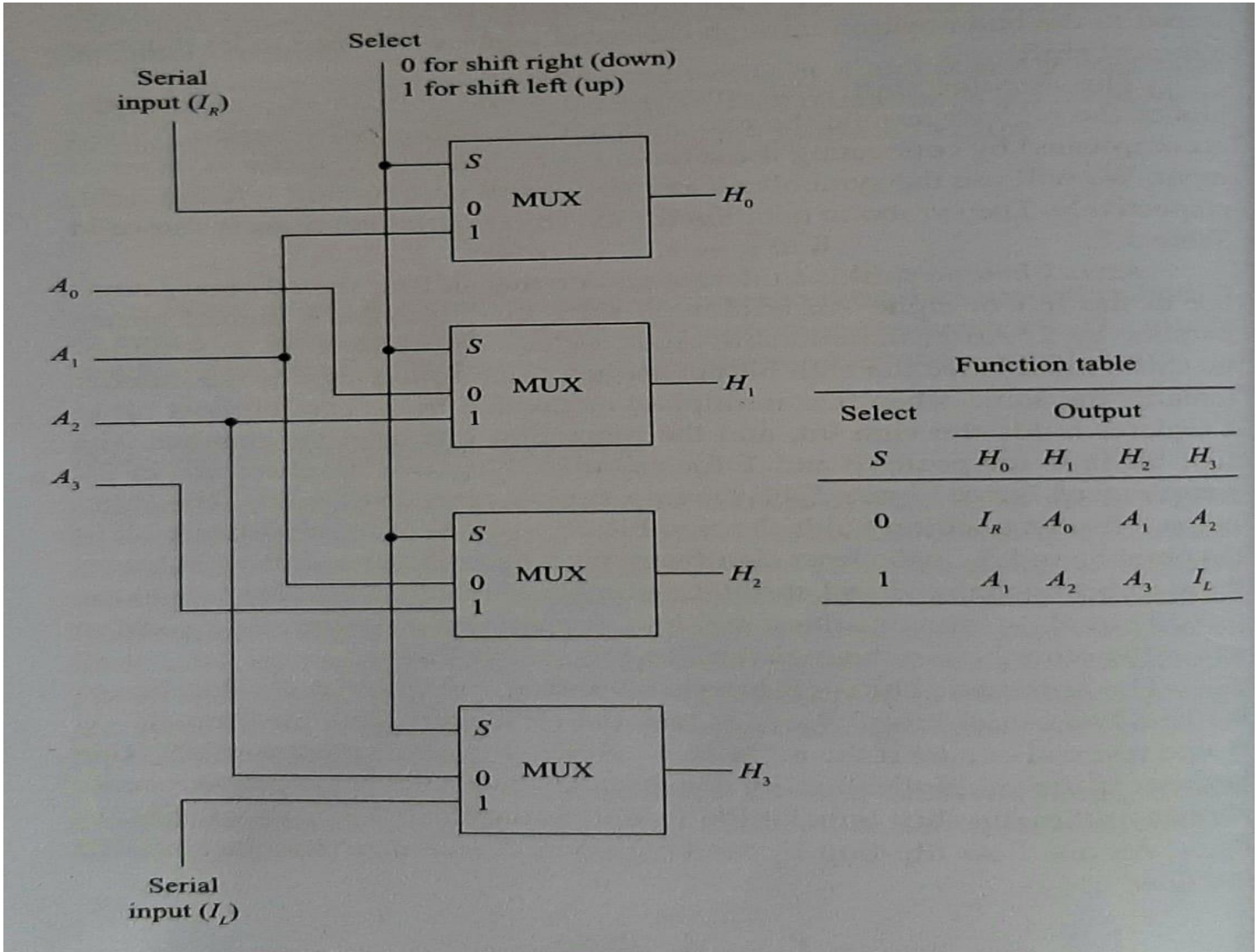
1001 0000 B (insert)

1001 1010 A after insertion

The mask operation is an AND microoperation and the insert operation is an OR microoperation

Arithmetic operation

Shift microoperations :- are used for serial transfer of data. They are also used in conjunction with arithmetic, logic, and other data-processing operations. The contents of a register can be shifted to the left or the right.



