UNIT 1- DIGITAL LOGIC CIRCUITS

TOPICS:

- **Logic Gates**
  - **AND, OR, NOT, NAND, NOR, XOR, Exclusive NOR gates, Universal Gate**
- **Boolean Algebra**
  - **Boolean algebra?**
  - **Boolean variable and Boolean function (Analog and Digital Signals)**
  - **Truth table**
  - **Postulates**
  - **Theorem related to postulates**
  - **Simplified Boolean function using postulates and draw logical diagram of simplified function**
  - **Simplified Boolean function using Karnaugh map method with DON'T CARE condition**
- **Sequential and Combinational Circuits**
  - **Clock pulses**
  - **Combinational circuit**
  - **sequential circuit and adder**
- **Flip Flops**
  - **SR, Clocked SR, D, JK, JK – Master Slave, T**

- ✓ Digital logic circuits are the basis of digital systems. These logic circuits are a set of logic gates that show logical equivalence between two different groups of binary numbers.
- ✓ The most common digital circuits are based on the binary number system, although some systems use non-binary values.
- ✓ The digital circuit contains switches, which are either on or off. It processes information as a sequence of "1" s and "0" s.

Why are digital circuits also called Logical Circuits?

- ✓ Digital circuits are also called logical circuits because they perform logical operations on digital signals.
- ✓ **Digital circuits use logic gates like AND, OR, NOT, NAND, and NOR to perform the required digital operations.**
- ✓ A digital circuit is a circuit containing digital logic.

- ✓ **Digital circuits are the most common physical implementation of Boolean algebra and binary arithmetic and are the basis of all modern computers.**
- ✓ It is because digital circuits are mainly used to process data that has only two values, such as true or false.
- ✓ The logic gates are the basic component of digital circuits that can perform the conversion of binary information.
- ✓ A Digital circuit contains a network of multiple logic gates which are interconnected to each other.
- ✓ Each gate has a different symbol by which they are represented, and an algebraic function defines its operation.
- ✓ **A truth table can determine the connection between the output and input variables of each gate.**
- ✓ **A timing diagram determines the reciprocation of the signal of logic gates.**
- ✓ **In binary, the value "true" is represented by "1" and "false" is represented by "0"**

Types of Logic Gates

Based on the operation performed, there are basically three types of **logic gates**:

1. **Basic logic gate**
2. **Universal logic gate**
3. **Exclusive logic gate**

Each of the above types can further have different logic gates based on their behaviour.

Basic Logic gates

There are three basic logic gates:

1. AND logic gate
2. OR logic gate
3. NOT logic gate

Universal Logic gates

There are two universal logic gates:

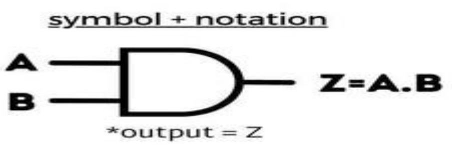
1. NAND gates
2. NOR gates

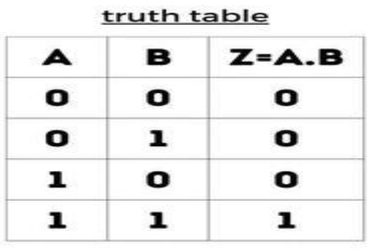
Exclusive Logic gates

There are two exclusive logic gates:

1. XOR gates
2. XNOR gates

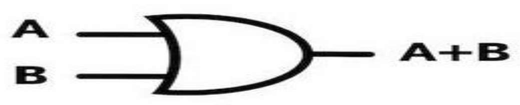
1. **AND GATE**
    - The AND gate is named so because, if 0 is false and 1 is true, the gate acts in the same way as the logical "and" operator. The following illustration and table show the circuit symbol and logic combinations for an AND gate. (In the symbol, the input terminals are on the left, and the output terminal is on the right.) **The output is "true" when both inputs are "true." Otherwise, the output is "false." In other words, the output is 1 only when both inputs are 1.**
    - **Boolean function: - C= A.B OR C=(AB)**

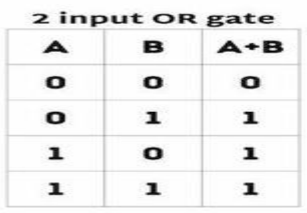



truth table

A	B	Z=A.B
0	0	0
0	1	0
1	0	0
1	1	1

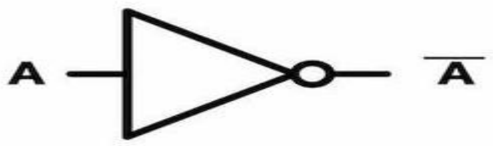
2. **OR GATE**
    - The OR gate gets its name from behaving like the logical inclusive "or." The output is true if one or both of the inputs are true. **If both inputs are false, then the output is false. In other words, for the output to be 1, at least one input must be 1.**
    - **Boolean function: - C= A+B OR C=(A+B)**

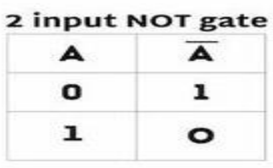



2 input OR gate

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

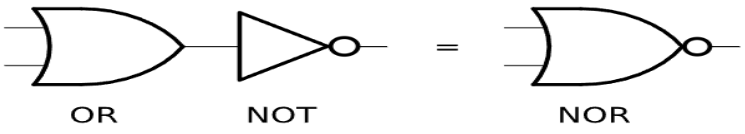
3. **LOGICAL INVERTER (NOT GATE)**
    - A logical inverter, sometimes called a NOT gate to differentiate it from other types of electronic inverter devices, has only one input. **A NOT gate reverses the logic state. If the input is 1, then the output is 0. If the input is 0, then the output is 1.**
    - **Boolean function: - $\overline{A}$= A' OR $\overline{A}$=A**




2 input NOT gate

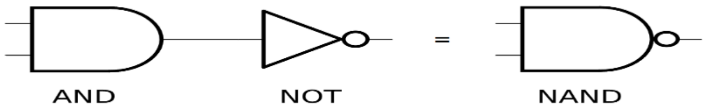
A	$\overline{A}$
0	1
1	0

4. **NOR GATE (NEGATIVE OR)**
    - The NOR (NEGATIVE OR) gate is a combination OR gate followed by an inverter. **Its output is true if both inputs are false. Otherwise, the output is false.**
    - **Boolean function: - $\overline{C}$= (A+B)**



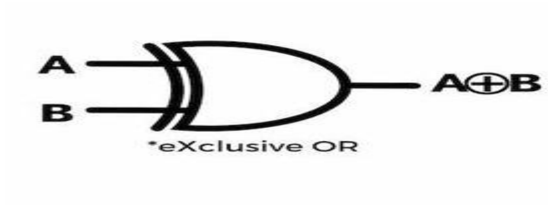
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

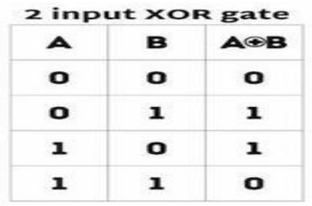
5. **NAND GATE (NEGATIVE AND)**
    - The NAND (NEGATIVE AND) gate operates as an AND gate followed by a NOT gate. **The output is false if both inputs are true. Otherwise, the output is true.**
    - Another way to visualize it is that a NAND gate inverts the output of an AND gate. The NAND gate symbol is an AND gate with the circle of a NOT gate at the output.
    - **Boolean function: - $\overline{C}$= (A.B)**



A	B	$\overline{A\cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

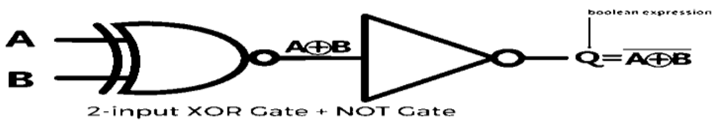
6. **XOR (Exclusive-OR) GATE**
    - The XOR (exclusive-OR) gate acts in the same way as the logical "either/or." **The output is true if either, but not both, of the inputs are true. The output is false if both inputs are "false" or if both inputs are true.** Similarly, **the output is 1 if the inputs are different but 0 if the inputs are the same.**

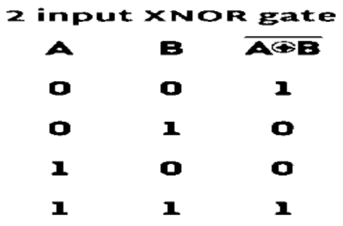



2 input XOR gate

A	B	$A\oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

7. **XNOR GATE (EXCLUSIVE-NOR)**
    - The XNOR (exclusive-NOR) gate is a combination of an XOR gate followed by an inverter. **Its output is true if the inputs are the same and false if the inputs are different.**




2 input XNOR gate

A	B	$\overline{A\oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

Boolean algebra

- ✓ Boolean algebra is the category of algebra in which the variable's values are the truth values, true and false, ordinarily denoted 1 and 0 respectively.
- ✓ **It is used to analyse and simplify digital circuits or digital gates. It is also called Binary Algebra or logical Algebra.**
- ✓ It has been fundamental in the development of digital electronics and is provided for in all modern programming languages. It is also used in set theory and statistics.
- ✓ **Boolean Algebra Rules**
  - o Variable used can have only two values. Binary 1 for HIGH and Binary 0 for LOW.
  - o **The complement of a variable is represented by an overbar.**
  - o **Thus, complement of variable B is represented as $\overline{B}$. Thus if B= 0 then $\overline{B}$=1 and B =1 then $\overline{B}$=0.**
  - o OR-ing of the variables is represented by a plus (+) sign between them. For example, the OR-ing of A, B, and C is represented as A + B + C.
  - o Logical AND-ing of the two or more variables is represented by writing a dot between them, such as A.B.C. Sometimes, the dot may be omitted like ABC.
- ✓ **Terminologies used in boolean Algebra**
  - ✓ **Variable –**
    - o The symbol which represent an element of a Boolean algebra is known as Boolean variable. In an expression, A+BC, the variables are A, B, C, which can value either 0 or 1.
  - ✓ **Constant –**
    - o It is a fixed value. In an expression, Y=A+1, A represents a variable and 1 is a fixed value, which is termed as a constant.
  - ✓ **Literal –**
    - o Each occurrence of a variable in Boolean function either in complemented or normal form is said to be literal.

Boolean variable and Boolean function (Analog and Digital Signals)

Boolean Variables:

- ✓ A Boolean variable is defined as a variable or a symbol defined as a variable or a symbol, generally an alphabet that represents the logical quantities such as 0 or 1.

Binary Variables

- ✓ A binary variable is a symbol that can take one of the two possible values i.e., 0 and 1.
- ✓ If a binary variable has a value 0 associated to it. Then, it represents a low or false state.
- ✓ While if the value of the binary variable is 1, then it represents the high or true state.

Boolean Function

- ✓ A Boolean function is a mathematical expression consists of binary variables and logical operators. It defines a logical relationship between the binary variables and binary output.
- ✓ The Boolean functions are defined using the rules of Boolean algebra and binary number system. These functions build the foundation of design and development of digital circuits and systems.

Truth Table:

- ✓ The truth table is a table that gives all the possible values of logical variables and the combination of the variables.
- ✓ It is possible to convert the Boolean equation into a truth table. The number of rows in the truth table should be equal to 2n, where "n" is the number of variables in the equation.

Boolean Algebra Truth Table

Now, if we express the above operations in a truth table, we get;

A	**B**	**A ∧ B (AND OPERATION)**	**A ∨ B (OR OPERATION)**
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Postulates & Theorem related to postulates

A postulate is a statement that is assumed true without proof. A theorem is a true statement that can be proven.

