UNIT 3

Dr. S. Anbukkarasi
AP/CSE

ALU

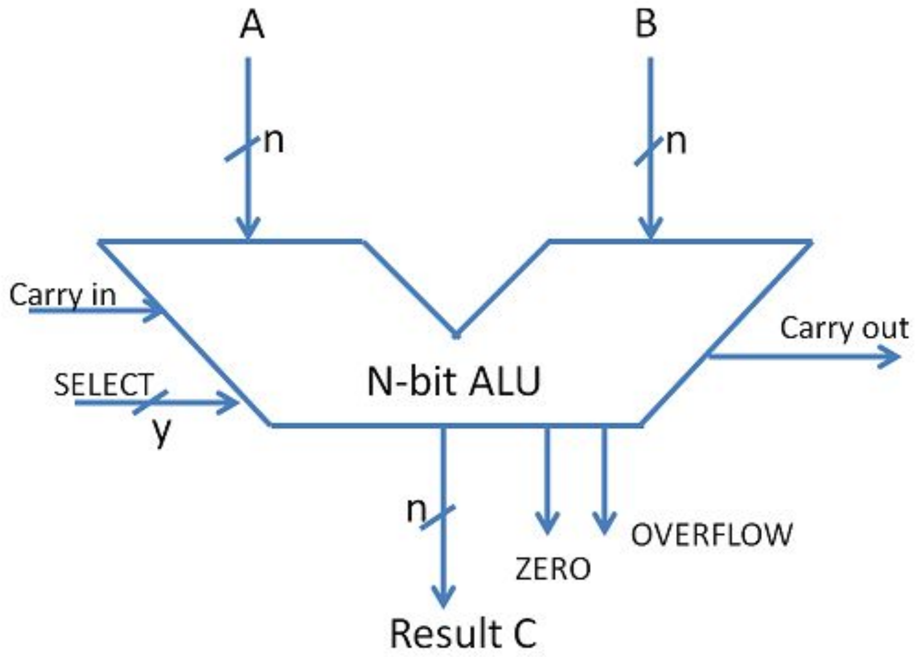
- ALU is the heart of any Central Processing Unit.
- A simple ALU is constructed with Combinational circuits.
- It is a digital circuit to do arithmetic operations like addition, subtraction, multiplication and division.
- Logical operations such as OR, AND, NOR etc.
- Data movement operations such as LOAD and STORE.
- Complex ALUs are designed for executing Floating point, decimal operations and other complex numerical operations. These are called as Co-processor and work in tandem with the main processor.
- The design specifications of ALU are derived from the Instruction set Architecture.
- The ALU must have the capabilities to execute the instructions of ISA.
- Modern CPUs have multiple ALU to improve the efficiency.

ALU includes the following Configurations :

- Instruction Set Architecture
- Accumulator
- Stack
- Register – Register architecture
- Register – Stack architecture
- Register – memory architecture

The size of input quantities of ALU is referred as **word length** of a computer

ALU Symbol


A
n
B
n
Carry in
SELECT
y
N-bit ALU
Carry out
n
ZERO
OVERFLOW
Result C

Gates Boolean Algebra

AND => C (output) = $A . B$

OR => C (output) = $\overline{A+B}$

NAND => C (output) = $\overline{A. B}$

NOR => C (output) = $\overline{A}+B$

XOR => C (Output) = $\overline{A}B + A\overline{B}$

XNOR => C (Output) = $\overline{A}B + AB$ (Complement of XOR)

ADDERS

The basic building blocks of an ALU in digital computers are Adders.

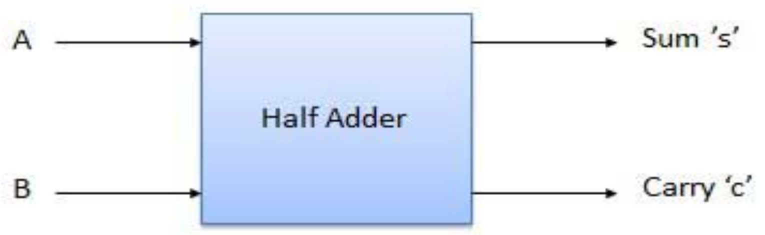
Types of Basic Adders are

- Half Adder
- Full Adder

- Parallel adders are nothing but cascade of full adders. The number of full adders used will depend on the number of bits in the binary digit which require to be added.
- Ripple carry adders are used when the input sequence is large. It is used to add two n-bit binary numbers.
- Carry look ahead adder is an improved version of Ripple carry adder.
- It generates the carry-in of each full adder simultaneously without causing any delay.

Half Adder

- Half adder is a combinational logic circuit with two input and two output. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of two single bit numbers. This circuit has two outputs, carry and sum.



Truth Table and Circuit Diagram

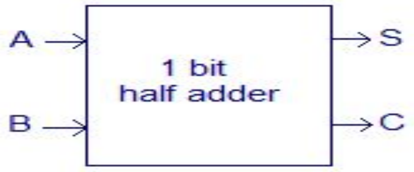
Inputs		Outputs	
A	B	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Truth table

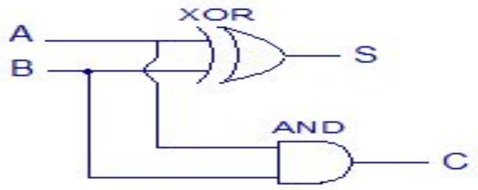
SUM $S = A.\overline{B} + \overline{A}.B$

CARRY $C = A.B$

Boolean Expression



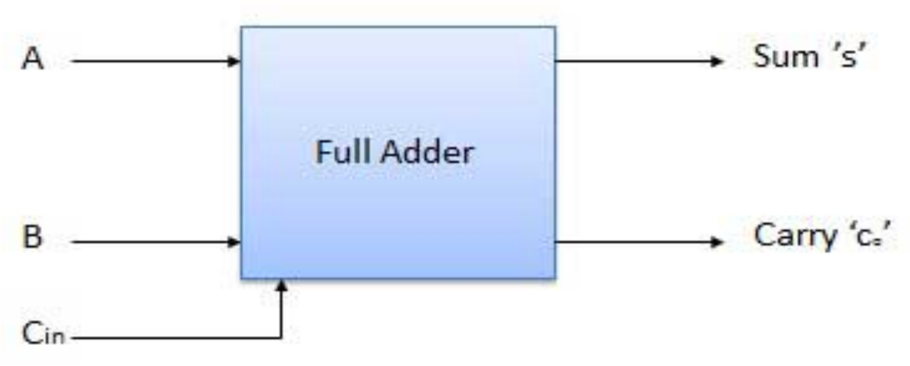
Schematic



Realization

Full Adder

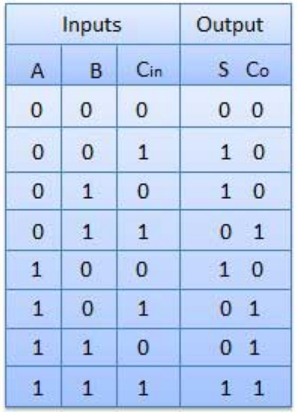
- Full adder is developed to overcome the drawback of Half Adder circuit. It can add two one-bit numbers A and B, and carry c. The full adder is a three input and two output combinational circuit.



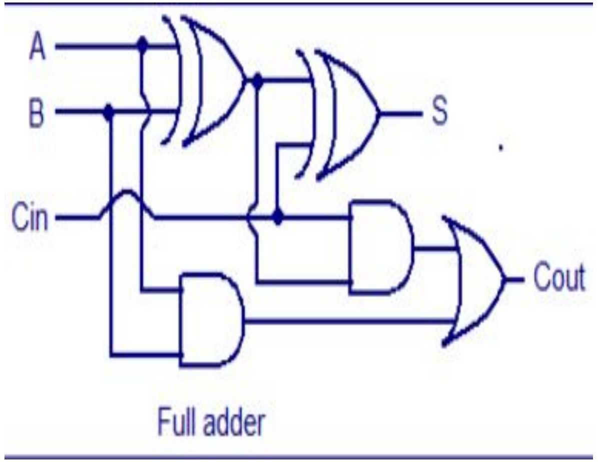
Truth Table and Circuit Diagram

$$
\begin{aligned}
S &= A\overline{B}\,\overline{C} + \overline{A}\,\overline{B}C + ABC + \overline{A}B\overline{C} \\
&= C\,(AB + \overline{A}\,\overline{B}) + \overline{C}\,(\overline{A}B + A\,\overline{B}) \\
&= C\,(\overline{\overline{A}B + A\,\overline{B}}) + \overline{C}\,(\overline{A}B + A\,\overline{B}) \\
&= C\,(\overline{A \oplus B}) + \overline{C}\,(A \oplus B) \;= A \oplus B \oplus C
\end{aligned}
$$

$$
\begin{aligned}
C_{out} &= \overline{A}BC + A\overline{B}C + AB\overline{C} + ABC \\
&= (\overline{A}B + A\overline{B})\,C + AB\,(\overline{C} + C) \\
&= (A \oplus B).C + AB.
\end{aligned}
$$



Inputs			Output	
A	B	Cin	S	Co
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

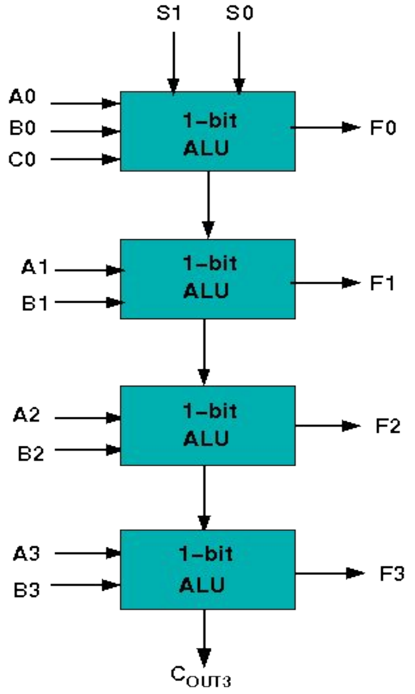


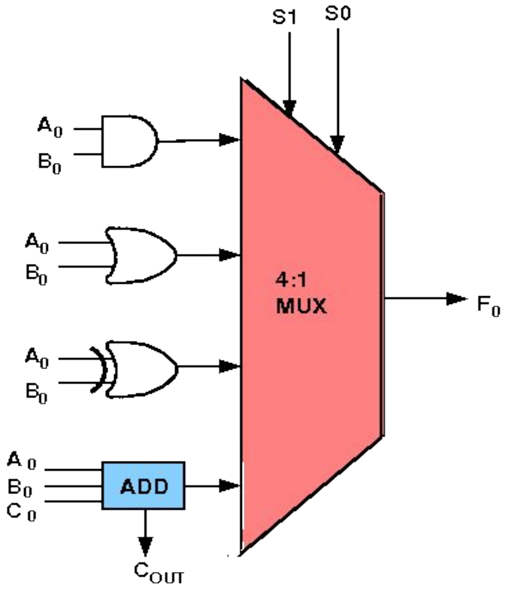
Design of ALU

- ALU or Arithmetic Logical Unit is a digital circuit to do arithmetic operations like addition, subtraction, division, multiplication and logical operations like AND, OR, XOR, NAND, NOR etc.

- A simple block diagram of a 4 bit ALU for operations AND, OR,XOR and ADD is shown here :

The 4-bit ALU block is combined using 4 1-bit ALU block





Design Issues

The circuit functionality of a 1 bit ALU is shown here, depending upon the control signal S_1 and S_0 the circuit operates as follows:

for Control signal $\mathbf{S_1 = 0}$, $S_0 = 0$, the output is **A And B**,
for Control signal $\mathbf{S_1 = 0}$, $S_0 = 1$, the output is **A Or B**,
for Control signal $\mathbf{S_1 = 1}$, $S_0 = 0$, the output is **A Xor B**,
for Control signal $\mathbf{S_1 = 1}$, $S_0 = 1$, the output is **A Add B**.