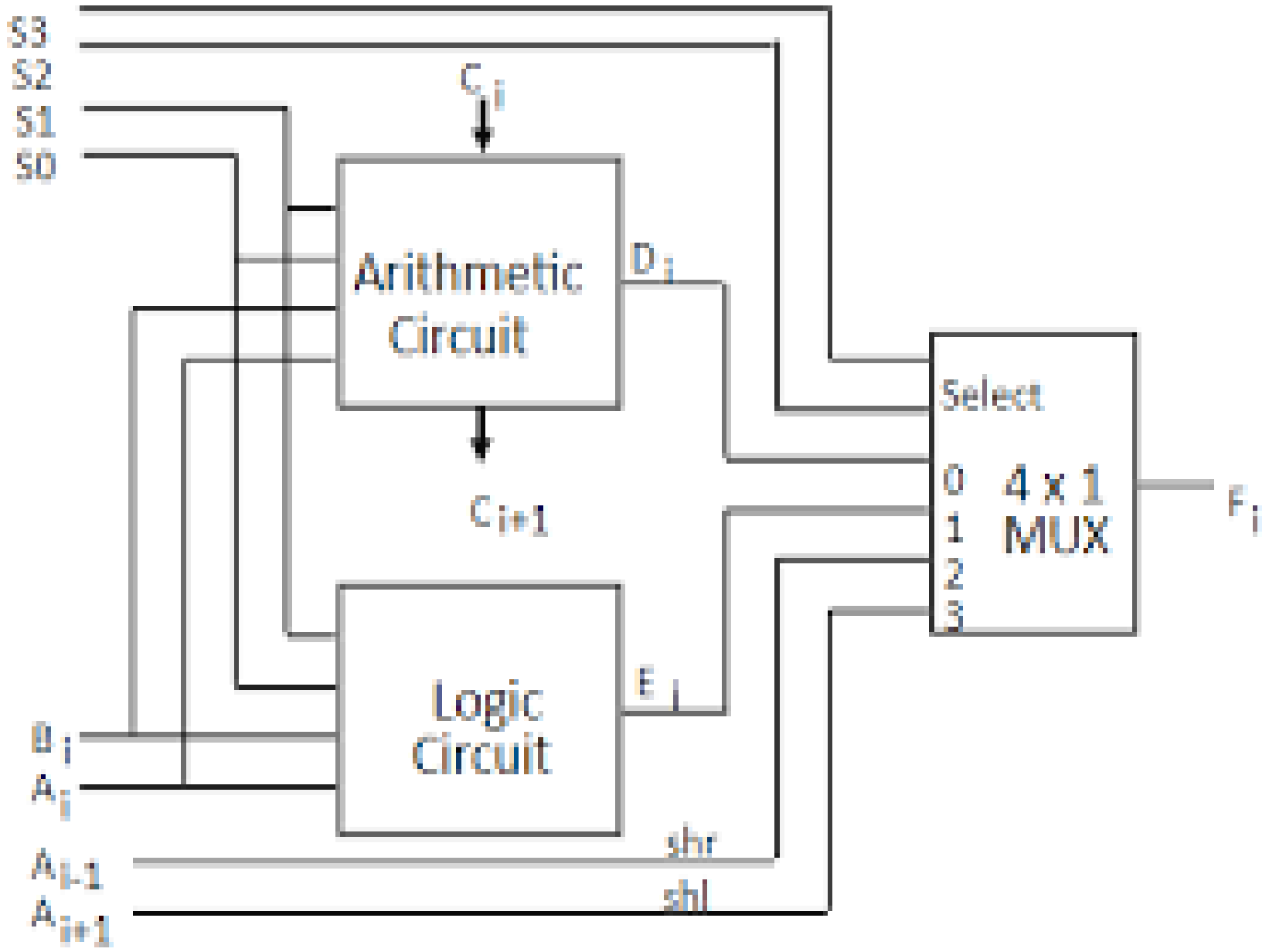
Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

Arithmetic operation

Arithmetic Logic Shift Unit:- Instead of having individual registers performing the microoperations directly, computer systems employ a number of storage registers connected to a common operational unit called an arithmetic logic unit, perform a microoperation, the contents of specified registers are placed in the inputs of the common AW. The AW performs an operation and the result of the operation is then transferred to a destination register. The AW is a combinational source registers circuit so that the entire register transfer operation from the performed through the ALU and into the destination register can be performed during one clock pulse period. The shift microoperations are often overall in a separate unit, but sometimes the shift unit is made part of the ALU.

Arithmetic operation



Arithmetic operation

Operation Select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B'	Subtract with borrow
0	0	1	0	1	F = A + B' + 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A
0	0	1	1	1	F= A	Transfer A
0	1	0	0	×	F = A ^ B	AND
0	1	0	1	×	F = A v B	OR
0	1	1	0	×	F = A XOR B	XOR
0	1	1	1	×	F = A'	Complement A
1	0	×	×	×	F = shr A	Shift right A into F
1	1	×	×	×	F = shl A	Shift left A into F

IEEE Standard for Floating Point Numbers

IEEE Standard for Floating Point No:-

↳ Institute of Electrical & Electronics Engineering, USA formulate a Stard Called IEEE-754 floating Point standard for

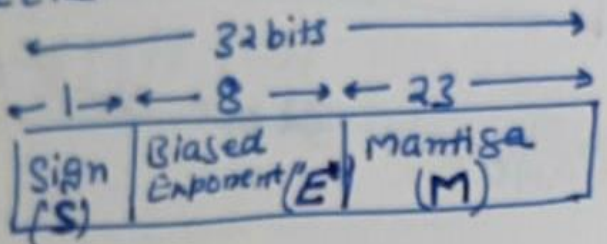
1. Representing floating point No
2. Perform Arithematic Operation in Computer System