Laws for Boolean Algebra

The basic laws of the Boolean Algebra are added in the table added below,

Law	OR form	AND form
Identity Law	P + 0 = P	P.1 = P
Idempotent Law	P + P = P	P.P = P
Commutative Law	P + Q = Q + P	P.Q = Q.P
Associative Law	P + (Q + R) = (P + Q) + R	P.(Q.R) = (P.Q).R
Distributive Law	P + QR = (P + Q).(P + R)	P.(Q + R) = P.Q + P.R
Inversion Law	(A')' = A	(A')' = A
De Morgan's Law	(P + Q)' = (P)'.(Q)'	(P.Q)' = (P)' + (Q)'

1. **Identity Law**
   a. In the Boolean Algebra, we have identity elements for both AND(.) and OR(+) operations. **The identity law state that in boolean algebra we have such variables that on operating with AND and OR operation we get the same result**, i.e.

      A + 0 = A

      A.1 = A
2. **Commutative Law**
   a. Binary variables in Boolean Algebra follow the commutative law. **This law states that operating boolean variables A and B is similar to operating boolean variables B and A.** That is,

      A. B = B. A

      A + B = B + A
3. **Associative Law**
   a. Associative law state that the order of performing Boolean operator is illogical **as their result is always the same**. This can be understood as,

      (A. B). C = A. (B. C)

      (A + B) + C = A + (B + C)
4. **Distributive Law**
   a. Boolean Variables also follow the distributive law and the expression for Distributive law is given as:

      A. (B + C) = (A. B) + (A. C)

5. **Inversion Law**
    a. Inversion law is the unique law of Boolean algebra this law states that, the **complement of the complement of any number is the number itself.**

        (A')' = A
6. **AND Law**
    a. AND law of the Boolean algebra uses AND operator and the AND law is,

        A. 0 = 0
        A. 1 = A
        A. A = A
7. **OR Law**
    a. OR law of the Boolean algebra uses OR operator and the OR law is,

        A + 0 = A
        A + 1 = 1
        A + A = A

8. **De Morgan's Laws**
    a. De Morgan's Laws are also called De morgan's Theorem.
    b. They are the most important laws in Boolean Algebra and these are added below under the heading Boolean Algebra Theorem
    c. **Boolean Algebra Theorems**
        i. There are two basic theorems of great importance in Boolean Algebra, **which are De Morgan's First Laws, and De Morgan's Second Laws**. These are also called De Morgan's Theorems.
        ii. **De Morgan's First laws**
            1. De Morgan's Law states that the **complement of the product (AND) of two Boolean variables (or expressions) is equal to the sum (OR) of the complement of each Boolean variable** (or expression).

                (P.Q)' = (P)' + (Q)'

The truth table for the same is given below:

P	Q	(P)'	(Q)'	(P.Q)'	(P)' + (Q)'
T	T	F	F	F	F
T	F	F	T	T	T
F	T	T	F	T	T
F	F	T	T	T	T

2. We can clearly see that truth values for (P.Q)’ are equal to truth values for (P)’ + (Q)’, corresponding to the same input. Thus, De Morgan’s First Law is true.

iii. De Morgan’s Second laws

1. Statement: The Complement of the sum (OR) of two Boolean variables (or expressions) is equal to the product(AND) of the complement of each Boolean variable (or expression).
(P + Q)’ = (P)’.(Q)’
The truth table for the same is given below:

P	Q	(P)’	(Q)’	(P + Q)’	(P)’.(Q)’
T	T	F	F	F	F
T	F	F	T	F	F
F	T	T	F	F	F
F	F	T	T	T	T

2. We can clearly see that truth values for (P + Q)’ are equal to truth values for (P)’.(Q)’, corresponding to the same input. Thus, De Morgan’s Second Law is true.

Example

Draw Truth Table for P + P.Q = P
Solution:
The truth table for P + P.Q = P

P	Q	P.Q	P + P.Q
T	T	T	T
T	F	F	T
F	T	F	F
F	F	F	F

In the truth table, we can see that the truth values for P + P.Q is exactly the same as P.

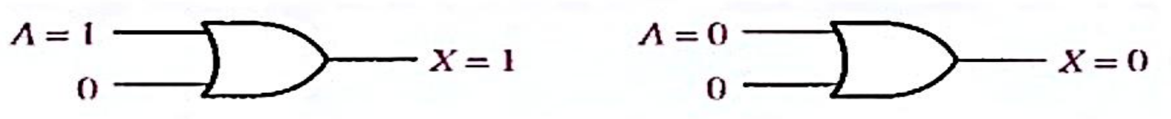
Draw Truth Table for P.Q + P + Q
Solution:
The truth table for P.Q + P + Q

P	Q	P.Q	P.Q + P + Q
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

Simplified Boolean function using postulates and draw logical diagram of simplified function

Rule 1. A + 0 = A

A variable ORed with 0 is always equal to the variable. If the input variable A is 1, the output variable X is 1, which is equal to A. If A is 0, the output is 0, which is also equal to A. This rule is illustrated in Fig., where the lower input is fixed at 0.



A+O=A

Rule 2. A + 1 = 1

A variable ORed with 1 is always equal to 1. A 1 on an input to an OR gate produces a 1 on the output, regardless of the value of the variable on the other input. This rule is illustrated in Fig., where the lower input is fixed at 1.

A = 1
1
X = 1
A = 0
1
X = 1

X=A+1=1

Rule 3. A . 0 = 0

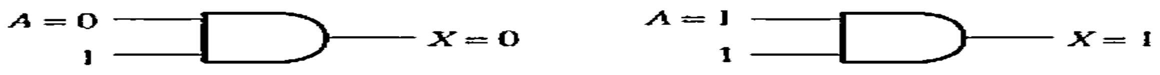
A variable ANDed with 0 is always equal to 0. Any time one input to an AND gate is 0, the output is 0, regardless of the value of the variable on the other input. This rule is illustrated in Fig., where the lower input is fixed at 0.

A = 1
0
X = 0
A = 0
0
X = 0

X= A.O = 0

Rule 4. A . 1 = A

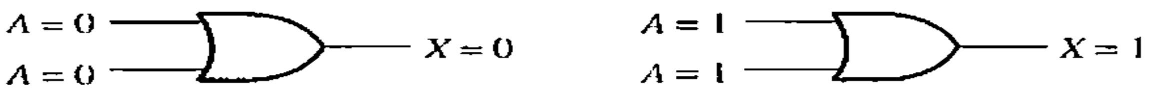
A variable ANDed with 1 is always equal to the variable. If A is 0 the output of the AND gate is 0. If A is 1, the output of the AND gate is 1 because both inputs are now 1s. This rule is shown in Fig., where the lower input is fixed at 1.



$X = A . 1 = A$

Rule 5. A + A = A

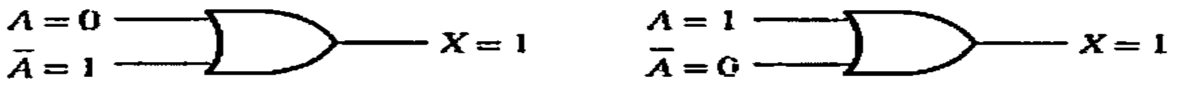
A variable ORed with itself is always equal to the variable. If A is 0, then 0 + 0 = 0; and if A is 1, then 1 + 1 = 1. This is shown in Fig., where both inputs are the same variable.



$X = A + A = A$

Rule 6. A + A = 1

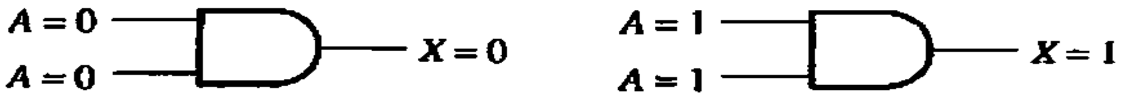
A variable ORed with its complement is always equal to 1. If A is 0, then 0 + 0 = 0 + 1 = 1. If A is l, then 1 + 1 = 1+ 0 = 1. See Fig., where one input is the complement of the other.



$X = A + \overline{A} = 1$

Rule 7. A . A = A

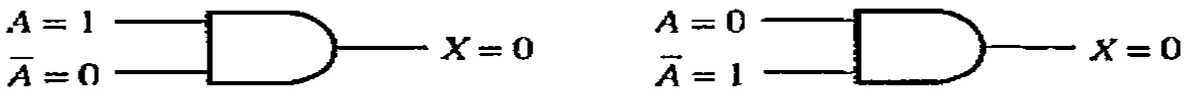
A variable ANDed with itself is always equal to the variable. If A = 0, then 0.0 = 0; and if A = 1. then 1.1 = 1. Fig. illustrates this rule.



$A . A = A$

Rule 8. A . A = 0

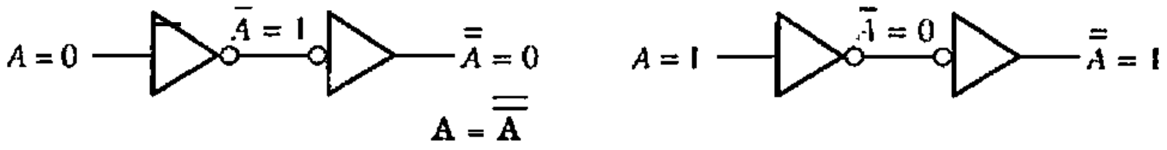
A variable ANDed with its complement is always equal to 0. Either A or A will always be 0: and when a 0 is applied to the input of an AND gate. the output will be 0 also. Fig. illustrates this rule.



$A . \overline{A} = 0$

Rule 9 $A = \overline{\overline{A}}$

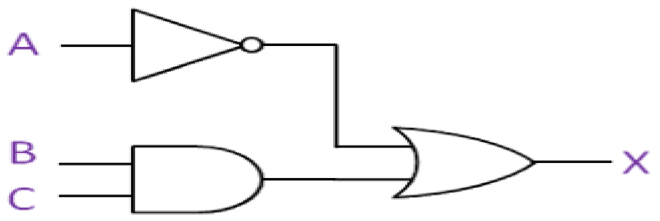
The double complement of a variable is always equal to the variable.



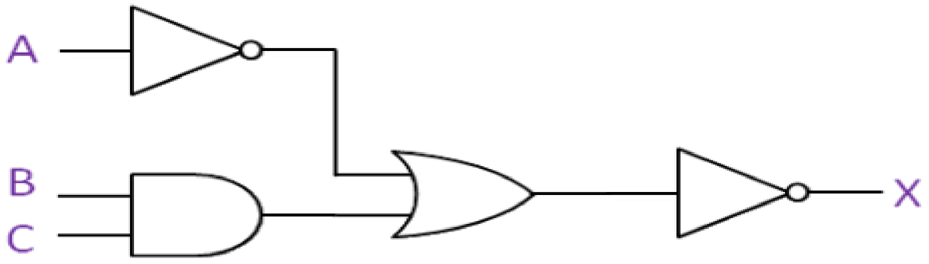
$A = 0$
$\overline{A} = 1$
$\overline{\overline{A}} = 0$
$A = 1$
$\overline{A} = 0$
$\overline{\overline{A}} = 1$
$A = \overline{\overline{A}}$

Logic Gate Diagrams

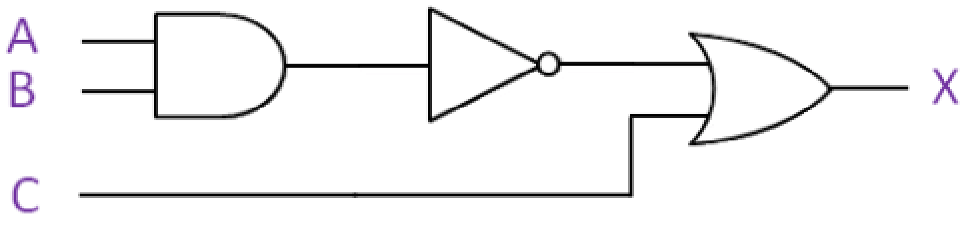
(NOT A) OR (B AND C)



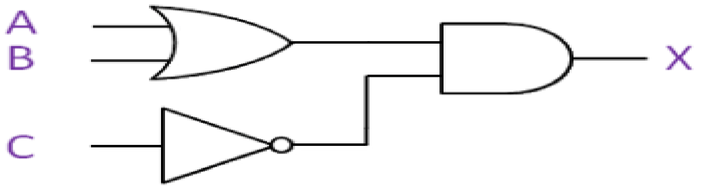
NOT ((NOT A) OR (B AND C))



(NOT (A AND B)) OR C

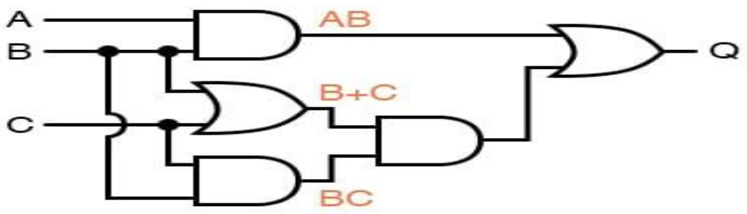


(A OR B) AND (NOT C)

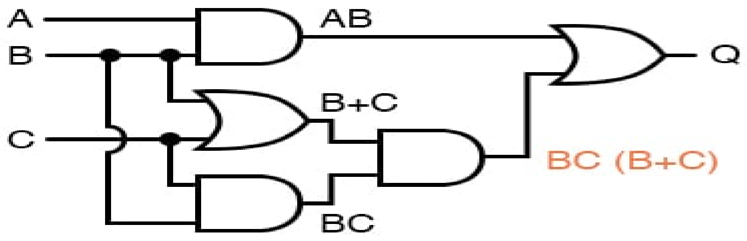


How to Write a Boolean Expression to Simplify Circuits?