16 bit ALU Design

Design the 16-bit Arithmetic Logic Unit (ALU) shown in Figure 1. The function table of the ALU is shown in Table 1.

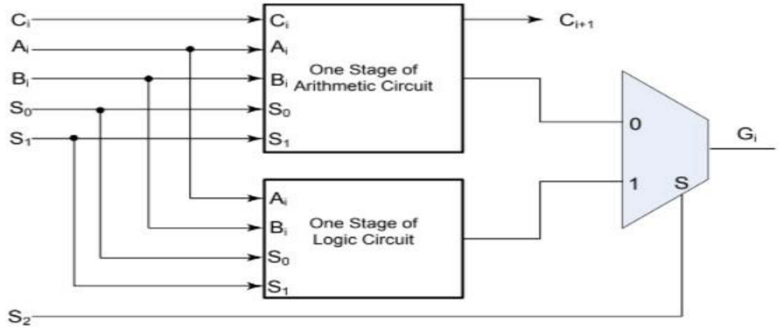


Figure 1: 16-bit ALU

Table 1: Function table of the ALU

Operation Select					
S_2	S_1	S_0	C_{in}	**Operation**	**Function**
0	0	0	0	$G = A$	Transfer A
0	0	0	1	$G = A + 1$	Increment A
0	0	1	0	$G = A + B$	Addition
0	0	1	1	$G = A + \underline{B} + 1$	Add with carry input of 1
0	1	0	0	$G = A + \underline{B}$	A plus 1's complement of B
0	1	0	1	$G = A + \bar{B} + 1$	Subtraction
0	1	1	0	$G = A - 1$	Decrement A
0	1	1	1	$G = A$	Transfer A
1	0	0	X	$G = A \wedge B$	AND
1	0	1	X	$G = A \vee B$	OR
1	1	0	X	$G = \underline{A} \oplus B$	XOR
1	1	1	X	$G = A$	NOT (1's complement)

Two Theorems

- Theorem1

$$\overline{A.B} = \overline{A} + \overline{B}$$

NAND = Bubbled OR

- The left hand side (LHS) of this theorem represents a NAND gate with inputs A and B, whereas the right hand side (RHS) of the theorem represents an OR gate with inverted inputs.
- This OR gate is called as **Bubbled OR**.

Theorem1 – Logic diagram

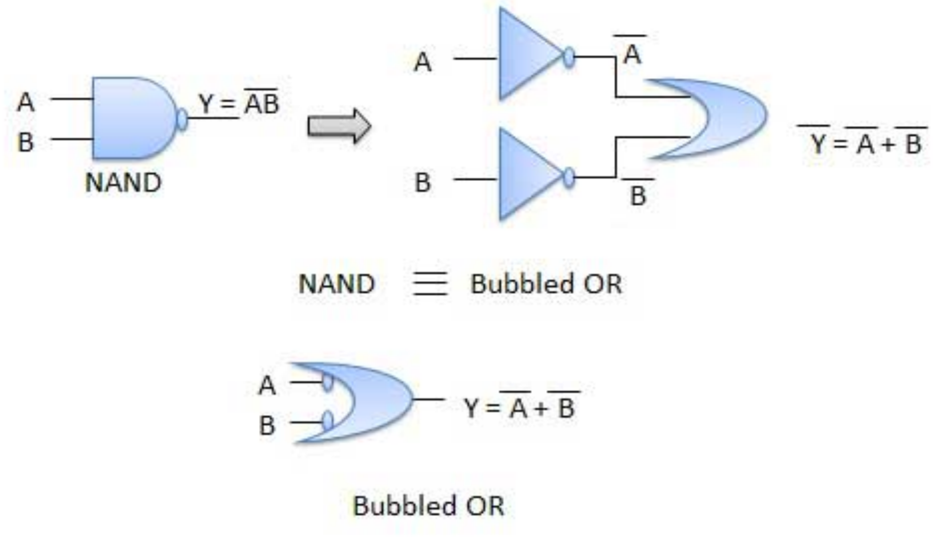
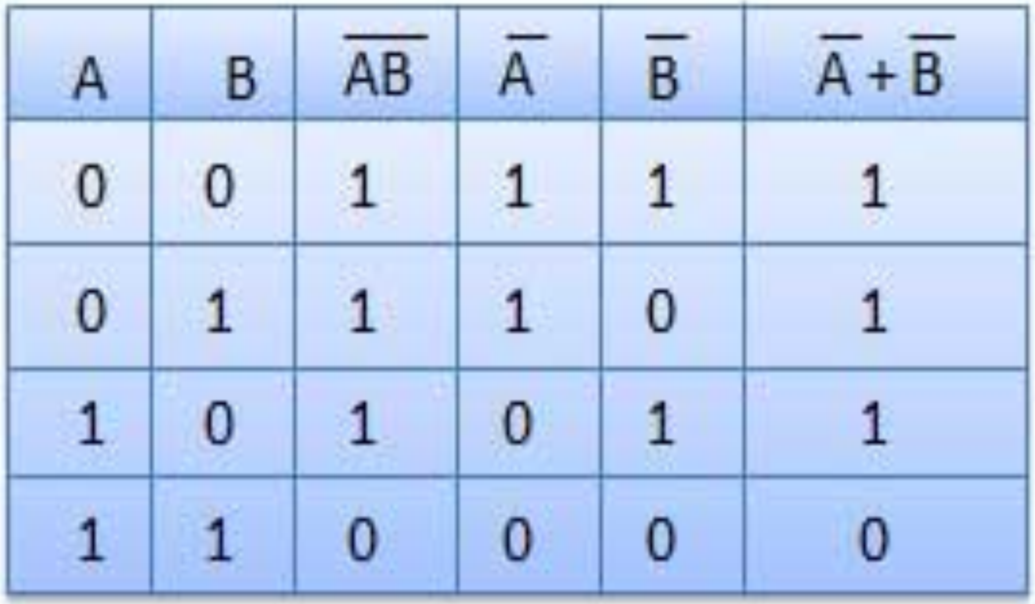


Table showing verification of the De Morgan's first theorem



A	B	$\overline{AB}$	$\overline{A}$	$\overline{B}$	$\overline{A}+\overline{B}$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

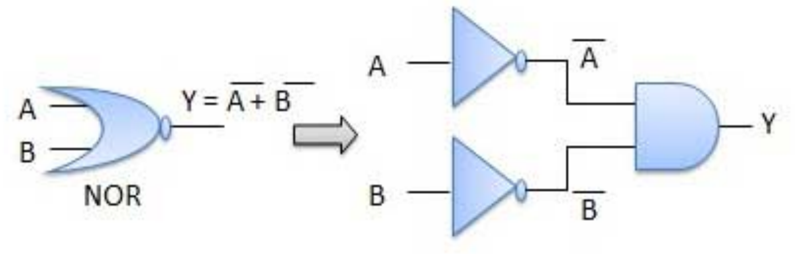
Theorem 2

$$\overline{A + B} = \overline{A} . \overline{B}$$

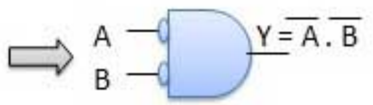
NOR = Bubbled AND

- The LHS of this theorem represents a NOR gate with inputs A and B, whereas the RHS represents an AND gate with inverted inputs.
- This AND gate is called as **Bubbled AND**.

Theorem 2- Logic diagram

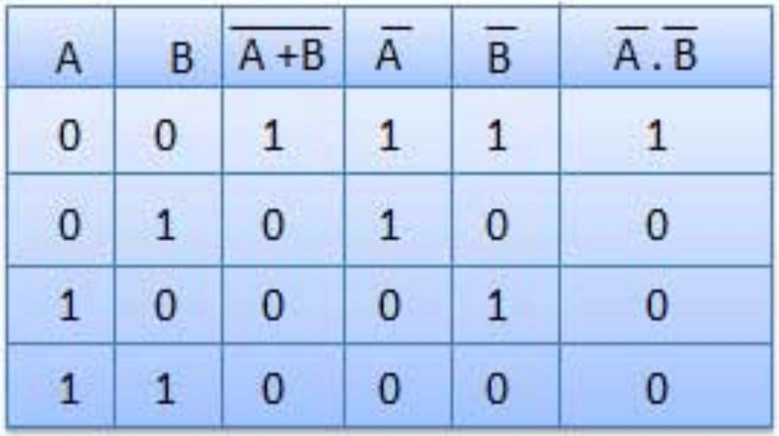


NOR ≡ Bubbled AND

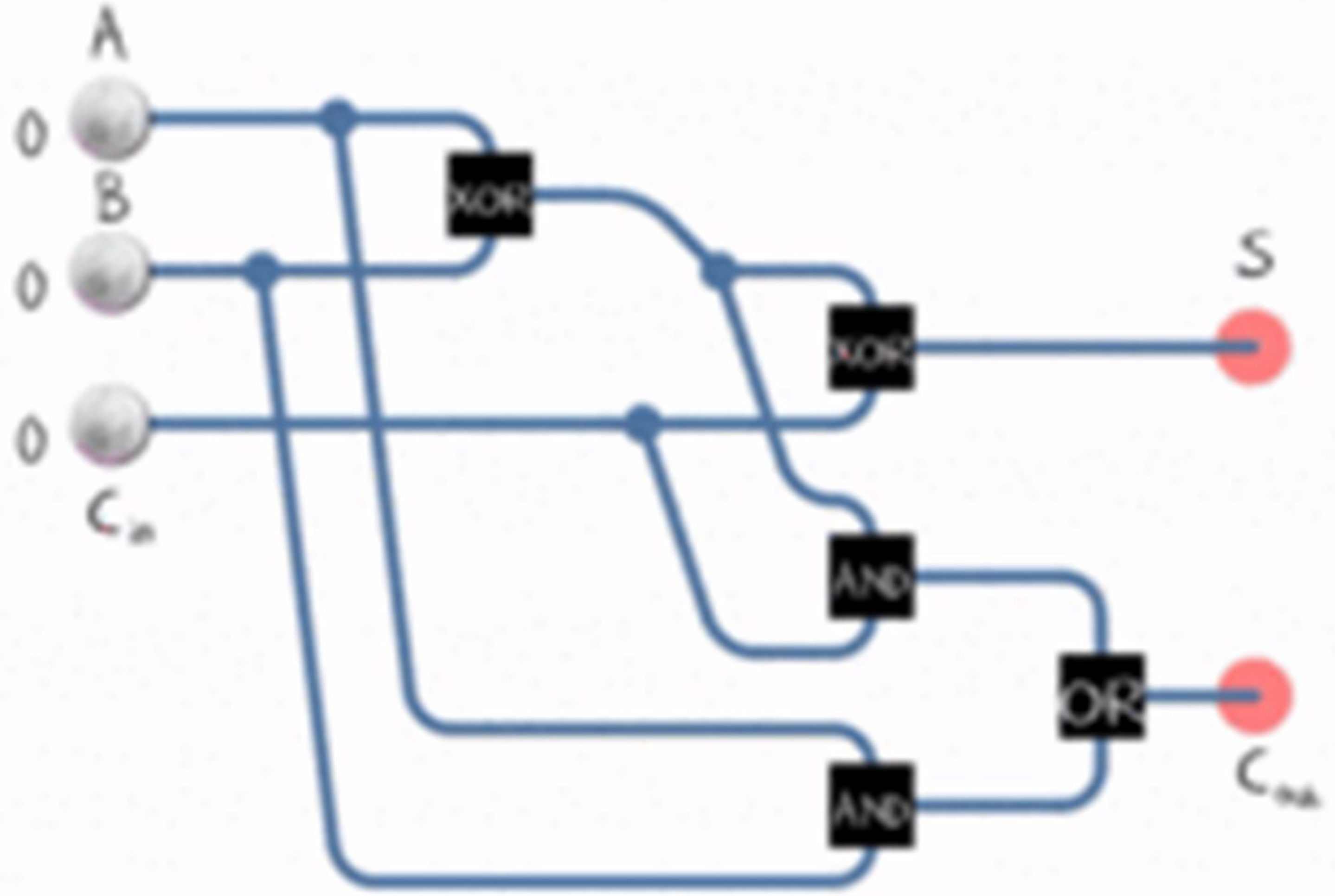


Bubbled AND

Table showing verification of the De Morgan's second theorem



A	B	$\overline{A+B}$	$\overline{A}$	$\overline{B}$	$\overline{A}.\overline{B}$
0	0	1	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	0


A
B
0
0
0
C_in
XOR
XOR
AND
AND
OR
S
C_out

Applications of De Morgan’s Theorem

- In the domain of engineering, using De Morgan’s laws, Boolean expressions can be built easily only through one gate which is usually NAND or NOR gates. This results in hardware design at a cheaper cost.
- Used in the verification of SAS code.
- Implemented in computer and electrical engineering domain.
- De Morgan’s laws are also employed in Java programming.