The 2 Standards for floating point No are:-

1. Single Precision
2. Double Precision.

IEEE Standard for Floating Point Numbers

1. Single Precision :- It has 32 bits divided into 3 parts



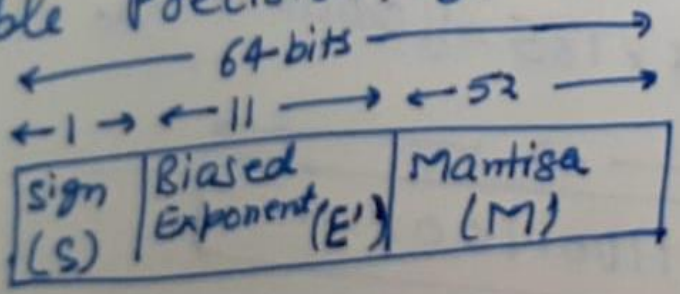
$E' = E + \text{Bias value}$.

Bias value is $127\ (2^7 - 1)$.

$\therefore E' = E + 127$.

It is also called Excess 127 format.

2. Double Precision :- It has 64 bits divided into 3 parts.



$E' = E + \text{Bias value}$

Bias value is 1023

$E = E + 1023$

It is also called Excess 1024 format.

IEEE Standard for Floating Point Numbers

Q: Represent $(1259.125)_{10}$ in 32 bit and 64 bit Representation

$\downarrow$

$(10011101011.001)_2$

$\Rightarrow 1.0011101011001 \times 2^{10}$

(a) **32-bit**

$E = 10$

$E' = E + 127 = 10 + 127 = (137)_{10} = (10001001)_2$

← 1 →	← 8 →	← 23 →
0	10001001	00111010 11001 ...

(b) **64-bit**

$E = 10$

$E' = E + 1023 = 10 + 1023 = (1024)_{10} = (10000001001)_2$

← 1 →	← 11 →	← 52 →
0	10000001001	00111010 11001 ...

IEEE Standard for Floating Point Numbers

Q. Represent $(-307.1875)_{10}$ in 32 bit and 64 bit Repr...

$\downarrow$

$(100110011.0011)_2$

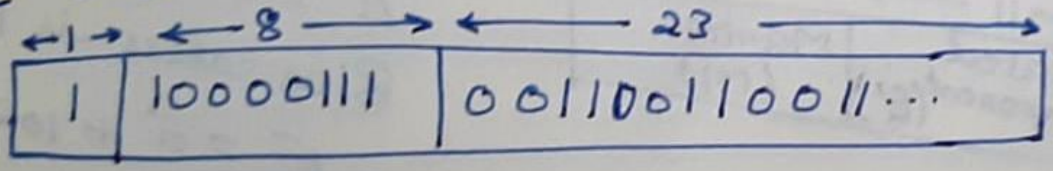
$\downarrow$

$1.00110011 0011 \times 2^8$

(a) 32-bit

$E = 8$

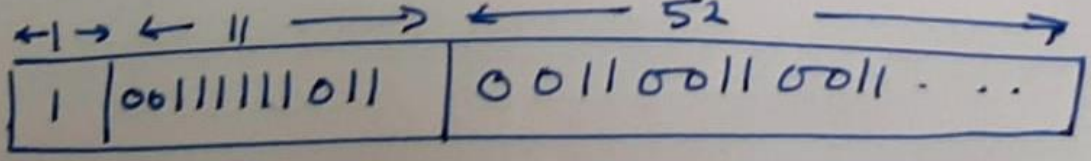
$E' = E + 127 = 8 + 127 = 135 = (10000111)_2$



(b) 64-bit

$E = 8$

$E' = E + 1023 = 8 + 1023 = (1031)_{10} = 11111111011$



IEEE Standard for Floating Point Numbers

Q: $(.0625)_{10}$ Represent in 32 & 64 bit Repr.

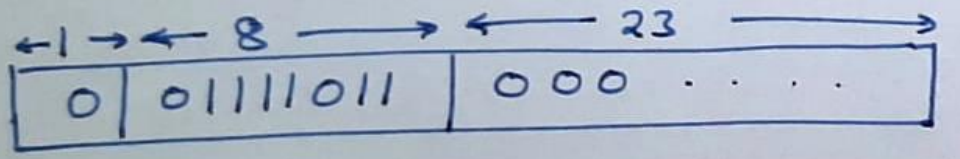
$(.0001)_2$

1.0×2^{-4}

(a) **32 bit**

$E = -4$

$E' = E + 127 = -4 + 127 = (123)_{10} = (1111011)_2$



(b) **64- bit**

$E = -4$

$E' = E + 1023 = -4 + 1023 = (1019)_{10} = (1111111011)_2$