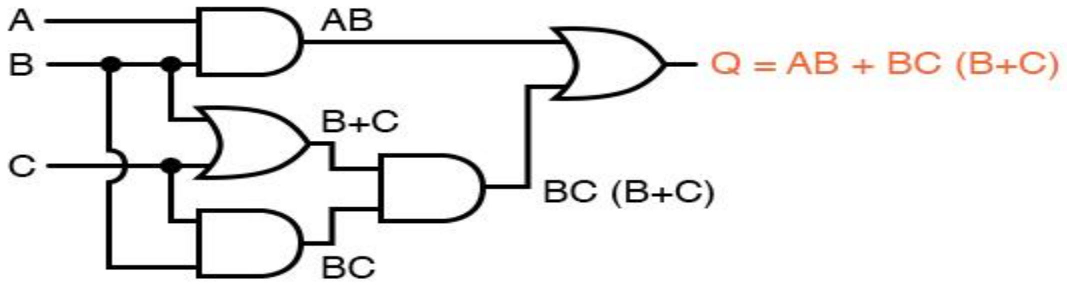
- ✓ Our first step in simplification must be to write a Boolean expression for this circuit.
- ✓ This task is easily performed step by step if we start by writing sub-expressions at the output of each gate, corresponding to the respective input signals for each gate.
- ✓ **Remember that OR gates are equivalent to Boolean addition, while AND gates are equivalent to Boolean multiplication.**
  Q= AB+BC(B+C)



- ✓ . . . then another sub-expression for the next gate:



- ✓ Finally, the output ("Q") is seen to be equal to the expression AB + BC(B + C):
- ✓ Now that we have a Boolean expression to work with, we need to apply the rules of Boolean algebra to reduce the expression to its simplest form (simplest defined as requiring the fewest gates to implement):

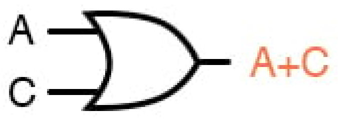


$$
\begin{array}{cl}
AB + BC(B + C) & \\
\downarrow & \text{Distributing terms} \\
AB + BBC + BCC & \\
\downarrow & \text{Applying identity } AA = A \text{ to 2nd and 3rd terms} \\
AB + BC + BC & \\
\downarrow & \text{Applying identity } A + A = A \text{ to 2nd and 3rd terms} \\
AB + BC & \\
\downarrow & \text{Factoring B out of terms} \\
B(A + C) &
\end{array}
$$

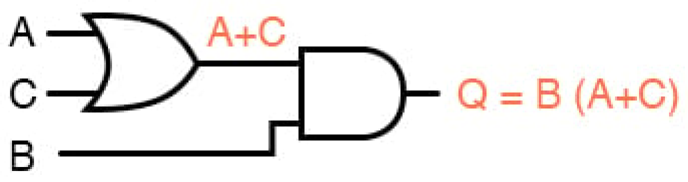
- ✓ The final expression, B(A + C), is much simpler than the original, yet performs the same function.

Generating Schematic Diagrams from Boolean Expressions

- ✓ Now, we must generate a schematic diagram from this Boolean expression.



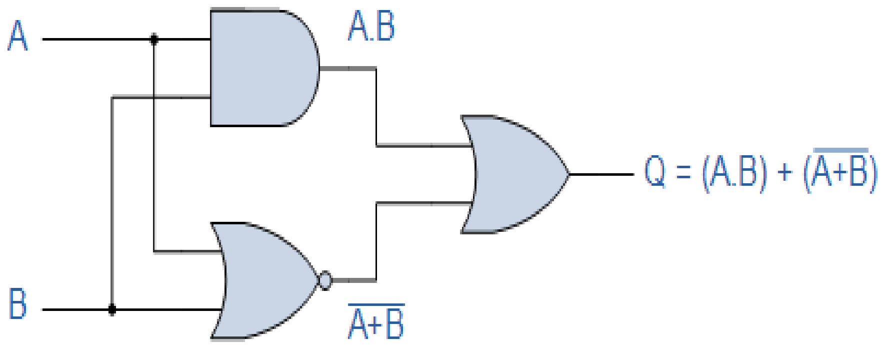
- ✓ The next step in evaluating the expression "B(A + C)" is to multiply (AND gate) the signal B by the output of the previous gate (A + C):



- ✓ Obviously, this circuit is much simpler than the original, having only two logic gates instead of five.

- ✓ Such component reduction results in higher operating speed (less delay time from input signal transition to output signal transition), less power consumption, less cost, and greater reliability.

$Q = (A.B) + (\overline{A+B})$



Inputs		Intermediates		Output
B	A	A.B	A + B	Q
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Simplified Boolean function using K-map method with DON'T CARE condition

- ✓ K-Map is a **graphical method of simplifying Boolean expression**.
- ✓ A K-Map **composed of an arrangement of adjacent squares or cells, where each cell represent a particular combination of variables in sum or product form**.
- ✓ In the K-map method, there is a useful condition namely, **Don't Care Condition, which helps in simplifying a Boolean function.** The don't care condition makes the grouping of variables in K-map easy.
- ✓ **It can be done using the K-map without using any theorems of Boolean algebra.**
- ✓ **The K-map can easily take two forms, namely, Sum of Product or SOP and Product of Sum or POS, according to what we need in the problem.**
- ✓ K-map is a representation that is table-like, **but it gives more data than the TRUTH TABLE.**
- ✓ Fill a grid of K-map with 1s and 0s, then solve it by creating various groups.

- **Rules of K-Maps**
  - Each cell with a **1 must be included** in at least one group.
  - Try to form the **largest** possible groups.
  - Try to end up with as few groups as possible.
  - Groups may be in sizes that are **powers of 2** :
  - The larger a group is, the more redundant inputs there are.

    (a) A group of 1 has no redundant inputs.
    (b) A group of 2, known as **Pair** has 1 redundant input.
    (c) A group of 4, known as **Quad** has 2 redundant input.
    (d) A group of 8, known as **Octet** has 3 redundant input.

Solving an Expression Using K-Map

1. Select a K-map according to the total number of variables.
2. Identify maxterms or minterms as given in the problem.
3. **For SOP, put the 1's in the blocks of the K-map with respect to the minterms (elsewhere 0's).**
4. **For POS, putting 0's in the blocks of the K-map with respect to the maxterms (elsewhere 1's).**
5. Making rectangular groups that contain the total terms in the power of two, such as 2,4,8 ..(except 1) and trying to cover as many numbers of elements as we can in a single group.
6. From the groups that have been created in step 5, find the product terms and then sum them up for the SOP form.

MIN TERM OR SUM OF PRODUCT(SOP) OR. STANDARD PRODUCT

- ✓ Consider if we have two binary variables like A and B
- ✓ if we perform And operation (*) between this two variable, so we can get four unique combination this combination is called is min term. ⌡ in min term always use And gate. ⌡ two variable min term as per follow.

A	B	MINTERM	DESIGNATION
0	0	A' B'	M0
0	1	A' B	M1
1	0	A B'	M2
1	1	A B	M3

- ✓ Three variable min term as per follow.

INPUTS			MINTERM	MIN TERM DESIGNATION
X	**Y**	**Z**	**STANDARD PRODUCT TERM**	
0	0	0	X’ Y’ Z’	M0
0	0	1	X’ Y’ Z	M1
0	1	0	X’ Y Z’	M2
0	1	1	X’ Y Z	M3
1	0	0	X Y’ Z’	M4
1	0	1	X Y’ Z	M5
1	1	0	X Y Z’	M6
1	1	1	X Y Z	M6

- ✓ Min term are also known as standard product.
- ✓ **The symbol of min term is ” mi”**

MAX TERM OR PRODUCT OF SUM(POS) OR. STANDARD SUM.

- ✓ Consider if we have two binary variables like A and B.
- ✓ If we perform OR operation (+) between this two variable, so we can get four unique combinations this combination is called is max term.

A	B	MAXTERM	DESIGNATION
0	0	A + B	M0
0	1	A + B’	M1
1	0	A’+ B	M2
1	1	A’+B’	M3

- ✓ Three variable max term as per follow.

INPUTS			MAXTERM	MAXTERM DESIGNATION
X	**Y**	**Z**	**STANDARD PRODUCT TERM**	
0	0	0	X + Y + Z	M0
0	0	1	X + Y + Z’	M1
0	1	0	X + Y’ + Z	M2
0	1	1	X + Y’ + Z’	M3
1	0	0	X + Y + Z	M4
1	0	1	X’ + Y + Z’	M5
1	1	0	X’ + Y’ + Z	M6
1	1	1	X’ + Y’ + Z’	M6

- ✓ Max term is also known as product of sum.
- ✓ **The symbol of max term is” Mi”.**

K-MAP for 2 variables

As shown in the fig. 1 the first row is for X’ and the second row for X. similarly, the first column is for Y’ and second column for Y.

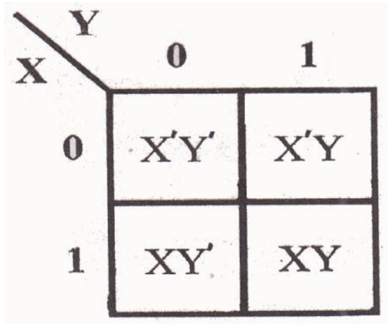


Fig. 1 Two variable K-Map.

- ✓ For any square, see the variables in the both the row and column. For 1st square of the 1st row, variables are X’ and Y’, and hence, it will represent the product term X’Y’. For the 2nd square of the 1st row, the variables are X’ and Y, so it represents X’Y.
- ✓ 0 and 1 are written at the top of the map shown in the above fig. they indicate 0 and 1 for variable Y. It means 0 represents variable in complemented form (Y’) and 1 represents variable in uncomplemented form (Y). Similarly, on the extreme left side of the map 0 and 1 are written and they represent X’ and X respectively.
- ✓ To understand properly, let us draw 2-variable K Map for the Boolean Function F=X’Y = XY’. As shown in Fig. only two product terms are present. So 1s are written in the corresponding squares.