Ripple Carry Adder

Typical Ripple Carry Addition is a Serial Process:

- Addition starts by adding LSBs of the augend and addend.
- Then next position bits of augend and addend are added along with the carry (if any) from the preceding bit.
- This process is repeated until the addition of MSBs is completed.
- Speed of a ripple adder is limited due to carry propagation or carry ripple.
- Sum of MSB depends on the carry generated by LSB.

Example: 4-bit Carry Ripple Adder

- Assume to add two operands A and B where

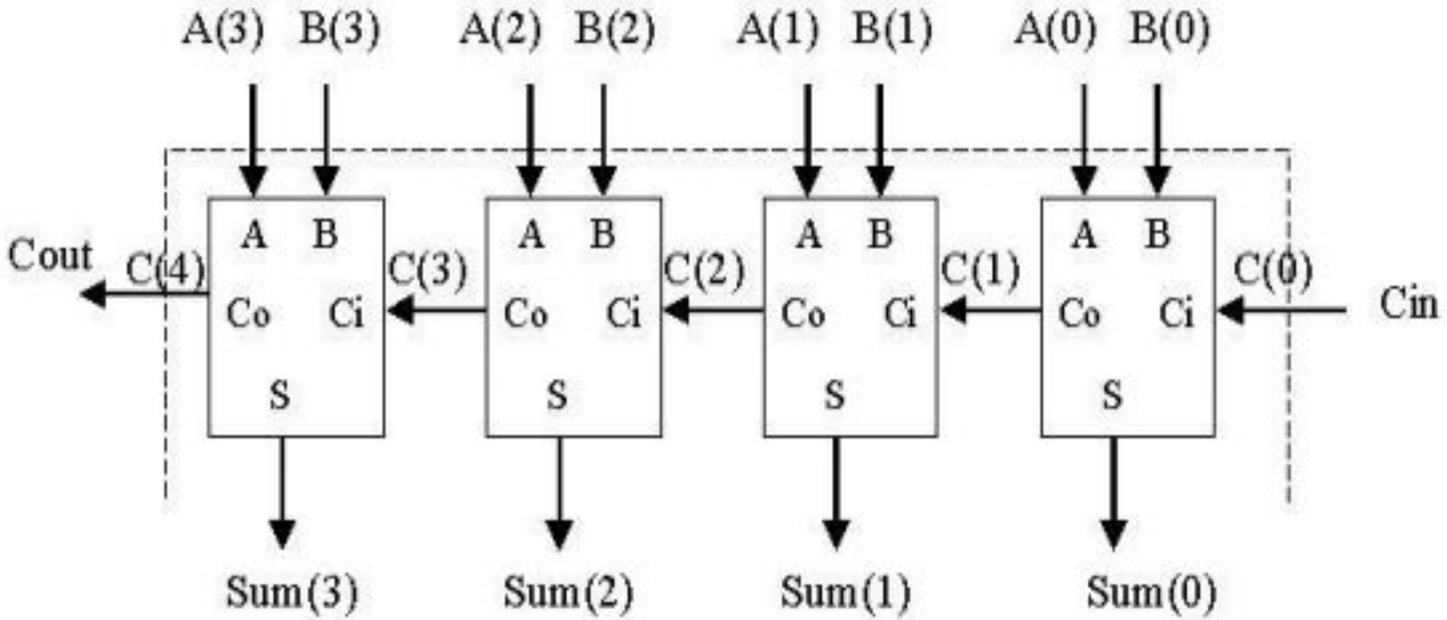
  A = A3 A2 A1 A0

  B = B3 B2 B1 B0

```
  A=   1    0     1   1 +
  B =  1    1     0   1
        ---------------------
A+B=1 1    0     0    0
 Cout   S3  S2   S1  S0
```

Carry Propagation

- From the above example it can be seen that we are adding 3 bits at a time sequentially until all bits are added.
- *A* full adder is a combinational circuit that performs the arithmetic sum of three input bits: augends *Ai*, addend Bi and carry in *Cin* from the previous adder.
- Its result contain the sum Si and the carry out, Cout to the next stage.
-


A(3)
B(3)
A(2)
B(2)
A(1)
B(1)
A(0)
B(0)
A
B
Co
Ci
S
Cout
C(4)
C(3)
C(2)
C(1)
C(0)
Cin
Sum(3)
Sum(2)
Sum(1)
Sum(0)

Ripple Carry Adder:-

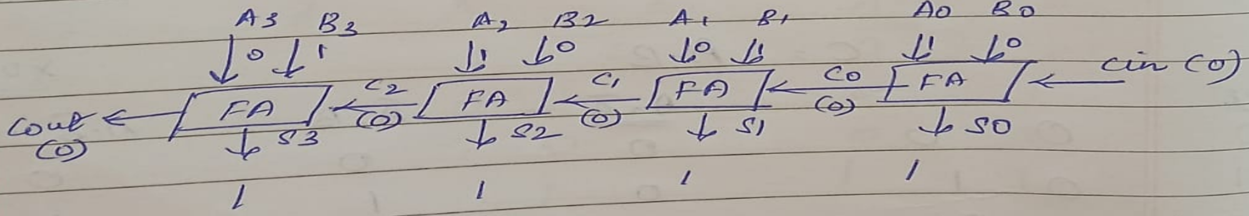
Ex:-

A = 0101 , B = 1010 , Cin = 0 (initial condition)

A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0
0	1	0	1	1	0	1	0

```
 0101  <= 5
 1010  <= 10
 ----     --
 1111     15
S3 S2 S1 S0
```

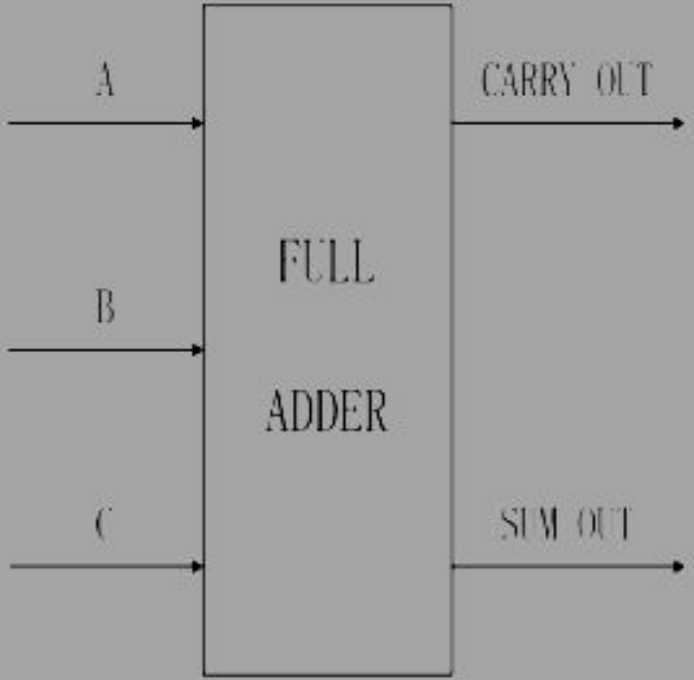
$C_0, C_1, C_2, C_3 \Rightarrow 0$

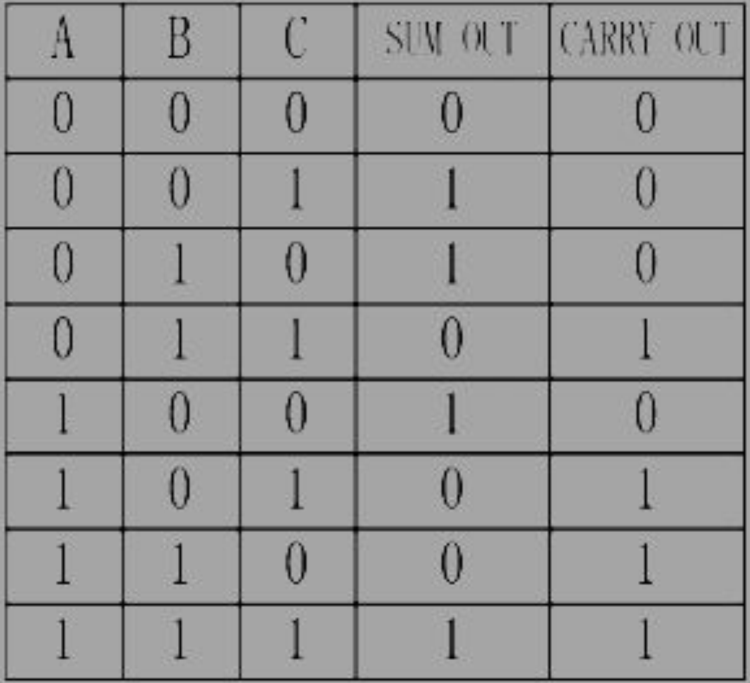


4-bit Adder

- A 4-bit adder circuit can be designed by first designing the 1-bit full adder and then connecting the four 1-bit full adders to get the 4-bit adder as shown in the diagram above.
- For the 1-bit full adder, the design begins by drawing the Truth Table for the three input and the corresponding output SUM and CARRY.
- The Boolean Expression describing the binary adder circuit is then deduced.
- The binary full adder is a three input combinational circuit which satisfies the truth table given below.

Full Adder

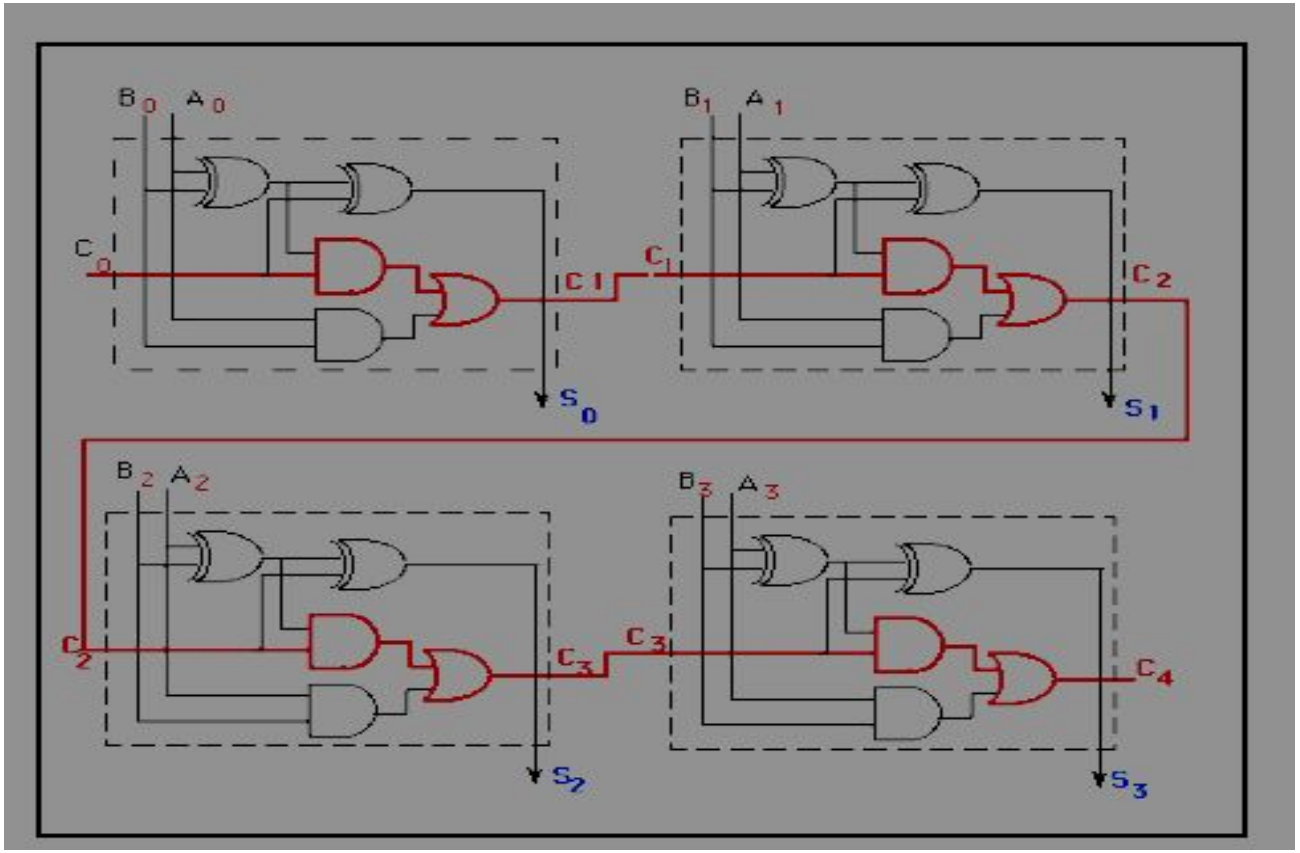




A	B	C	SUM OUT	CARRY OUT
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Fig.2. Diagram and Truth Table of Full Adder