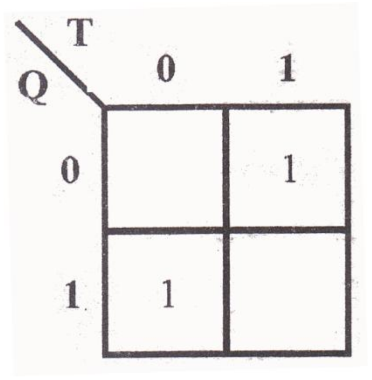
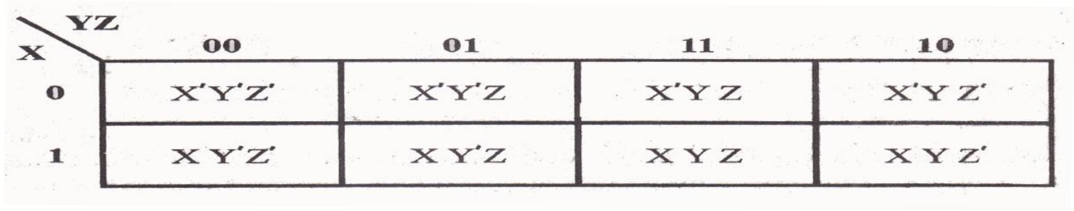


Fig. 2 Example of 1 variables K-Map
Solution : F=XY’ + X’Y

K-MAP for 3 variables.

- ✓ A three variable K-Map has eight square or cells and each square represent different product term X’Y’Z’, X’Y’Z, X’YZ’ X’YZ’, XY’Z’, XY'Z, XYZ and XYZ' respectively. Each of these sum of product term (minterm) designated by m_0, m_1, m_3, m_2, m_4, m_5, m_7, and m_8, respectively. (See table-2)



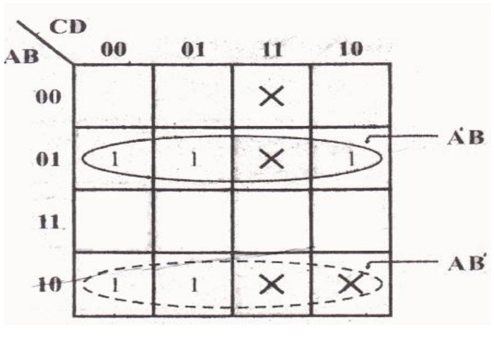
X \ YZ	00	01	11	10
0	X'Y'Z'	X'Y'Z	X'YZ	X'YZ'
1	XY'Z'	XY'Z	XYZ	XYZ'

Fig. 3 Three variable K-Map

- ✓ The binary numbers 00, 01, 11, and 10 written at the top of the map indicate the condition of Y and Z for each column.
- ✓ While binary number on the extreme left side of the map against each row indicates the condition the condition of X for that row.
- ✓ For example, the binary number 01 on the top of the second column in above fig indicates that the variable Y appears in complemented form and the variable Z in non-complemented form in all the minterms, in that column.
- ✓ Same way binary number 0 on the left of the first row indicates that the variable X appears in complemented form in all the minterms in that row.
- ✓ Observe that the ordering of the variables written on the top of the map is 00, 01, 11, and 10. One should not write straight binary codes, i.e., 00,01, 10, and 11.

Don't Cares

- ✓ Don't cares in a K map, or truth table, may be either 1s or 0s, as long as we don't care what the output is for an input condition we never expect to see.
- ✓ **We plot these cells with an asterisk, *, among the normal 1s and 0s.**
- ✓ In K map every cell represents a minterm (or maxterm). '1' or '0' in a cell means the given function produces '1' or '0' for the minterm.
- ✓ **There are many occasions when it doesn't matter whether the function output is '1' or '0' for the minterm.**
- ✓ **Minterm that may produce either "0" or "1" are said to be don't care condition and generally mark with "X" in the map.**
- ✓ This don't care condition can be used to provide further simplification of the algebraic expression.



AB \ CD	00	01	11	10
00			×	
01	1	1	×	1
11				
10	1	1	×	×

- ✓ The important point is that an "X" need not be used at all if it doesn't contribute to the simplification of the function.
- ✓ In each case the choice depends only on the simplification which can be achieved.

Sequential and Combinational Circuits

What is a Combinational Circuit?

- ✓ **The output of a Combinational Circuit depends entirely on the present input.**
- ✓ **It exhibits a faster speed.**
- ✓ It is comparatively easier to design.
- ✓ **No feedback is present between the input and output.**
- ✓ **The combinational circuit depends on time.**
- ✓ They do not possess any memory element.
- ✓ Users can feasibly use as well as handle them.
- ✓ The example of Combinational Circuit includes:
    - Half Adder
    - Full Adder
    - Half Subtractor
    - Full Subtractor

- ✓ **Advantages of Combinational Circuits**
    - **Simplicity:** It is easier to design and implement more so because it lacks memory elements mainly.
    - **Speed:** Operational at a faster rate as the output automatically adjusts with the changes in inputs.
    - **Resource Efficiency:** Generally, it needs far fewer components as compared to its equivalent sequential circuits.

- ✓ **Disadvantages of Combinational Circuits**
    - **Limited Functionality:** Is not able to perform operations that need historical information or sequence details.
    - **Complexity with Increased Inputs:** It becomes difficult to design combinational circuits when there are many inputs.

What is a Sequential Circuit?

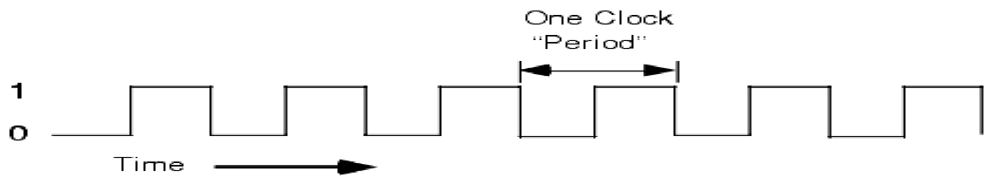
- ✓ **The output of a Sequential Circuit depends on both- past as well as present inputs.**
- ✓ **It works at a comparatively slower speed.**

- ✓ The design of these circuits is comparatively **much tougher than the Combinational Circuit.**
- ✓ **The circuit is time-dependent.**
- ✓ Flip-flops constitute the building blocks of such a circuit.
- ✓ **People mainly use them for storing data and information.**
- ✓ **Because a Sequential circuit depends on a clock, it usually requires triggering.**
- ✓ A user may not be able to handle and use these circuits easily.
- ✓ Examples of a Sequential circuit are:
    - **Flip-Flops**
    - Registers
    - Counters

- ✓ **Advantages of Sequential Circuits**
    - **Memory Utilization:** Able to store the previous states in order to perform the operations.
    - **Functional Versatility:** It is used for those tasks which need a series of operations, state machines, and counters.
- ✓ **Disadvantages of Sequential Circuits**
    - **Complexity:** More difficult to design compared to counter-propagators due to the use of memory elements.
    - **Slower Operation:** Output changes may represent a delay because often they require data processing from the past.

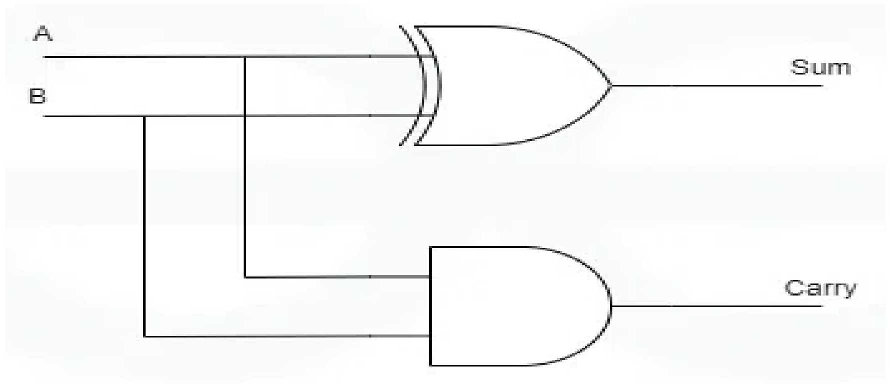
Clock Signal

- ✓ A clock signal is a particular type of signal that oscillates between a high and low state.
- ✓ With the signal acting as a metronome, the digital circuit follows in time to coordinate its sequence of actions.
- ✓ Digital circuits rely on clock signals to know when and how to execute the functions that are programmed.
- ✓ If the clock in a design is like the heart of an animal, then clock signals are the heartbeats that keep the system in motion.



Half Adder

- ✓ Half Adder is a combinational logic circuit that is **designed by connecting one EX-OR gate and one AND gate.**
- ✓ **The half-adder circuit has two inputs: A and B, which add two input digits and generate a carry and a sum.**
- ✓ The output obtained from the EX-OR gate is the sum of the two numbers while that obtained by AND gate is the carry.
- ✓ There will be no forwarding of carry addition because there is no logic gate to process that. Thus, this is called the Half Adder circuit.



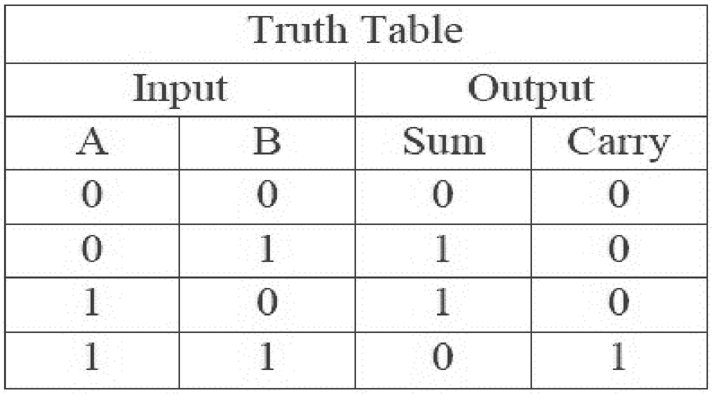
Logical Expression of Half Adder

Sum = A ⊕ B

Carry = A AND B

Truth Table of Half Adder

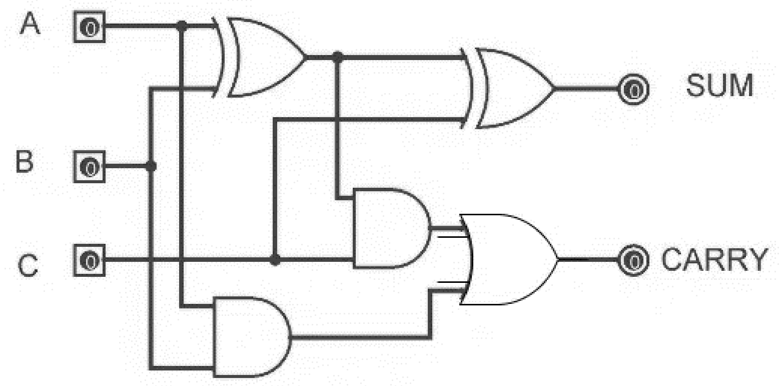
The Truth Table for Half Added is Given as



Truth Table			
Input		Output	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder

- ✓ Full Adder is the circuit that **consists of two EX-OR gates, two AND gates, and one OR gate.**
- ✓ **Full Adder is the adder that adds three inputs and produces two outputs which consist of two EX-OR gates, two AND gates, and one OR gate.**
- ✓ The first two inputs are A and B and the third input is an input carry as C-IN.
- ✓ The output carry is designated as C-OUT and the normal output is designated as S which is SUM.
- ✓ The equation obtained by the EX-OR gate is the sum of the binary digits. While the output obtained by AND gate is the carry obtained by addition.



- ✓ Logical Expression of Full Adder
- ✓ Given Below is the Logical Expression of Full Adder
  - o SUM = (A XOR B) XOR Cin = $(A \oplus B) \oplus Cin$
  - o CARRY-OUT = A AND B OR Cin(A XOR B) = $A.B + Cin(A \oplus B)$
- ✓ Truth Table of Full Adder

Input			Output	
A	B	Cin	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

Sum and Carry Operation

- ✓ In both Half Adders and Full Adders, **Sum is the output of the addition of the two inputs and Carry is the output which is an overflow of the output position and needs to be shifted to the next higher position while adding successive bit inputs.**
- ✓ **Sum (S) :** It results from the XOR gate, which is a logic gate that adds two or more bits together in the same way that you add in base 2 with no acknowledgement of carry from the previous bit.
- ✓ **Carry (C or Cout):** It is the output of the AND operation in the case of the Half Adder or both AND and OR Operations in the case of the Full Adder to indicate that a '1' has to be carried over to the next bit position.

Advantages and Disadvantages

Advantages of Half Adder

Flexible and easy when it comes to design.
Involves the use of fewer logic gates thus, is cheaper.

Disadvantages of Half Adder

Fails to process a carry input from the previously added numbers.
Restricted to the addition of only two bits.

Advantages of Full Adder

Can add 3 bits, it includes one carry input and a carry output, which can perform more elaborate computations.
It can be cascaded to produce adders for a number of bit additions which makes it suitable for multi bit arithmetic.

Disadvantages of Full Adder

Complex and needs more gates, hence making the design more complicate and expensive.
Slightly slower because normally 2 gate process are used instead of 1.

Difference Between Half Adder and Full Adder

Parameters	Half Adder	Full Adder
Description	Half Adder is a combinational logic circuit that adds two 1-bit digits. The half adder produces a sum of the two inputs.	A full adder is a combinational logic circuit that performs an addition operation on three one-bit binary numbers. The full adder produces a sum of the three inputs and carry value.

Previous carry	The previous carry is not used.	The previous carry is used.
Inputs	In Half adder, there are two input bits (A, B).	In full adder, there are three input bits (A, B, C-in).
Outputs	The generated output is of two bits-Sum and Carry from the input of 2 bits.	The generated output is of two bits-Sum and Carry from the input of 3 bits.
Used as	A half adder circuit cannot be used in the same way as a full adder circuit.	A full adder circuit can be used in place of a half adder circuit.
Feature	It is simple and easy to implement	The design of a full adder is not as simple as a half adder.
Logic gates	It consists of one EX-OR gate and one AND gate.	It consists of two EX-OR, two AND gates, and one OR gate.
Alternate name	There is no alternate name for half adder.	Full adder is also known as ripple-carry adder

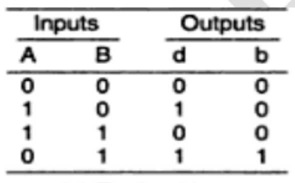
Subtractors:

- The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend.
- By this, the subtraction operation becomes an addition operation and instead of having a separate circuit for subtraction, the adder itself can be used to perform subtraction.
- This results in reduction of hardware. In subtraction, each subtrahend bit of the number is subtracted from its corresponding significant minuend bit to form a difference bit.
- If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position., that has been borrowed must be conveyed to the next higher pair of bits by means of a signal coming out (output) of a given stage and going into (input) the next higher stage.

1. The Half-Subtractor

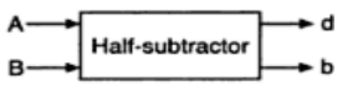
- ✓ A Half-subtractor is a combinational circuit that subtracts one bit from the other and produces the difference.
- ✓ It also has an output to specify if a 1 has been borrowed. It is used to subtract the LSB of the subtrahend from the LSB of the minuend when one binary number is subtracted from the other.

- ✓ A Half-subtractor is a combinational circuit with two inputs A and B and two outputs d and b. d indicates the difference and b is the output signal generated that informs the next stage that a 1 has been borrowed.
- ✓ When a bit B is subtracted from another bit A, a difference bit (d) and a borrow bit (b) result according to the rules given as



Inputs		Outputs	
A	B	d	b
0	0	0	0
1	0	1	0
1	1	0	0
0	1	1	1

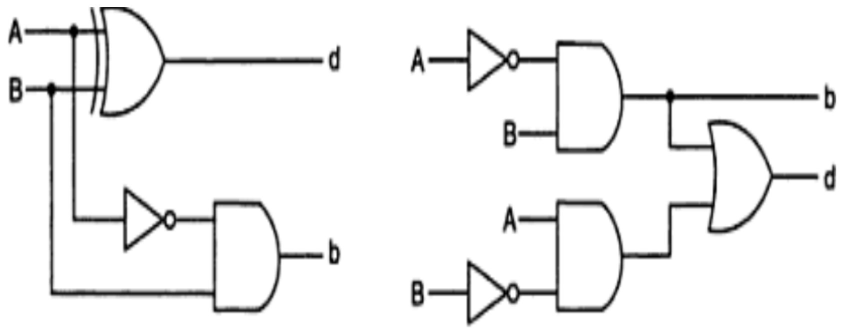
(a) Truth table



(b) Block diagram

Half-subtractor.

- ✓ The output borrow b is a 0 as long as A≥B. It is a 1 for A=0 and B=1. The d output is the result of the arithmetic operation2b+A-B.
- ✓ A circuit that produces the correct difference and borrow bits in response to every possible combination of the two 1-bit numbers is, therefore,
- ✓ **Diff = A´B+ AB´ and borr = Ā B**



Logic diagrams of a half-subtractor.

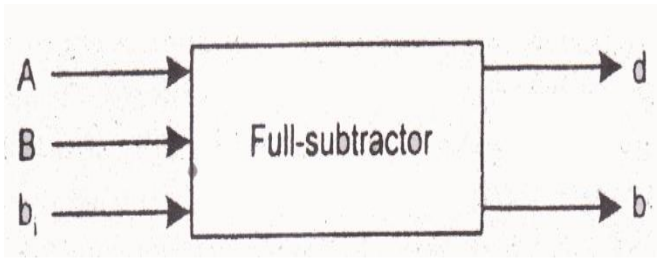
2. The Full-Subtractor:

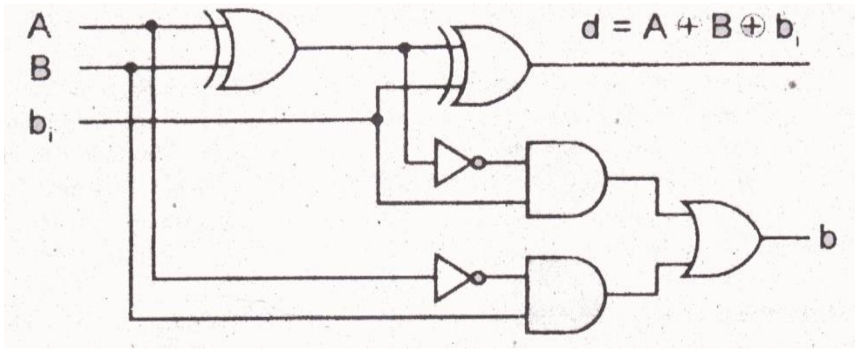
- ✓ The half-subtractor can be only for LSB subtraction.
- ✓ IF there is a borrow during the subtraction of the LSBs, it affects the subtraction in the next higher column; the subtrahend bit is subtracted from the minuend bit, considering the borrow from that column used for the subtraction in the preceding column.
- ✓ Such a subtraction is performed by a full-subtractor. It subtracts one bit (B) from another bit (A), when already there is a borrow bi from this column for

the subtraction in the preceding column, and outputs the difference bit (d) and the borrow bit(b) required from the next d and b.

- ✓ The two outputs present the difference and output borrow. The 1s and 0s for the output variables are determined from the subtraction of A-B-bi

Inputs			Difference	Borrow
A	**B**	**B_i**	**D**	**B**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1





$d = A \oplus B \oplus bi$

Flip Flops

flip-flops are bistable multivibrators used to store binary information. They are the building blocks for memory storage and timing circuits. Here are the main types of flip-flops:

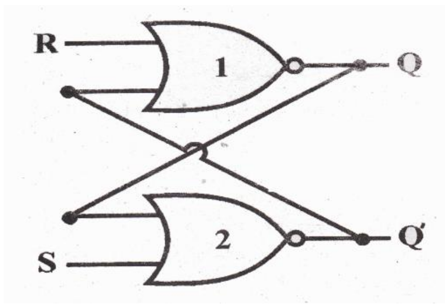
1. **SR Flip-Flop (Set-Reset Flip-Flop)**:
   - The simplest type of flip-flop, having two inputs: Set (S) and Reset (R).
   - It has two states: Set (1) and Reset (0).
   - If both Set and Reset are 1 simultaneously, it leads to an invalid state.
   - **Clocked R – S Flip Flop**
     1. In the simple R-S flip-flop (described earlier), the output is depending on input condition that is at any time the input conditions are changed, output is also changed. Therefore, they are called as asynchronous flip-flops.
     2. The clocked R-S flip-flop requires a clocked (Enabled) input. It means that this type of flip-flop has three inputs, named as S (Set), R (Reset), and C (Clock). Its S and R inputs will control the state of the flip-flop only when the clock inputs are high. When the clock is low, the inputs become ineffective and no change of state can take place.

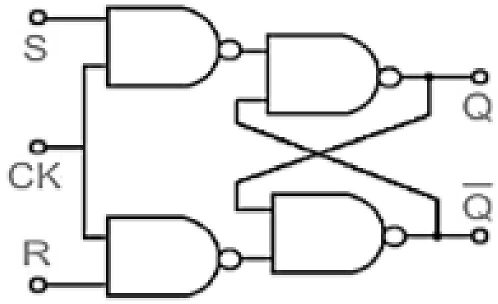
```
       _______
S ----|      |---- Q
      |      |
R ----|      |---- Q'
      ------
```

Truth Table:

S	**R**	**Q**	**Q'**
0	0	Q	Q'
0	1	0	1
1	0	1	0
1	1	Invalid	Invalid

- **Explanation:**
  - When S = 0 and R = 0, the output Q remains in its previous state (memory).
  - When S = 0 and R = 1, the output Q is reset to 0.
  - When S = 1 and R = 0, the output Q is set to 1.
  - When S = 1 and R = 1, the output is invalid (not allowed).





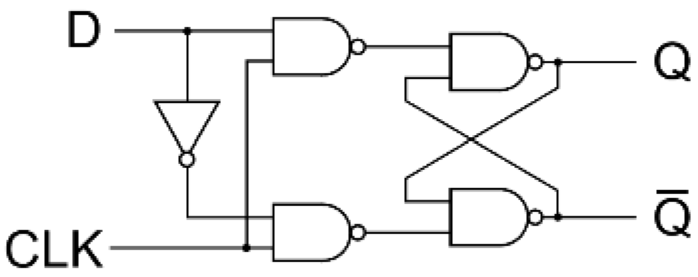
2. **D Flip-Flop (Data or Delay Flip-Flop)**:
    - It has a single data input (D) and a clock input.
    - The output follows the D input at the rising or falling edge of the clock signal, making it simpler to use than the SR flip-flop.

```
        _______
D ----|      |---- Q
        |      |
Clk --|      |---- Q'
        ------
```

Truth Table:

D	Clock	Q	Q'
0	↑	0	1
1	↑	1	0

- **Explanation:**
    - The output Q follows the input D on the rising edge of the clock signal.
    - When the clock is high, the state of Q is equal to D.



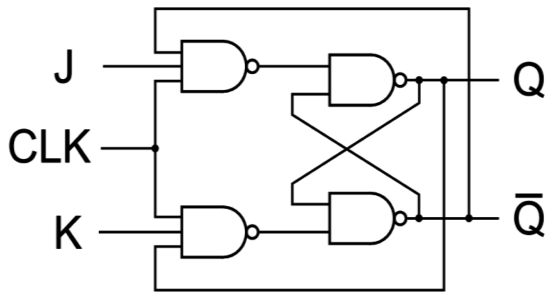
3. **JK Flip-Flop**:
   - It has two inputs: J and K, along with a clock input.
   - It is a more versatile flip-flop, as it solves the invalid state problem of the SR flip-flop.
   - When both J and K are 1, it toggles its state.

```
       _______
J ---- |     |---- Q
       |     |
K ---  |     |---- Q'
       |     |
Clk --|      |
        ------
```

Truth Table:

J	K	Clock	Q	Q'
0	0	↑	Q	Q'
0	1	↑	0	1
1	0	↑	1	0
1	1	↑	Toggle	Toggle

- **Explanation:**
  - When J = 0 and K = 0, the output remains the same (no change).
  - When J = 0 and K = 1, the output Q is reset to 0.
  - When J = 1 and K = 0, the output Q is set to 1.
  - When J = 1 and K = 1, the output toggles its state every clock pulse.