4 bit Adder



Design of Fast adder: Carry Look-ahead Adder

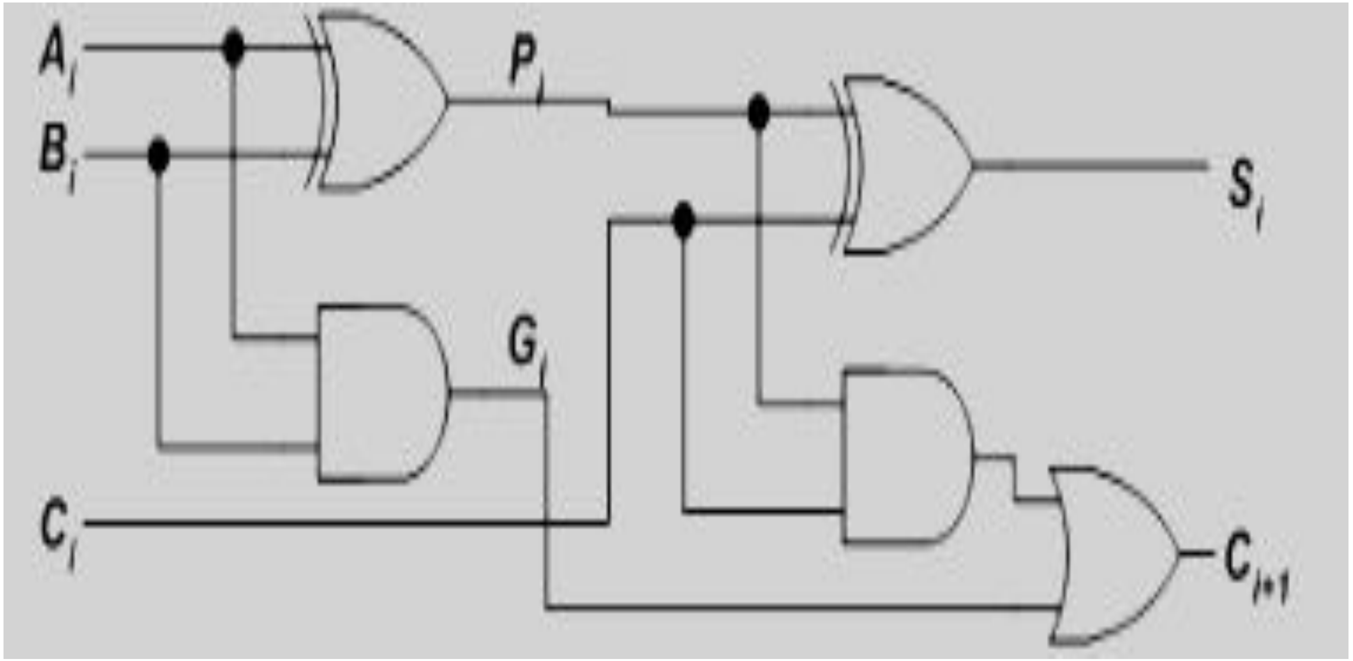
- A carry-look ahead adders (CLA) is a type of adder used in digital logic. A carry-look ahead adder improves speed by reducing the amount of time required to determine carry bits.
- It can be contrasted with the simpler, but usually slower, ripple carry adder for which the carry bit is calculated alongside the sum bit, and each bit must wait until the previous carry has been calculated to begin calculating its own result and carry bits (see adder for detail on ripple carry adders)

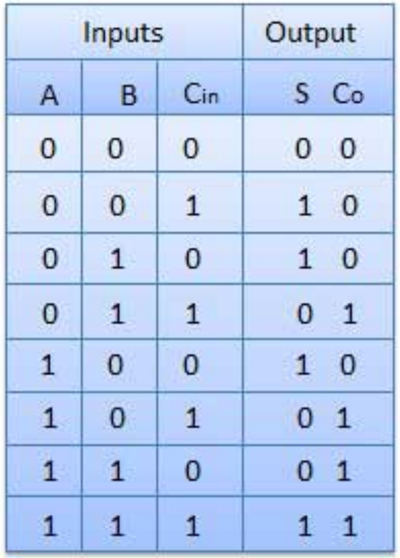
- The carry-look ahead adder calculates one or more carry bits before the sum, which reduces the wait time to calculate the result of the larger value bits.
- In a ripple adder the delay of each adder is 10 ns , then for 4 adders the delay will be 40 ns.
- To overcome this delay Carry Look-ahead Adder is used.

Carry Look-ahead Adder

- Different logic design approaches have been employed to overcome the carry propagation delay problem of adders.
- One widely used approach employs the principle of carry look-ahead solves this problem by calculating the carry signals in advance, based on the input signals.
- This type of adder circuit is called as carry look-ahead adder (CLA adder). A carry signal will be generated in two cases:
    - (1) when both bits Ai and Bi are 1, or
    - (2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.

The Figure shows the full adder circuit used to add the operand bits in the Ith column; namely Ai & Bi and the carry bit coming from the previous column (Ci).





Inputs			Output	
A	B	C_{in}	S	C_o
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

$$C_0 = A.B + (A \oplus B) . C_{in}$$

$G \Rightarrow A.B \Rightarrow$ Carry Generator

$P \Rightarrow (A \oplus B) \Rightarrow$ Carry Propagator

$$C_0 = G + PC_{in}$$

$$C_i = G_i + P_iC_{i-1}$$

$\underline{\underline{C_i = G_i + P_i C_{i-1}}}$

If $i=0$

$\underline{\underline{C_0 = G_0 + P_0 C_{-1}}}$ — ①

If $i=1$,

$C_1 = G_1 + P_1 \underline{\underline{C_0}}$

$= G_1 + P_1(G_0 + P_0 C_{-1})$

$\underline{\underline{C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{-1}}}$ — ②

If $i=2$,

$C_2 = G_2 + P_2 \underline{\underline{C_1}}$

$C_2 = G_2 + P_2(G_1 + P_1 G_0 + P_1 P_0 C_{-1})$

$\underline{\underline{C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{-1}}}$ — ③

If $i=3$,

$C_3 = G_3 + P_3 C_2$

$C_3 = G_3 + P_3(G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{-1})$

$\underline{\underline{C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{-1}}}$ — ④