4. **JK Master-Slave Flip-Flop**
   - **JK Master-Slave Flip-Flop** is a combination of two JK flip-flops connected in a way that the output of the first (master) flip-flop serves as the input to the second (slave) flip-flop. This arrangement helps eliminate the possibility of invalid states and improves the overall reliability of the circuit.
   - In the JK Master-Slave flip-flop, there are two stages:

1. **Master Flip-Flop**: This flip-flop captures the inputs when the clock signal is low (or inactive).
2. **Slave Flip-Flop**: This flip-flop changes its output based on the master's output when the clock signal is high (or active).

- The idea behind this design is to prevent both flip-flops from changing state at the same time, which could lead to glitches or instability. The **Master** flip-flop only responds to inputs when the clock is low, and the **Slave** flip-flop responds when the clock is high.

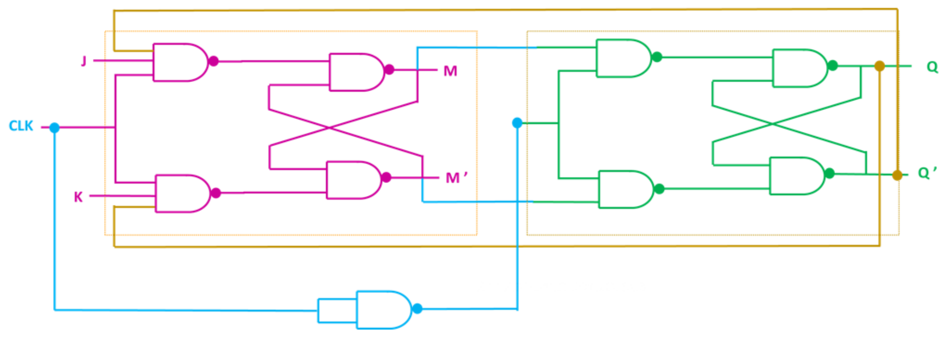
```
       ______________      ________
  J ----|            |--- Q    |          |--- Q
        |Master  |             |  Slave |
 K ---- |   JK     |--- Q'   |   JK    |--- Q'
        |________|           |_______|
            |                       |
           Clk                     Clk
            |                       |
          (low)                  (high)
```

Truth Table of JK Master-Slave Flip-Flop

J	K	Clock (↑)	Master Q	Slave Q (Output)	Slave Q' (Output)
0	0	↑	Q	Q	Q'
0	1	↑	0	0	1
1	0	↑	1	1	0
1	1	↑	Toggle	Toggle	Toggle

Explanation:

- When the clock signal rises (clock edge), the **slave** flip-flop takes the value from the **master** flip-flop and updates its output based on the inputs **J** and **K**.
- The **Master** flip-flop latches the inputs only when the clock is **low**.
- The JK Master-Slave flip-flop behaves similarly to a JK flip-flop but ensures the outputs are stable because the clock edges are controlling when the flip-flops update.



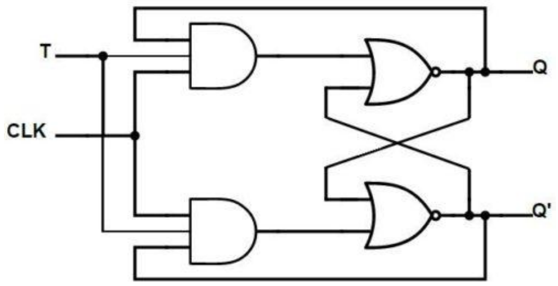
5. **T Flip-Flop (Toggle Flip-Flop)**:
    - It has a single input, T (Toggle), and a clock input.
    - The output toggles its state every time the T input is 1 at the clock edge.
    - It can be derived from a JK flip-flop by connecting both J and K inputs together.

```
         ______
T ----|     |---- Q
         |     |
Clk --|     |---- Q'
         ------
```

Truth Table:

T	**Clock**	**Q**	**Q'**
0	↑	Q	Q'
1	↑	Toggle	Toggle

- **Explanation:**
    - When T = 0, the output remains the same (no change).
    - When T = 1, the output toggles between 0 and 1 with each clock pulse.

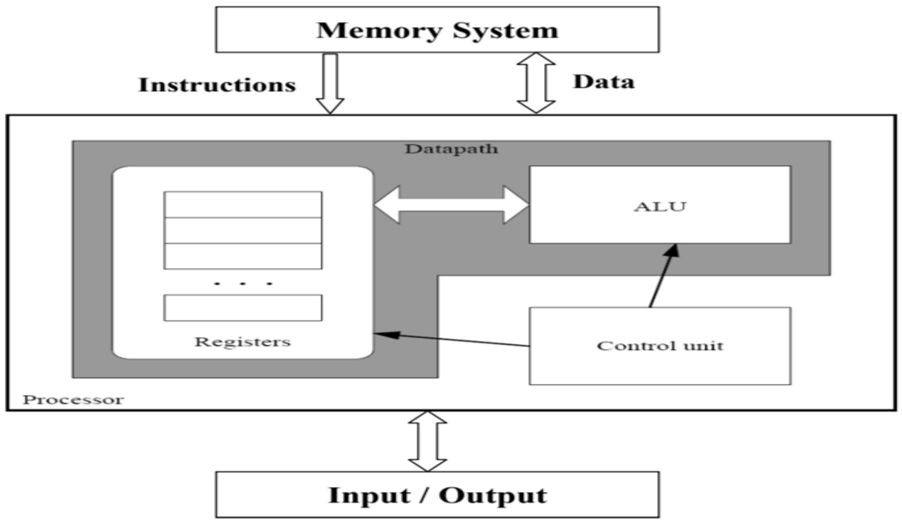


UNIT 2- CENTRAL PROCESSING UNIT

- Introduction of CPU
- Major component of CPU
- General register organization
    - Control word
    - Accumulator register
- Stack organization
    - Register stack
    - Memory stack
    - Polish notation and reverse polish notation
- Arithmetic and logic unit
    - Block diagram of ALU
- Interrupts

- **Introduction of CPU**
  - The CPU is the brain of a computer, **containing all the circuits** needed **to process input, store data, and output results.**
  - The CPU is constantly **following instructions of computer programs** that tell it which data to process and how to process it.
  - Without a CPU, we could not run programs on a computer. Because the CPU performs many different functions, you need to divide it up into its main parts to understand it.
  - Only then can you successfully describe what it does. It controls the functioning of the other units and processes the data.
  - **The CPU is sometimes called the processor, or in the personal computer field called "microprocessor".**
  - **It is a single integrated circuit that contains all the electronics needed to execute a program.**
  - The processor calculates (add, multiplies and so on), performs logical operations (compares numbers and make decisions), and controls the transfer of data among devices.
  - The processor acts as the controller of all actions or services provided by the system.
  - Processor actions are synchronized to its clock input. A clock signal consists of clock cycles.
  - **The time to complete a clock cycle is called the clock period.** Normally, we use the clock frequency, which is the inverse of the clock period, to specify the clock.
  - **The clock frequency is measured in Hertz, which represents one cycle/second. Hertz is abbreviated as Hz.**
  - Usually; we use mega Hertz (MHz) and giga Hertz (GHz) as in 1.8 GHz Pentium.
  - The processor can be thought of as executing the following cycle forever:
    - **Fetch an instruction from the memory,**
    - **Decode the instruction (i.e., determine the instruction type),**
    - **Execute the instruction (i.e., perform the action specified by the instruction).**
- **Major component of CPU**
  - A typical CPU has three major components:
    - Register set,
    - Arithmetic logic unit (ALU),
    - Control unit (CU).

- **The register set differs from one computer architecture to another.**
- It is usually a combination of general-purpose and special purpose registers.
- **General-purpose registers are used for any purpose, hence the name general purpose.**
- Special purpose registers have specific functions within the CPU. For example, the **program counter (PC) is a special-purpose register that is used to hold the address of the instruction to be executed next.**
- Another example of special-purpose registers is the instruction register (IR), which is used to hold the instruction that is currently executed.
- Figure shows the main components of the CPU and its interactions with the memory system and the input/output devices.



- The ALU provides the circuit needed to perform the arithmetic, logical and shift operations demanded of the instruction set.
- The control unit is the entity responsible for fetching the instruction to be executed from the main memory and decoding and then executing it.
- The CPU can be divided into a data section and a control section.
- **The data section, which is also called the data path, contains the registers (known as the register file) and the ALU.**
- The data path is capable of performing certain operations on data items. The register file can be thought of as a small, fast memory, separate from the system memory, which is used for temporary storage during computation.

- The **control section is basically the control unit, which issues control signals to the data path.**
- The control unit of a computer is responsible for executing the program instructions, which are stored in the main memory.
- It can be thought of as a form of a "computer within a computer" in the sense that it makes decisions as to how the rest of the machine behaves.

- **General register organization**
  - In General Register Organization, the CPU uses a set of general-purpose registers to store data temporarily during program execution.
  - These registers are small, high-speed storage locations within the CPU.
  - **Unlike accumulator-based architectures, where data is temporarily stored in a single accumulator, this organization uses multiple registers, each capable of storing different types of data such as integers, floating-point numbers, addresses, and control information.**
  - The use of these general-purpose registers allows for faster data processing since accessing data from registers is significantly faster than fetching it from memory.
  - **Types of General Register Organization**
    - **Register - Memory Reference Architecture**
      - In this type of organization, the CPU has a relatively smaller number of registers.
      - One operand must always be located in a register, while the second operand can either be in a register or memory.
      - This organization typically uses two-address instruction formats, where one address refers to a register and the other may refer to either a register or a memory location.
    - **Register - Register Reference Architecture**
      - This organization utilizes a larger number of general-purpose registers and typically employs three-address instruction formats.
      - In this case, all operands involved in the arithmetic or logical operation must be in the registers. The results of computations are also stored in registers.
- **Control word**
  - In General Register Organization, the control word refers to the specific set of signals used by the CPU to manage register operations.

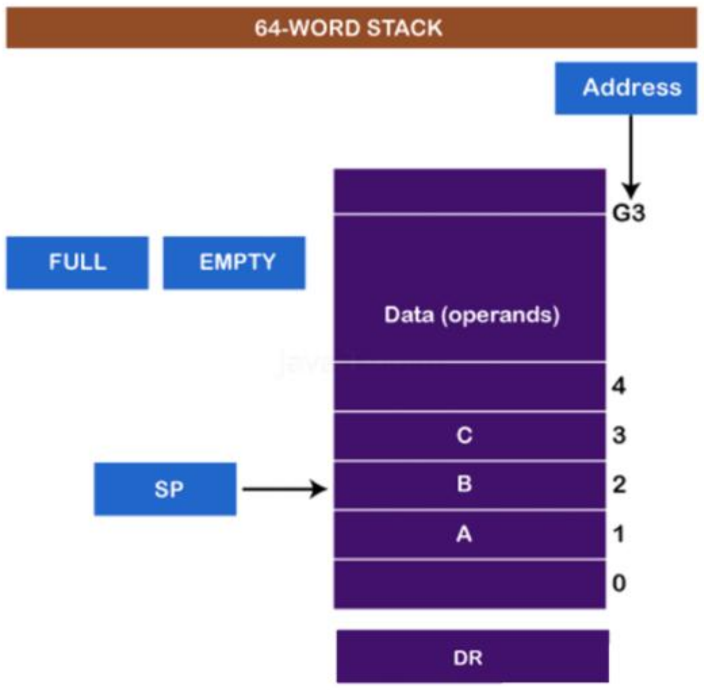
- These signals control which registers are read, written to, or updated during the execution of an instruction.