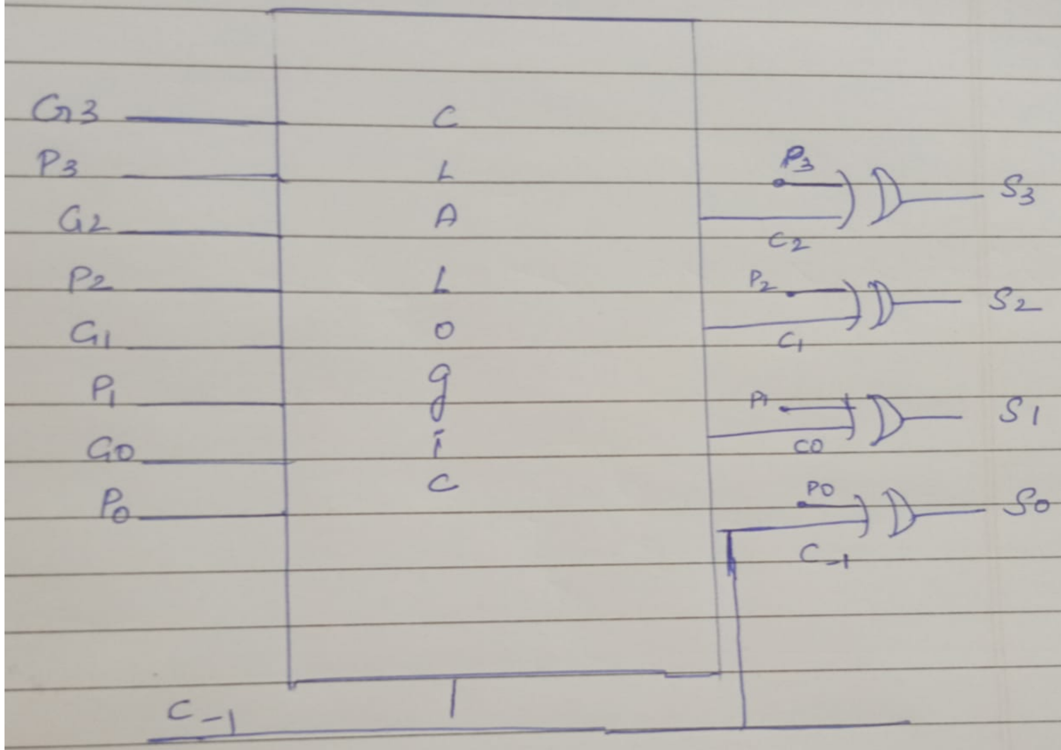
If $i=4$,

$C_4 = G_4 + P_4 C_3$

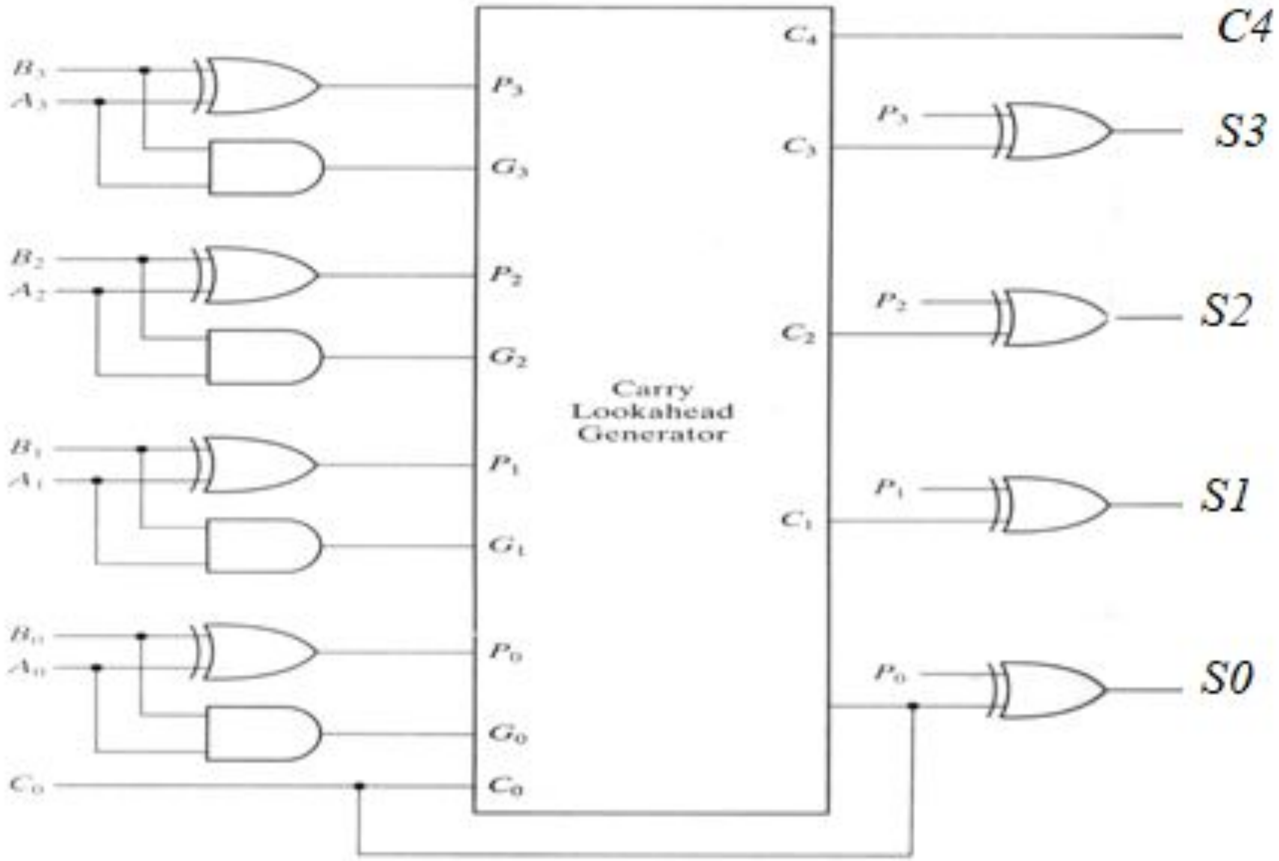
$C_4 = G_4 + P_4(G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{-1})$

$\underline{C_4 = G_4 + P_4 G_3 + P_4 P_3 G_2 + P_4 P_3 P_2 G_1 + P_4 P_3 P_2 P_1 G_0 + P_4 P_3 P_2 P_1 P_0 C}$

— ⑤



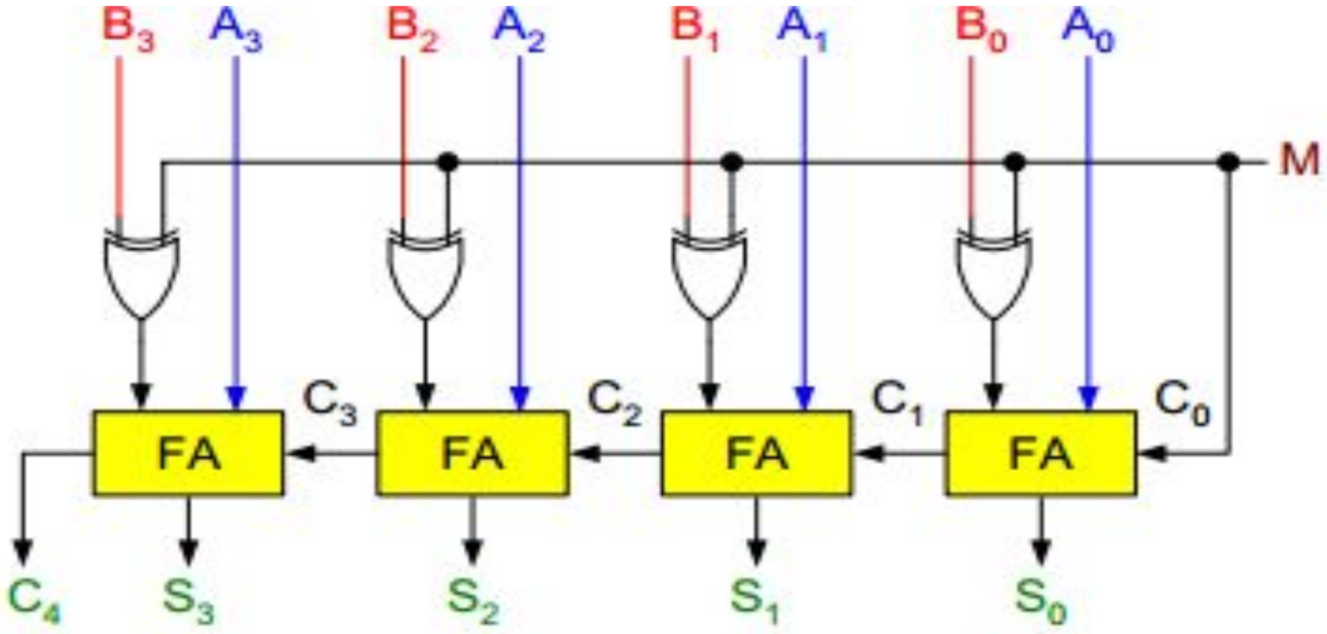
G3
P3
G2
P2
G1
P1
G0
P0
C L A Logic
C_{-1}
P_3
C_2
S_3
P_2
C_1
S_2
P_1
C_0
S_1
P_0
C_{-1}
S_0



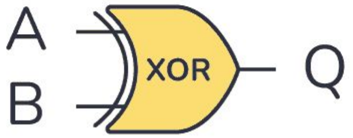
B_3
A_3
B_2
A_2
B_1
A_1
B_0
A_0
C_0
P_3
G_3
P_2
G_2
P_1
G_1
P_0
G_0
C_0
Carry Lookahead Generator
C_4
C_3
C_2
C_1
P_3
P_2
P_1
P_0
C4
S3
S2
S1
S0

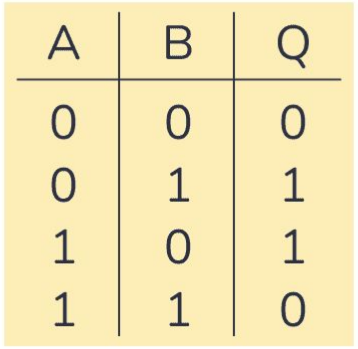
Binary Parallel Adder/Subtractor:

The addition and subtraction operations can be done using an Adder-Subtractor circuit.



- The circuit has a **mode control signal M** which determines if the circuit is to operate as an adder or a subtractor.
- Each XOR gate receives input M and one of the inputs of B, i.e., Bi. To understand the behavior of XOR gate consider its truth table given below.





A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

- If one input of XOR gate is **zero** then the output of XOR will be **same as the second input**.

- While if one input of XOR gate is **one** then the output of XOR will be **complement of the second input.**

- So when M = 0, the output of XOR gate will be $Bi \oplus 0 = Bi$. If the full adders receive the value of B, and the input carry C0 is 0, the circuit **performs A plus B**.
- When M = 1, the output of XOR gate will be $Bi \oplus 1 = Bi$. If the full adders receive the value of B', and the input carry C0 is 1, the circuit performs A plus 1's complement of B plus 1, **which is equal to A minus B**.

MULTIPLIER

What is multiplier in logic?
A binary multiplier is a combinational logic circuit used in digital systems to perform the multiplication of two binary numbers. These are most commonly used in various applications especially in the field of digital signal processing to perform the various algorithms.

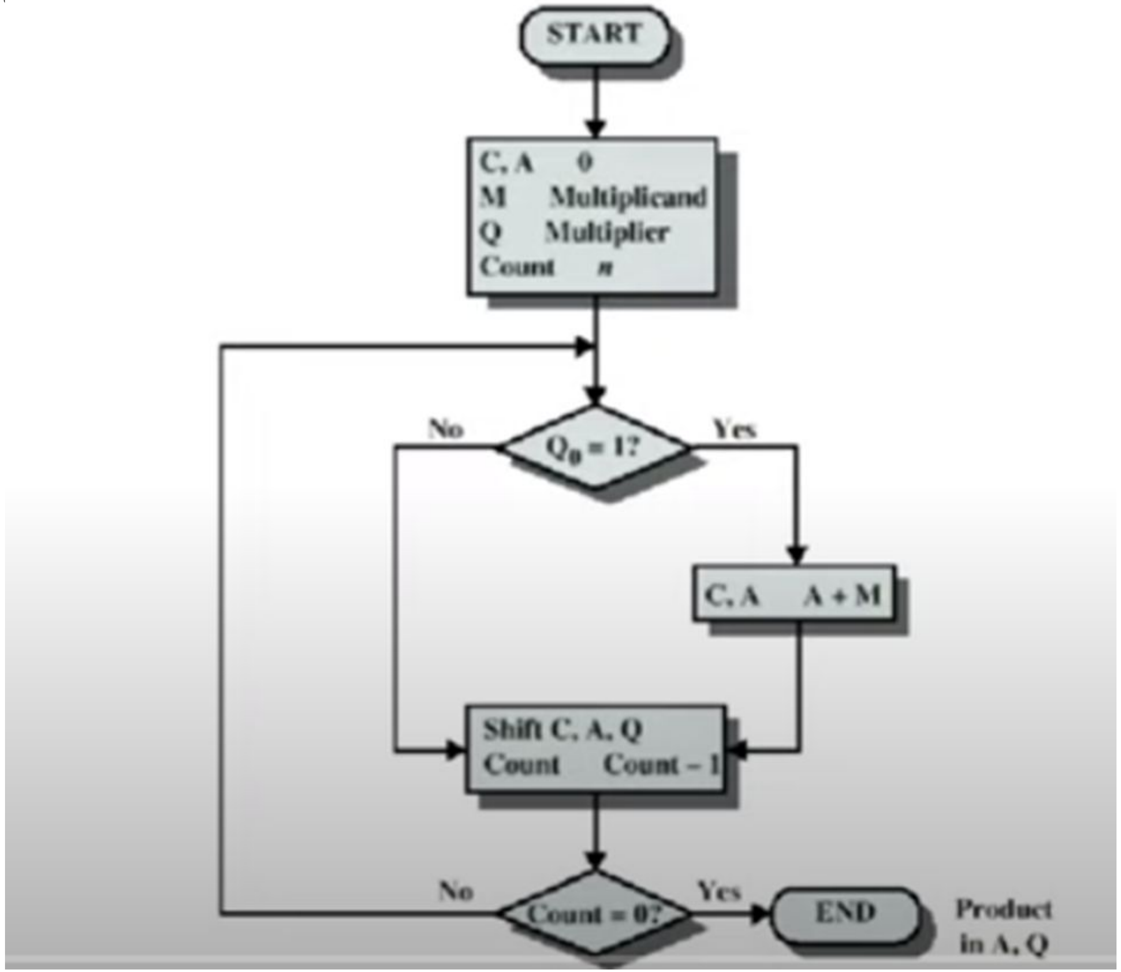
•Two numbers are loaded into registers (fast word storage) and multiplication is carried out by repeated addition and shifting (moving the bits in a register to the right or left).
•Multiplication by an integer is performed by shifting the bits in the word to the left. Each shift multiplys by 2. Note that the ALU consists of the adder and associated circuitry that can also do ands, ors, inversions and other operations, so multiplications can be done by the adder or even by programs steps which can do the same process.
•A hardware multiplier can also be used, which is faster than using the adder and shifter or using programmming. To multiply two floating point numbers, add the exponents, multiply the mantissas, normalize the mantissa of the product, and adjust the exponent accordingly. Floating point multiplication will be covered later.
•A variety of computer arithmetic techniques can be used to implement a digital multiplier. Most techniques involve computing a set of partial products, and then summing the partial products together. This process is similar to the method taught to primary schoolchildren for conducting long multiplication on base-10 integers, but has been modified here for application to a base-2 (binary) numeral system.

The four multipliers include
•Array multiplier
•Column Bypass multiplier
•Modified Booth multiplier
•Wallace tree Multiplier.

Basics
The method taught in school for multiplying decimal numbers is based on calculating partial products, shifting them to the left and then adding them together. The most difficult part is to obtain the partial products, as that involves multiplying a long number by one digit (from 0 to 9):

```
      123
  x 456
  =====
      738  (this is 123 x 6)
      615   (this is 123 x 5, shifted one position to the left)
 + 492    (this is 123 x 4, shifted two positions to the left)
  =====
  56088
```

Unsigned Multi



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.

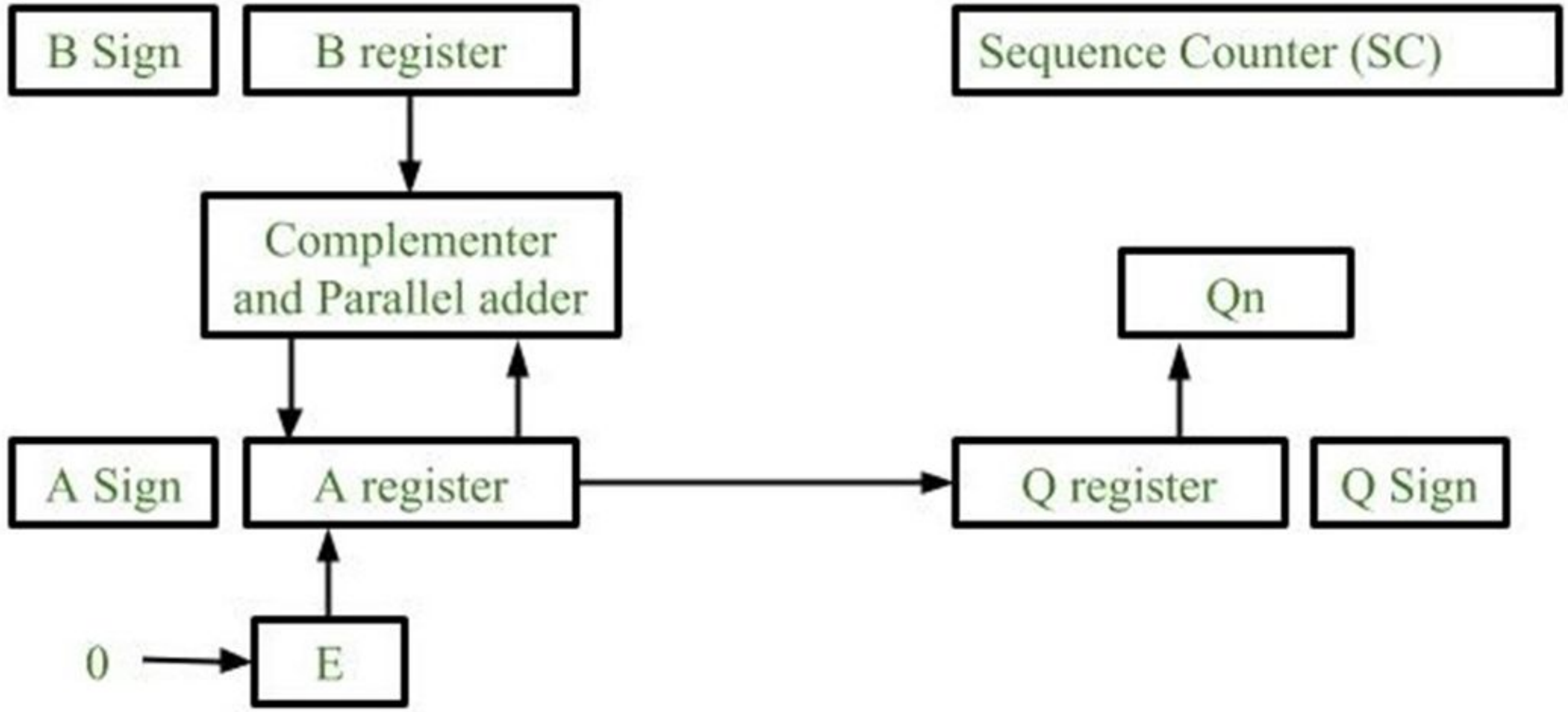
The numbers copied down in successive lines are shifted one position to the left from the previous number.

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.

Hardware Implementation :

Following components are required for the Hardware Implementation of multiplication algorithm :


B Sign
B register
Sequence Counter (SC)
Complementer and Parallel adder
Qn
A Sign
A register
Q register
Q Sign
0
E

Registers:

•Two Registers B and Q are used to store multiplicand and multiplier respectively.

•Register A is used to store partial product during multiplication.

•Sequence Counter register (SC) is used to store number of bits in the multiplier.

Flip Flop:

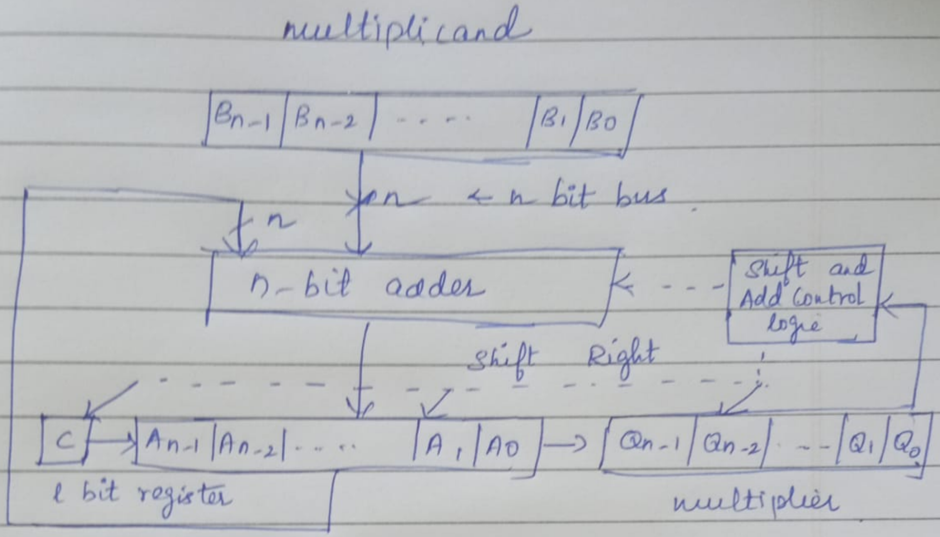
•To store sign bit of registers we require three flip flops (A sign, B sign and Q sign).

•Flip flop E is used to store carry bit generated during partial product addition.

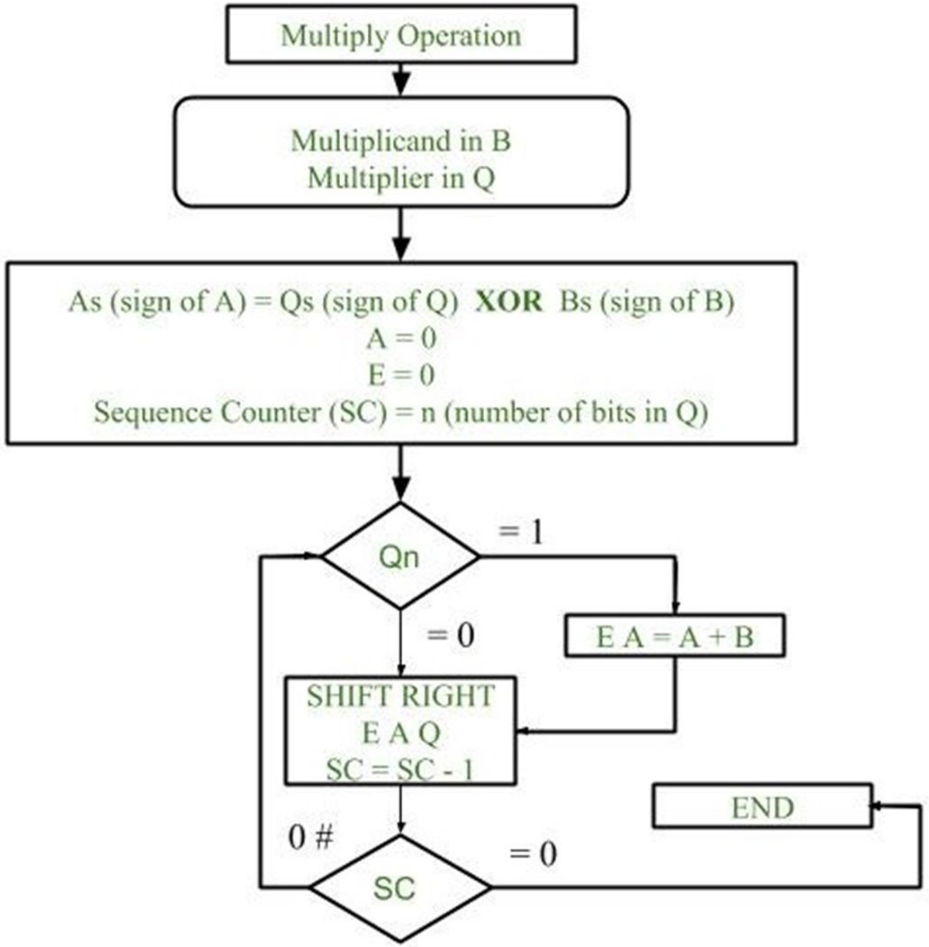
Complement and Parallel adder:

•This hardware unit is used in calculating partial product i.e, perform addition required.

- Initially multiplicand is stored in B register and multiplier is stored in Q register.
- 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.
- 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.
- If Qn = 0, only shift right operation on content of E A Q is performed in a similar fashion.
- Content of Sequence counter is decremented by 1.
- 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.



H/w implementation of unsigned binary multiplication.



Multiply Operation
Multiplicand in B
Multiplier in Q
As (sign of A) = Qs (sign of Q) XOR Bs (sign of B)
A = 0
E = 0
Sequence Counter (SC) = n (number of bits in Q)
Qn
= 1
= 0
E A = A + B
SHIFT RIGHT
E A Q
SC = SC - 1
0 #
= 0
SC
END

Unsigned Multiplication

four Registers:-

1. Multiplicand ⇒ B register
2. Multiplier ⇒ Q register
3. carry ⇒ carry register
4. Temporary ⇒ Temporary register

Two operations :-

If $Q_0 \Rightarrow 0$ ⇒ Right shift.

$Q_0 \Rightarrow 1$ ⇒ Add $A+B$

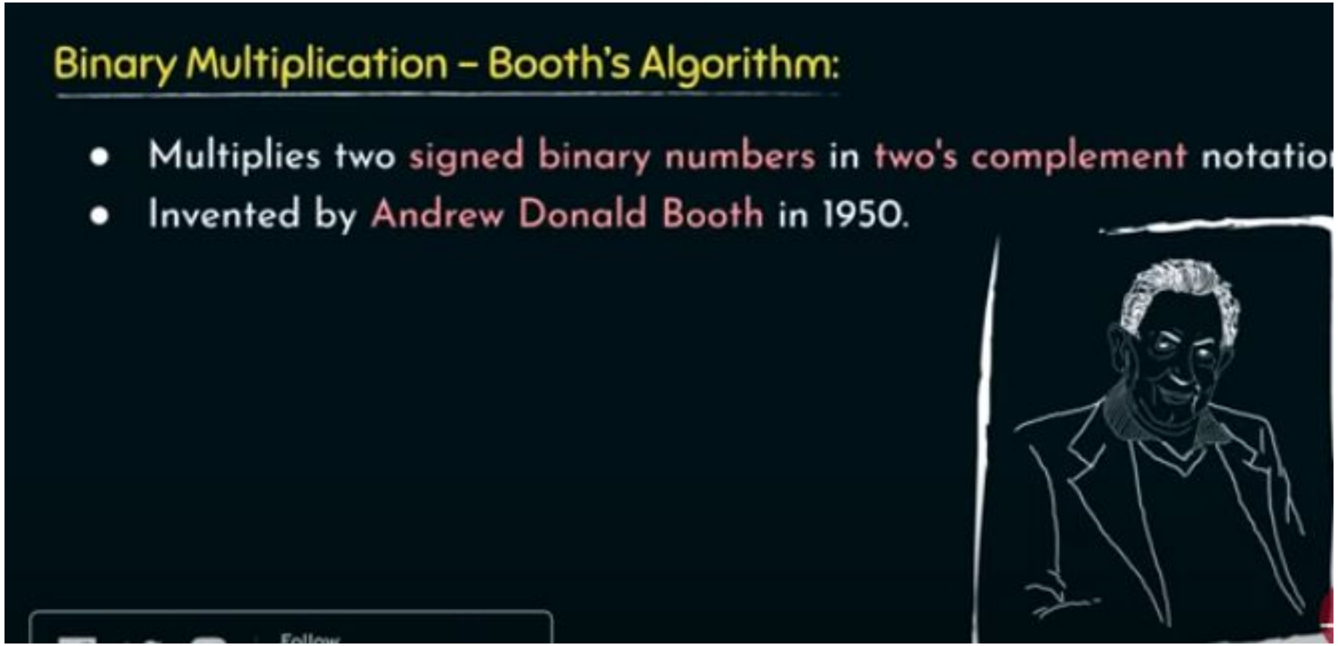
Then Right shift.