- **Accumulator register**
    - An accumulator is a **type of register for short-term, intermediate storage of arithmetic and logic data in a computer's central processing unit (CPU).**
    - However, the term is rarely used in reference to contemporary CPUs, having been replaced around the turn of the millennium by the term register.
    - **Function:**
        - ❖ **Temporary Storage:** The accumulator holds data during calculations, allowing the CPU to perform multiple operations without having to constantly access main memory.
        - ❖ **Implicit Operand:** In some CPU architectures, the accumulator serves as an implicit operand for arithmetic and logical instructions, meaning the instruction automatically uses the accumulator as a source or destination for data.
        - ❖ **Efficiency:** Using an accumulator can improve the speed and efficiency of computations by reducing the need to access slower main memory for intermediate results.
    - **Example:**
        - Imagine you want to perform the calculation X = Y + Z.
        - In an accumulator-based architecture, the process might look like this:
            - Load the value of Y into the accumulator.
            - Add the value of Z to the accumulator.
            - Store the result (now in the accumulator) into variable X.

➢ **Stack organization**

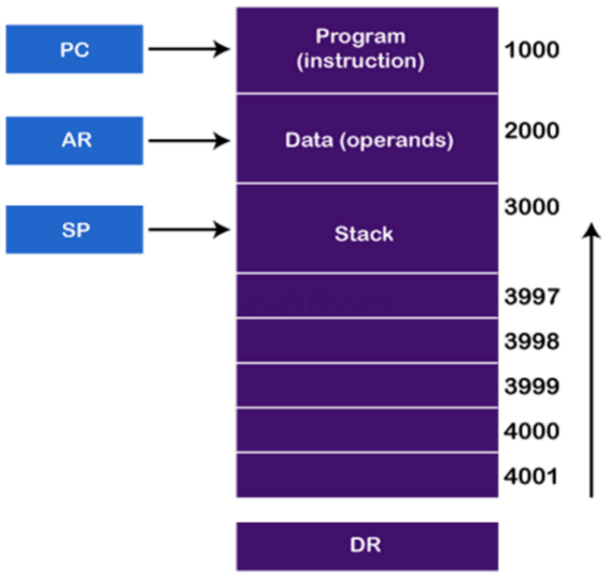
- ❖ A stack is a linear data structure that stores data in a sequential manner.
- ❖ Data is **added to or removed from one end, commonly referred to as the top of the stack.**
- ❖ The **primary operations on a stack are Push (adding an element) and Pop (removing an element).**
- ❖ Stack organization refers to the way a computer system uses a stack for managing execution flow, storing temporary data, and performing arithmetic or logical operations. The stack is implemented as a portion of memory, with a stack pointer (SP) that tracks the current top of the stack.
- ❖ The primary operations in a stack organization are:
- ❖ Push: Pushes an element to the top of the stack.
- ❖ Pop: Removes the topmost element from the stack.

- In stack organization, when an operation is performed, the stack pointer is updated. This makes stack operations efficient and allows for orderly memory access.
- **Types of Stack Organization**
    - There are two primary types of stack organization in computer architecture:
    - **Register Stack**
        - In a register stack, memory words or registers are placed on top of each other. The address of the top element is stored in the stack pointer register. Each time an element is added (pushed), the stack pointer is incremented. When an element is removed (popped), the pointer is decremented.



- Where,
- **Stack Pointer (SP):** Points to the top of the stack.
- **Data Register (DR):** Holds data being transferred.
- **Full:** The stack is at maximum capacity and cannot hold more data (stack overflow).
- **Empty:** The stack contains no data, and no items can be popped (stack underflow).

- **Memory Stack**
  - A memory stack is created by reserving a portion of memory for the stack.
  - The stack pointer (SP) points to the current top of the stack in memory.
  - The stack pointer register keeps track of the top of the stack, and elements are inserted or removed using this pointer.



  - Where,
  - **Program Counter (PC):** Holds the address of the next instruction.
  - **Address Register (AR):** Stores memory addresses.
  - **Stack Pointer (SP):** Points to the top of the stack.
  - **Data Register (DR):** Holds data being transferred.

- **Polish notation and reverse polish notation**
  - ❖ The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –
    - Infix Notation
    - Prefix (Polish) Notation
    - Postfix (Reverse-Polish) Notation

❖ These notations are named as how they use operator in expression.

A. Infix Notation

- We write expression in infix notation, e.g. a - b + c, where operators are used in-between operands.
- It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices.
- An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.
- Infix is the day to day notation that we use of format A + B type. The general form can be classified as (a op b) where a and b are operands(variables) and op is Operator.
    - Example 1 : A + B
    - Example 2 : A * B + C / D

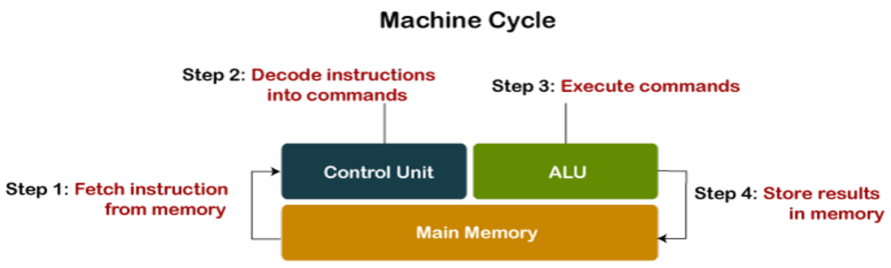
B. Prefix Notation (Polish)

- In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands.
- For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.
- Prefix is notation that compiler uses/converts to while reading right to left (some compilers can also read prefix left to right) and is of format +AB type.
- The general form can be classified as (op ab) where a and b are operands(variables) and op is Operator.
    - Example 1 : +AB

C. Postfix Notation (Reverse-Polish)

- This notation style is known as Reversed Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+.
- This is equivalent to its infix notation a + b.Postfix is notation that compiler uses/converts to while reading left to right and is of format AB+ type.
- The general form can be classified as (ab op) where a and b are operands(variables) and op is Operator.
    - Example 1 : AB+

- **Arithmetic and logic unit**
    - In computing **arithmetic logic unit (ALU) is a digital circuit that performs arithmetic and logical operations.**
    - The ALU is a **fundamental building block of the central processing unit (CPU)** of a computer, and even the simplest microprocessors contain one for purposes such as maintaining timers.
    - **In some processors, the ALU is divided into two units: an arithmetic unit (AU) and a logic unit (LU).**
    - Some processors contain more than one AU -- for example, one for fixed-point operations and another for floating-point operations.
    - It has the ability to perform all processes related to arithmetic and logic operations such as addition, subtraction, and shifting operations, including Boolean comparisons (XOR, OR, AND, and NOT operations).
    - When the ALU completes the processing of input, the information is sent to the computer's memory.


Machine Cycle
Step 2: Decode instructions into commands
Step 3: Execute commands
Step 1: Fetch instruction from memory
Control Unit
ALU
Step 4: Store results in memory
Main Memory

- **Block diagram of ALU**
    - Arithmetic logic unit (ALU) is a circuit board embedded within a computer's central processing unit (CPU), which performs mathematical and logical operations using gateways made of electrical transistors that can convey signals in 0s and 1s.

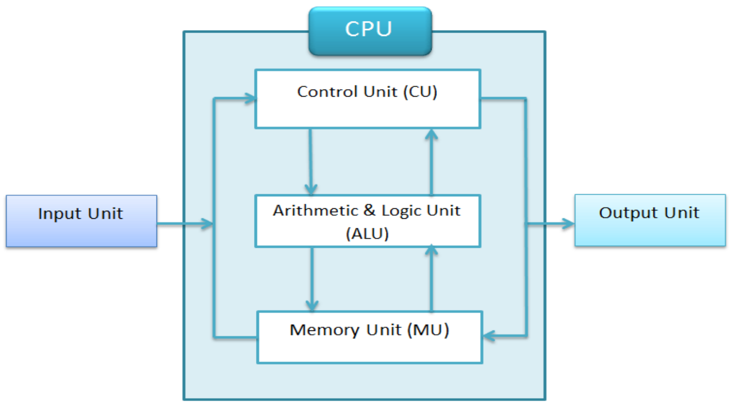
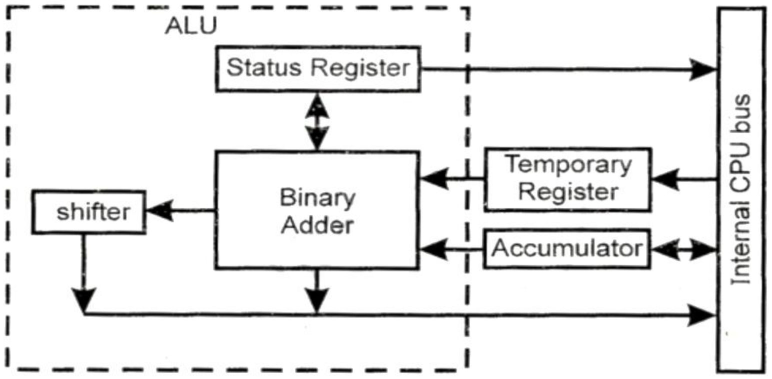


Illustration of ALU

- At its core, the ALU is a digital combinational circuit that executes arithmetic and bitwise functions on integer binary variables.
- The organization of arithmetic and logic unit is shown in the figure



- The arithmetic and logic unit is an 8-bit unit.
- The ALU contains
  - Adder
  - Shifter
  - Status register
- It performs arithmetic, logic and rotates operations.
- It consists of the binary adder to perform addition and subtraction by 2's complement method.
- The result is typically stored in an accumulator.
- Accumulator, temporary register and flag register are closely associated with A.L.U.
- The temporary register is used to hold data during an arithmetic/logic operation.
- The flags are set or reset according to the result of operations in the status register.

➢ **Interrupts**

- ❖ The concept of **program interrupt is use to handle variety of problems that arrives out of normal program sequence.**
- ❖ Program interrupt refers to the **transfer of program control from a currently running program to another service program** as a result of an external or internal generated request.
- ❖ The interrupt facility is **useful in multi programming environment when two or more program is resided in the memory at same time.**
- ❖ The function of interrupt facility is to take care of data transfer of one or more program while another program is currently being executed.
- ❖ **Advantage**
  - The advantage of interrupt is waiting time of input output program is eliminated.

- **Types of Interrupts**
- There are three major type of interrupt that cause break in the normal execution program.
    - External
    - Internal
    - Software.
- **External Interrupt**
    - The external interrupt **caused by external events**, they come from input-output devices, timing devices or power supply or any other external sources.
    - Examples that because external interrupt are input/output device requesting transfer of data input/output device finish transfer of data, or power failure.
    - **This interrupt is initiated from signal that occurs in the hardware of the C.P.U. – That's why they are also known as hardware interrupts.**
- **Internal Interrupt**
    - Internal interrupt is **initiated by some exceptional condition cause by the program itself.**
    - Examples that cause the internal interrupts are register overflow, stack overflow, attempt to divide by zero and invalid operation code etc.
    - This interrupts are synchronous with program. If the program is return the internal interrupts will occur in the same place each time.
    - **Internal interrupts are synchronous with the program while external interrupts are asynchronous.**
    - External interrupts depend on external conditions that are independent of the program being executed at the time.
- **Software Interrupt**
    - Software interrupts initiated by executing an instruction.
    - The most common use of this interrupt is associated with supervisor call instruction.
    - This instruction provides switching from C.P.U. user mode to the supervisor mode.
    - When input/output transfer is required the supervisor mode is requested by means of supervisor call instruction.
    - This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode.