$B \Rightarrow \underset{9}{\overset{\Downarrow}{1001}}$ $Q = \underset{7}{\overset{\Downarrow}{0111}}$ $9 \times 7 = 63$

C	A	Q	Step.	cycle
0	0000	0111	Initial	
	1001		A+B	
0	1001	0111		
0	0100	1011	shift.	①
	1001			
0	1101	1011	A+B	
0	0110	1101	shift	②
	1001.			
0	1111	1101		
0	0111	1110	shift.	③.
0	0011	1111	shift	④.

Ans:- 00111111 $\Rightarrow$ 62.

Booth Algorithm



Binary Multiplication – Booth's Algorithm:

- Multiplies two signed binary numbers in two's complement notatio
- Invented by Andrew Donald Booth in 1950.

Date ______
Page ______

M = -7 Q = -3

M ⇒ -7
7 ⇒ 0111
-7 ⇒ 1000
$\underline{\quad\ \ 1}$
M ⇒ 1001
-M ⇒ 0111

3 ⇒ 0011
-3 ⇒ 1101

Q ⇒ 1101

A	Q	Q_{-1}	step	cycle
0000	110$\underline{1}$	$\underline{0}$		
$\underline{0111}$			A-M	
0111	1101	0		
0011	111$\underline{0}$	$\underline{1}$	shift	①
$\underline{1001}$				
1100	1110	1		
1110	011$\underline{1}$	$\underline{0}$		②
$\underline{0111}$				
1) 0101	0111	0	A-M	
0010	1011	1	shift	③
0001	0101	1	shift	④

= 21

Fast Multiplication

Fast Multiplication

- There are two techniques for speeding up the multiplication operation.
- 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 reduces the time needed to add the summands (carry-save addition of summands method).

- BIT PAIR RECODING OF MULTIPLIERS
- CARRY SAVE ADDITION OF SUMMANDS

Bit-Pair Recoding of Multipliers

- This bit-pair recoding technique halves the maximum number of summands. It is derived from the Booth algorithm.
- Group the Booth-recoded multiplier bits in pairs, and observe the following: The pair (+1 -1) is equivalent to the pair (0 +1).
- That is, instead of adding —1 times the multiplicand M at shift position i to + 1 x M at position i + 1, the same result is obtained by adding +1 x M at position I Other examples are: (+1 0) is equivalent to (0 +2),(-l +1) is equivalent to (0 —1). and so on

1) Bit Pair recoding of multipliers :-

→ Fast Multiplication

Bit Pair Recoding Table :-

Multiplier bit-pair		Multiplier bit on right	Multiplicand selected at position i
i+1	i	i-1	
0	0	0	0×M
0	0	1	+1×M
0	1	0	+1×M
0	1	1	+2×M
1	0	0	-2×M
1	0	1	-1×M
1	1	0	-1×M
1	1	1	0×M

Ex:— Recode :—

i+1 i i-1

sign extension ⇐ 1 1 1 0 1 0 [0 Additional bit

0 -1 -2

Steps

1. Write binary equivalent of given decimal
2. Equivalize the bits (Both numbers are of same number of bits)
3. Add the sign bit
4. Recode the Multiplier based on the table
5. While grouping the multiplier, add one extra zero at the LSB.
6. Now Multiply, multiplicand and receded multiplier.
7. For multiplying with the negative numbers, take the 2's complement of the multiplicand and multiply with multiplier. (if multiply by 2, do with 10)

Example :- -11×27

A B

$11 \Rightarrow 1011$

$27 \Rightarrow 11011$

$\therefore$ Represents in 5 bits

$11 \Rightarrow 01011$

$-11 \Rightarrow 10100$
$\quad\quad\quad\quad\quad\;\, 1$

$\underline{10101}$

$-11 \Rightarrow 10101$

$27 \Rightarrow 11\ 011$

Now add sign bit,

$-11 \Rightarrow \underline{1}10101$

$27 \Rightarrow 011011$

Take multiplier to recode :-

$011011\boxed{0} \Rightarrow$ Always add one additional bit

Group them into 3,

$0\,1\,1\,0\,1\,1\,0$

$+2 \quad -1 \quad -1$

```
        1 1 0 1 0 1  ×
⇓ 2n bits  +2  -1  -1
---------------------------------
0 0 0 0 0 0 0 0 1 0 1 1  ⇒ Take 2's comp of multiplicand
0 0 0 0 0 0 1 0 1 1      ⇒ Take 2's comp of multiplicand
1 1 1 0 1 0 1 0          ⇒ multiply by 2 (10)
---------------------------------
1 1 1 0 1 1 0 1 0 1 1 1  ⇒ -297
```

Example 2:- 13 x - 6
A   B

```
                                         6 ⇒ 0110
                                             1001
                                                1
                                        -6   1010
     13 ⇒    1101
     -6 ⇒    1010
Sign bit ⇒   13 ⇒   01101
             -6 ⇒   11010
Extended
         [1]11010[0] ⇒ Additional
          0   -1   -2

          01101 x
          0   -1   -2
  ─────────────────
 1111 1 0 0 1 1 0
 1111 0 0 1 1
 000000
 ─────────────────
[1]1 1 1 0 1 1 1 0 0 1 0    ⇒ -78
```

Carry-Save Addition of Summands

- A **carry-save adder** is a type of digital adder, used to efficiently compute the sum of three or more binary numbers.
- A carry-save adder (CSA), or 3-2 adder, is a very fast and cheap adder that does not propagate carry bits.
- A Carry Save Adder is generally used in binary multiplier, since a binary multiplier involves addition of more than two binary numbers after multiplication.
- It can be used to speed up addition of the several summands required in multiplication
- It differs from other digital adders in that it outputs two (or more) numbers, and the answer of the original summation can be achieved by adding these outputs together.
- A big adder implemented using this technique will usually be much faster than conventional addition of those numbers.
-

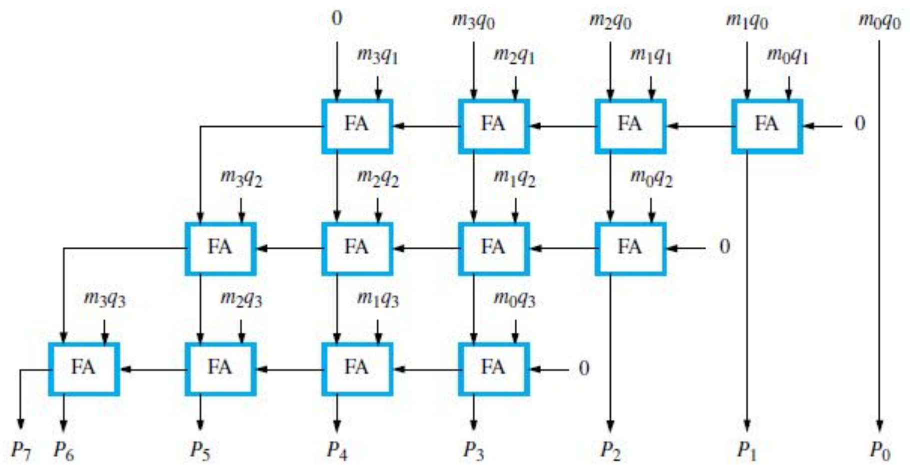
Fast Multiplication

- Bit pair recoding reduces summands by a factor of 2
- Summands are reduced by carry save addition
- Final product can be generated by using carry look ahead adder

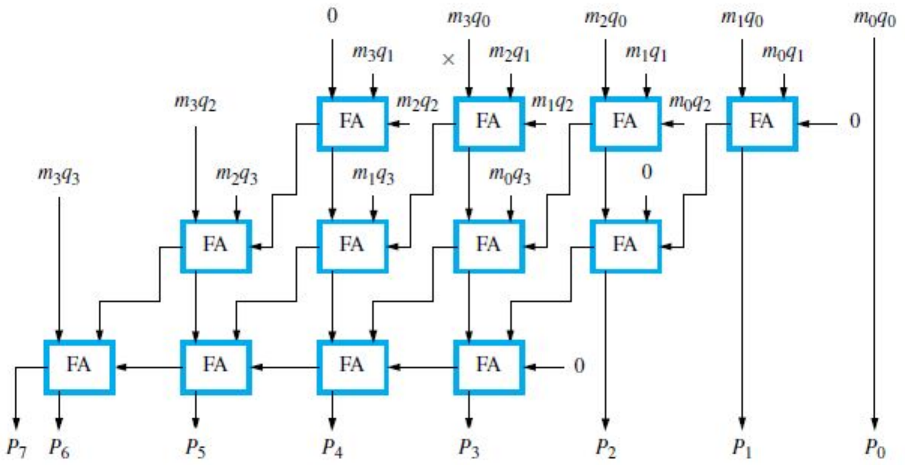
Carry-Save Addition of Summands

- **Disadvantage** of the **Ripple Carry Adder -** Each full adder has to wait for its carry-in from its previous stage full adder. This increase propagation time. This causes a delay and makes ripple carry adder extremely slow. RCAr is very slow when adding many bits.
- **Advantage of the Carry Look ahead Adder -** This is an improved version of the Ripple Carry Adder. Fast parallel adder. It generates the carry-in of each full adder simultaneously without causing any delay. So, CLAr is faster (because of reduced propagation delay) than RCAr.
- **Disadvantage the Carry Look-ahead Adder** - It is costlier as it reduces the propagation delay by more complex hardware. It gets more complicated as the number of bits increases.

Ripple Carry Array



Carry Save Array



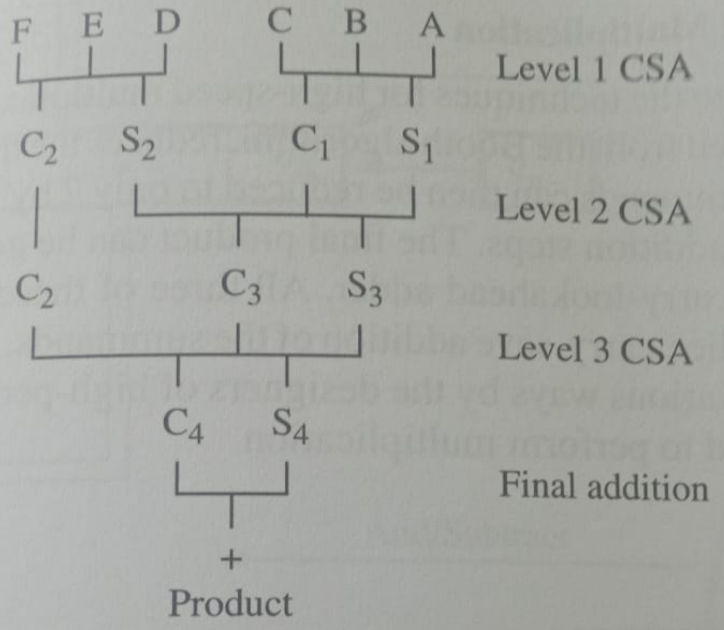
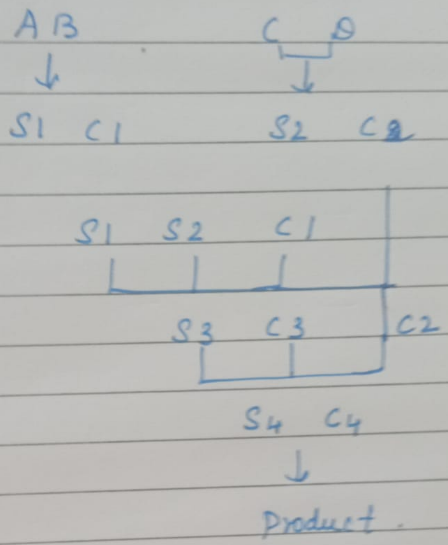


Figure 6.19 Schematic representation of the carry-save addition operations in Figure 6.18.