UNIT 3- INPUT OUTPUT ORGANIZATION

- Memory buses
- Block diagram and function
- Data bus, address bus and control lines
- Input output buses
- Concept of input output interface
- Input Output Processor (IOP)
- Direct memory access
- DMA controller

- **Memory buses**
  - The Memory Bus, also known as the memory controller or memory interface, is a subsystem within a computer or device that facilitates communication between the Central Processing Unit (CPU) and memory.
  - It acts as a conduit, transmitting data and control signals between the CPU and memory modules.
  - The speed and width of the memory bus can impact a computer's overall performance, as it directly influences the rate at which data can be transferred between components.
  - **Key Points**
    1. The memory bus connects the central processing unit (CPU) to the main memory (RAM) in a computer system, responsible for transmitting data and addressing information between them.
    2. Memory bus width, or the number of lines used for transferring data, determines the amount of data that can be transferred simultaneously, ultimately affecting the overall system performance.
    3. Memory bus speed, or clock rate, is another factor that directly impacts system performance by influencing the rate at which data is communicated between the CPU and RAM.
- **Block diagram and function**
  - A memory bus in a computer system is a communication pathway that connects the CPU, memory, and other components.
  - It is essential for transferring data between the processor and memory, enabling read and write operations.
  - The memory bus typically consists of several distinct lines that serve different functions, such as data, address, and control lines.

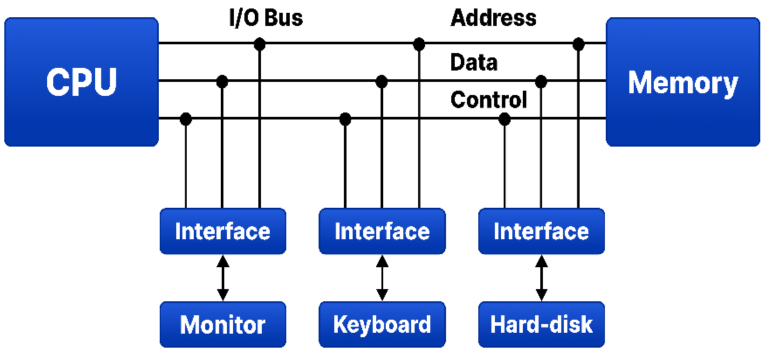
```
          +---------------+
          |    CPU        |
         +---------------+
                | (Address, Data, Control)
          +------------------+
          |  Memory Bus    |
          +------------------+
           /            |             |       \
 +------------+  +------------+  +------------+
 |  Address   | |   Data    | |  Control   |
 |  Bus        | |   Bus      | |  Lines     |
 +------------+  +------------+  +------------+
    +-----------------+
    |     Memory       |
    |   (RAM/ROM)  |
    +-----------------+
```

❖ **Data bus, address bus and control lines**

1. **Data Bus:**
   - **Function:** The data bus is responsible for transferring actual data between the CPU, memory, and other peripherals.
   - **Type:** It is bi-directional, meaning data can flow both ways (from memory to CPU or vice versa).
   - **Width:** The width of the data bus (number of bits) determines how much data can be transferred at once. For example, a 16-bit data bus can transfer 16 bits of data at a time, whereas a 32-bit bus can transfer 32 bits at a time. Modern systems typically use 64-bit data buses.
   - **Example:** When the CPU reads data from memory, the data travels over the data bus.
2. **Address Bus:**
   - **Function:** The address bus is responsible for carrying memory addresses from the CPU to memory or I/O devices. This tells the system where data should be read from or written to.
   - **Type:** The address bus is typically **unidirectional**, meaning the address only travels in one direction (from CPU to memory).
   - **Width:** The width of the address bus determines how much memory the system can address. For example, a 32-bit address bus can address 2^32 (about 4 billion) unique memory locations, whereas a 64-bit address bus can address 2^64 locations (much larger).
   - **Example:** When the CPU wants to read or write to a specific memory location, it places the address of that location on the address bus.
3. **Control Lines:**
   - **Function:** Control lines are responsible for coordinating and managing the various operations of the system, such as read and write operations, clock signals, and other control functions.
   - **Type:** These lines are used to control the flow of data and specify the type of operation to be performed. Control lines can be unidirectional or bidirectional, depending on the specific function.
   - **Examples of control lines:**
   - **Read/Write (R/W):** Indicates whether the operation is a read or write.
   - **Chip Select (CS):** Selects specific devices (e.g., memory or I/O devices) for communication.

        - **Example:** When the CPU sends data to memory, the control lines will specify whether it is a read or write operation, as well as which part of the memory is being accessed.

- **Input output buses**
    - The Input-Output bus and interface modules play a crucial role in facilitating communication between the CPU and external devices. The I/O bus connects all the I/O devices to the system, while the interface modules act as translators between the CPU's internal bus and the external devices. Let us see each component in detail



    - **I/O Bus**
        - The I/O bus consists of three primary buses: data bus, address bus, and control bus.
        - The data bus carries the actual data being transferred between the processor and peripherals.
        - The address bus allows the processor to select a specific peripheral device connected to the I/O bus.
        - The control bus provides control signals for managing data transfer, such as read/write, start/stop, and error detection.

- **Concept of input output interface**
    - The Input/Output Interface is a hardware component or system that manages the data transfer between the computer's internal memory or CPU and external devices.
    - These devices can either be input devices (such as a keyboard or mouse) that send data to the computer, or output devices (such as a printer or monitor) that receive data from the computer.
    - The I/O interface also supports serial and parallel communication, converting signals and ensuring compatibility between devices with differing data rates and formats.

- **Functions of Input-Output Interface**
  - The Input-Output Interface serves several critical functions that enable proper communication between the computer system and peripheral devices:
  - **Speed Synchronization**
    - The interface ensures that the CPU's operating speed is synchronized with the input-output devices.
  - **Processor Communication**
    - The interface accepts and decodes commands from the processor, reports the current status, and recognizes its unique address.
  - **Signal Control**
    - It generates and manages control and timing signals needed for data transfer, ensuring smooth communication between the CPU and peripherals.
  - **Data Buffering**
    - The interface enables buffering, which temporarily stores data as it moves between devices and the CPU, helping manage the difference in processing speeds.
  - **Error Detection**
    - The interface can detect errors in data transmission, ensuring that errors are flagged and corrected before they affect system performance.
  - **Data Conversion**
    - It converts serial data to parallel data and vice versa, as well as converting digital data to analog signals and vice versa, and ensures the format is compatible with the receiving device.
  - **Status Reporting**
    - The interface reports the current status of the peripheral device to the processor.
- **Types of Input-Output Interface in Computer Architecture**
  - **Programmed I/O (PIO)**
    - In programmed I/O, the CPU is responsible for controlling the entire process of data transfer. It must wait for the I/O operation to complete before moving on to the next task.

        - **Interrupt-Driven I/O**
            - In interrupt-driven I/O, the CPU does not continuously check for I/O requests. Instead, when an I/O device is ready for communication, it sends an interrupt signal to the CPU. The CPU then temporarily halts its current tasks to process the I/O request.
        - **Direct Memory Access (DMA)**
            - DMA is a method where peripherals can access the system's memory directly, bypassing the CPU for data transfer.
        - **Advanced Programmable Interrupt Controller (APIC)**
            - An APIC manages interrupts by notifying the processor when a device is ready to send data, improving the handling of interrupts in multi-core systems.
        - **Input-Output Memory Management Unit (IOMMU)**
            - It is used in virtualized environments, the IOMMU maps virtual addresses to physical addresses, enabling efficient memory management and isolation between virtual machines.

- **Input Output Processor (IOP)**
    - An Input/Output Processor (IOP) is a specialized processor used in computer systems to manage and facilitate input/output operations between the CPU and peripheral devices (such as hard drives, keyboards, printers, etc.).
    - The primary goal of an IOP is to offload some of the work from the main CPU, allowing the CPU to focus on processing instructions and computations rather than managing I/O operations directly.
    - **Key Functions of an IOP:**
        - **Managing I/O Operations:**
            - The IOP is responsible for managing the communication between the CPU and peripheral devices. It handles data transfer, error checking, and device control for various I/O devices like disks, printers, or communication devices.

        - **Directing Data Flow:**
            - The IOP controls the data flow between devices and the system’s memory, ensuring that data is transferred correctly without overloading the CPU with too many I/O-related tasks.
        - **Interrupt Handling:**
            - By handling interrupts, the IOP helps optimize the system's overall performance and responsiveness.
        - **DMA (Direct Memory Access) Support:**
            - This speeds up data transfers and reduces the burden on the CPU.
    - **IOP vs. CPU:**
        - CPU: The central processing unit executes general-purpose computations, program instructions, and logic operations. It is responsible for the majority of the system's processing tasks.
        - IOP: The input/output processor specializes in handling I/O operations, offloading these tasks from the CPU.
    - **Advantages of an IOP:**
        - **Efficiency:** The CPU can continue executing tasks without being distracted by I/O operations, improving system performance.
        - **Faster Data Transfer:** With features like DMA, the IOP can handle direct memory-to-device or device-to-memory transfers, which are typically faster than going through the CPU.
        - **Reduced CPU Overhead:** By handling interrupts and I/O tasks independently, the IOP reduces the load on the CPU, allowing it to perform other critical computations without delay.

- ❖ **Direct memory access**
    - Direct Memory Access (DMA) is a method used in computer systems that allows peripheral devices (like hard drives, sound cards, or network interfaces) to access the system's memory directly without involving the CPU. This method speeds up data transfer and reduces the workload on the CPU, enabling more efficient processing of data-intensive tasks.
    - Key Concepts of DMA:
    - **Purpose of DMA:**
        - The main goal of DMA is to allow peripheral devices to transfer data to or from memory directly, bypassing the need for the CPU to manage the transfer. This helps in offloading I/O

operations from the CPU, allowing it to focus on other tasks and improving the overall performance of the system.

- **How DMA Works:**
- **DMA Controller (DMAC):**
    - A dedicated hardware component, called the DMA Controller (DMAC), is responsible for managing DMA operations. The DMAC takes care of controlling the data transfer between the device and memory.
- **DMA Channels:**
    - Many DMA controllers support multiple DMA channels. Each channel can be used to manage a different device, allowing multiple devices to transfer data simultaneously.
- **Memory and Device Access:**
    - The DMA controller allows the peripheral device to directly read from or write to system memory, without the need for the CPU to process each data transfer. The DMAC controls the flow of data by setting the source and destination addresses in memory and triggering the transfer.
- **DMA Process:**
- **Step 1: Device Request:** A peripheral device requests access to memory via the DMA controller. For example, a disk drive might request DMA access to write data to memory.
- **Step 2: CPU Grant:** Once the request is received, the CPU may grant permission for the DMA transfer to begin, or it may simply enable the DMA controller to initiate the transfer at the right moment (depending on the system's design).
- **Step 3: Data Transfer:** Once DMA is activated, the peripheral device can transfer data directly to memory (or vice versa) under the control of the DMA controller, without involving the CPU.
- **Step 4: Interrupt and Completion:** Once the data transfer is completed, the DMA controller typically triggers an interrupt to notify the CPU that the operation has finished. The CPU may then process the data or perform other actions.

- **Benefits of DMA:**
- **Reduced CPU Overhead:**
    - By transferring data directly between memory and peripherals, DMA allows the CPU to focus on other tasks instead of managing each byte or word of data transfer.

- **Faster Data Transfers:**
    - DMA enables high-speed data transfer between memory and peripheral devices, often significantly faster than relying on the CPU to handle the transfer.
- **Efficient I/O Operations:**
    - DMA optimizes data-intensive operations like reading from a disk or transferring large data sets by providing an efficient, high-speed channel for I/O operations.

- **DMA controller**
    - A DMA (Direct Memory Access) controller is a hardware device that facilitates data transfer between I/O devices and main memory without requiring constant CPU intervention, thereby improving system performance and reducing CPU workload.
    - **How it works:**
        - **DMA Request:**
            - An I/O device initiates a DMA request, signaling the DMA controller that it needs to transfer data.
        - **CPU Notification:**
            - The DMA controller informs the CPU that a DMA transfer is about to occur.
        - **Bus Control:**
            - The DMA controller takes control of the system buses (address, data, and control buses) to initiate the data transfer.
        - **Data Transfer:**
            - The DMA controller transfers the data between the I/O device and memory without CPU involvement.
        - **Completion:**
            - Once the transfer is complete, the DMA controller signals the CPU, indicating that the operation is finished.