Carry-save Addition of Summands:-



```
      1 0 1 0 X
      0 1 1 0
   ____________
      0 0 0 0      A
    1 0 1 0 0      B
  1 0 1 0 0 0      C
0 0 0 0 0 0 0      D
_______________
        1
0 1 1 1 0̸ 0 0
_______________
```

A	0000
B	10100
S1	10100
C1	00000

C	101000
D	000000
S2	101000
C2	000000

S1	10100
S2	101000
C1	00000
S3	111100
C3	000000

S3 →	111100
C3 ⇒	0000 00
C2 ⇒	000000
product :—	111100

Carry-Save Addition of Summands

- Consider the addition of many summands, We can:
- Group the summands in threes and perform carry-save addition on each of these groups in parallel to generate a set of S and C vectors in one full-adder delay
- Group all of the S and C vectors into threes, and perform carry-save addition of them, generating a further set of S and C vectors in one more full-adder delay
- Continue with this process until there are only two vectors remaining
- They can be added in a Ripple Carry Adder (RPA) or Carry Look-ahead Adder (CLA) to produce the desired product

Division

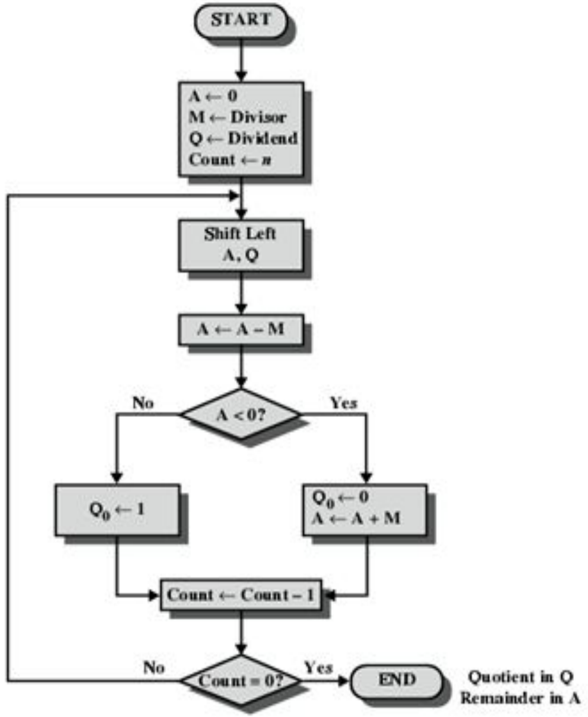
Division is carried out in two ways:

1. Restoring
2. Non Restoring

Restoring

- Similar to multiplication circuit
- 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 operation 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.

Flowchart for Restoring Division



Steps for Restoring

1. Left Shift
2. $A = A - M$
3. If MSB[A] = 0 then $Q_0 = 1$
4. If MSB[A] = 1 then $Q_0 = 0$, Restore A

Restoring

7/2 Q ⇒ 7 ⇒ 0111

−M ⇒ 11101 / 11110 M ⇒ 2 ⇒ 0010

M ⇒ 00010

A	Q	Operation	
00000	0111	initial	
00000	111□	Shift	I
11110			
11110	111□	A−M	
11110	1110	Q[0] 0	
00000	1110	Restore	
00001	110□	Shift	II
11110			
11111	110□	A−M	
11111	1100	Q[0] = 0	
00001	1100	Restore	
00011	100□	Shift	III
11110			
1] 00001	100□	A−M	
00001	1001	Q[0] = 1	
00011	001□	Shift	IV
11110			
1] 00001	001□	A−M	
0 $\boxed{0001}$	$\boxed{0011}$	Q[0] = 1	

Non-Restoring Division

- Initially Dividend is loaded into register Q, and n-bit Divisor is loaded into register M
- Let M’ is 2’s complement of M
- Set Register A to 0
- Set count to n
- SHL AQ denotes shift left AQ by one position leaving Q_0 blank.
- Similarly, a square symbol in Q_0 position denote, it is to be calculated later

Non restoring Steps

1. If MSB[A] = 0 then
   a. Left Shift
   b. A = A - M
2. If MSB[A] = 1 then
   a. Left Shift
   b. A = A + M
3. If MSB[A] = 0 then

   Q_0 = 1

1. If MSB[A] = 1 then Q_0 = 0

Non-Restoring Q = 7 0111

M = 00010

-M = 11110

$\boxed{7/2}$

A	Q	steps	
00000	0111		
00000	111□	shift.	I
11110			
11110	111□	A-M	
11110	1110	Q[0]=0	
11101	110□	shift.	II
00010			
11111	110□	A+M	
11111	1100	Q[0]=0	
11111	100□	shift.	III
00010			
1 00001	100□	A+M.	
00001	1001	Q[0]=1	
00011	001□	shift	
11110			
1 00001	001□	A-M	
0 $\boxed{0001}$	$\boxed{0011}$	Q[0]=1	

Ans:- Reminder :- 0001 ⇒ 1

Quotient :- 0011 ⇒

IEEE 754

Floating point numbers and operations.

Floating-Point Arithmetic (IEEE 754)

- ✔ The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation which was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).
- ✔ The standard addressed many problems found in the diverse floating point implementations that made them difficult to use reliably and reduced their portability.
- ✔ IEEE Standard 754 floating point is the most common representation today for real numbers on computers, including Intel-based PC's, Macs, and most Unix platforms.

IEEE 754 - Basic components

IEEE 754 has 3 basic components:

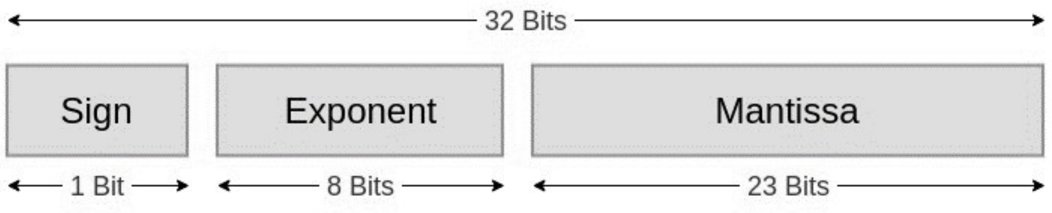
1. **The Sign of Mantissa**
   - ✔ 0 represents a positive number while
   - ✔ 1 represents a negative number.
2. **The Biased exponent**
   - ✔ The exponent field needs to represent both positive and negative exponents.
   - ✔ A bias is added to the actual exponent in order to get the stored exponent.
3. **The Normalised Mantissa**
   - ✔ The mantissa is part of a number in scientific notation or a floating-point number, consisting of its significant digits.
   - ✔ Here we have only 2 digits, i.e. O and 1. So a normalised mantissa is one with only one 1 to the left of the decimal.

IEEE 754 - Precision

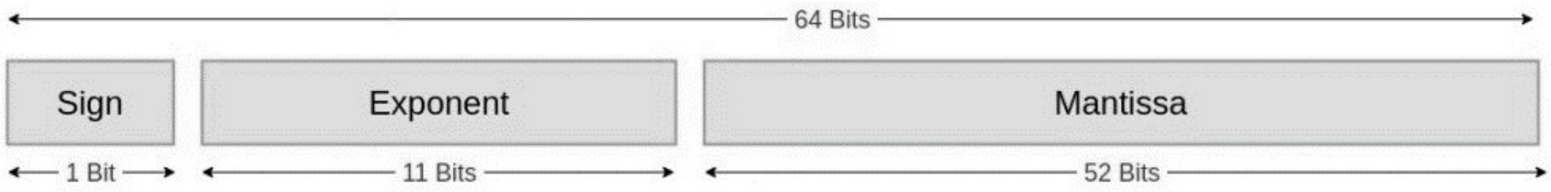
IEEE 754 numbers are divided into two based on the above three components:

- ✔ Single precision and
- ✔ Double precision.

1. Single p



2. Double precision



Example 1: Single precision

Biased exponent 127+6=133

133 = 10000101

Normalised mantisa = 010101001

we will add 0's to complete the 23 bits

The IEEE 754 Single precision is:

0 10000101 01010100100000000000000

This can be written in hexadecimal form **42AA4000**

Example 2: Double precision

Biased exponent 1023+6=1029

1029 = 10000000101

Normalised mantisa = 010101001

we will add 0's to complete the 52 bits

The IEEE 754 Double precision is:

0 10000000101 01010100100000000000000000000000000000000000000000000

This can be written in hexadecimal form **4055480000000000**

IEEE 754 - Special Values

EEE has reserved some values that can ambiguity.

1. **Zero**
   - ✔ Zero is a special value denoted with an exponent and mantissa of 0.
   - ✔ -0 and +0 are distinct values, though they both are equal.
2. **Denormalised**
   - ✔ If the exponent is all zeros, but the mantissa is not then the value is a denormalized number.
   - ✔ This means this number does not have an assumed leading one before the binary point.
3. **Infinity**
   - ✔ The values +infinity and -infinity are denoted with an exponent of all ones and a mantissa of all zeros.
   - ✔ The sign bit distinguishes between negative infinity and positive infinity.

IEEE 754 - Special Values

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

4. Not A Number (NAN)

- ✔ The value NAN is used to represent a value that is an error.
- ✔ This is represented when exponent field is all ones with a zero sign bit or a mantissa that it not 1 followed by zeros.
- ✔ This is a special value that might be used to denote a variable that doesn't yet hold a value.

Single Precision		
Exponent	**Mantisa**	**Value**
0	0	exact 0
255	0	Infinity
0	not 0	denormalised
255	not 0	Not a number (NAN)

Double Precision		
Exponent	**Mantisa**	**Value**
0	0	exact 0
2049	0	Infinity
0	not 0	denormalised
2049	not 0	Not a number (NAN)

Ranges of Floating point numbers

- ✔ The range of positive floating point numbers can be split into normalized numbers, and denormalized numbers which use only a portion of the fractions's precision.
- ✔ Since every floating-point number has a corresponding, negated value, the ranges above are symmetric around zero.
- ✔ There are five distinct numerical ranges that single-precision floating-point numbers are not able to represent with the scheme presented so far:

1. Negative numbers less than – (2 – 2-23) × 2127 (negative overflow)
2. Negative numbers greater than – 2-149 (negative underflow)
3. Zero
4. Positive numbers less than 2-149 (positive underflow)
5. Positive numbers greater than (2 – 2-23) × 2127 (positive overflow)

Ranges of Floating point numbers

✔ Overflow generally means that values have grown too large to be represented. Underflow is a less serious problem because is just denotes a loss of precision, which is guaranteed to be closely approximated by zero.

The total effective range of finite IEEE floating-point numbers is

	Binary	**Decimal**
Single Precision	$\pm (2 - 2^{-23}) \times 2^{127}$	approximately $\pm 10^{38.53}$
Double Precision	$\pm (2 - 2^{-52)} \times 2^{1023}$	approximately $\pm 10^{308.25}$

IEEE 754 - Special Operations

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

Operation	Result
n ÷ ±Infinity	0
±Infinity × ±Infinity	±Infinity
±nonZero ÷ ±0	±Infinity
±finite × ±Infinity	±Infinity
Infinity + Infinity Infinity – -Infinity	+Infinity

Operation	Result
-Infinity – Infinity -Infinity + – Infinity	– Infinity
±0 ÷ ±0	NaN
±Infinity ÷ ±Infinity	NaN
±Infinity × 0	NaN
NaN == NaN	False