Digital Principles

What Is a Digital Signal?

A digital signal refers to a type of signal that represents data as a sequence of discrete values. Unlike analog signals, which are continuous and can represent a vast range of values, digital signals take on only specific values. For instance, in digital electronics, these values are often represented as **'0s' and '1s'**. This binary method is fundamental to computers, smartphones, and many other electronic devices you use daily.

A signal in which the original information is converted into a string of bits before being transmitted. A radio signal, for example, will be either on or off. Digital signals can be sent for long distances and suffer less interference than analog signals.

The communications industry worldwide is in the midst of a switch to digital signals.

Sound and video can also be streamed via computer.

Sound storage in a compact disk is in digital form.

How Does a Digital Signal Work?

Digital signals function through the process of digital modulation. This can be understood by considering how a digital watch displays time with numbers changing at a fixed rate. Here are some points to explain further:

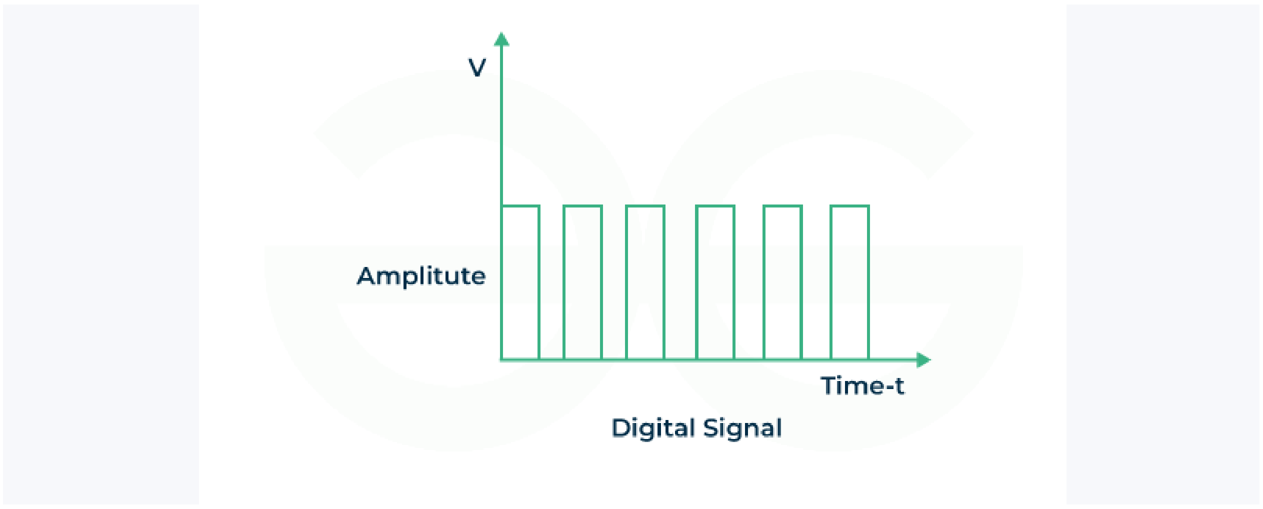
- **Binary Codes:** Digital signals use binary codes (0s and 1s) to represent information, which makes them less susceptible to interference and noise compared to analog signals.
- **Transmission:** They are transmitted via a series of pulses, and each pulse represents a specific value.

Why Are Digital Signals Important?

Digital signals are crucial for modern technology. Here's why:

- **Clarity:** They maintain clarity over long distances, which is why your phone calls and TV pictures are clear, even if transmitted across long distances.
- **Storage:** Digital format is easier to store and less prone to degradation over time compared to analog signals.
- **Processing:** Digital data can be processed and manipulated more easily with computers, supporting a wide range of applications from simple calculations to complex simulations.

Understanding digital signals helps us appreciate how data is handled and transmitted in various electronic devices and communication systems.



Digital logic

What is Digital Logic?

Modern computing system consists of complex system and technologies. These technologies are built upon some fundamental simple logics known as digital logic. By using digital logic gates we can develop complex logical circuit for various purposes like data storing, data manipulation or simply data representation.

What is Digital Logic?

One of the most important branch of Electronic and telecommunication Science sector is Digital electronics (logic). Digital logic is mainly used for data(must be digital information) representation, manipulation and processing of using discrete signals or binary digits (bits). It can perform logical operations, data retrieval or storing and data transformation by analyzing logical circuit design.

What is Digital?

Previously a continuous signal or values are used represent data which is known as Analog signal. In modern computing sectors, data representation changes to discrete/non-continuous signals or values (only 0 or 1) which are known as Digital. Here, the overall information is encoded in a sequence of bits where every bits represents only two states(1 for high and 0 for low) of the information. This is known as binary representation of information.

Why Digital Logic is Necessary?

In modern computing realm, Digital logic plays a significant role in many sectors which are discussed below:

Universal Representation: For any type of data representation like image, text, video, audio etc. digital logic/system is used by encoding the data in binary form. This binary formatted data enables uniform handling of diverse data and allows seamless integration and compatibility.

Error Reduction and Correction: Digital logic itself is very less prone to error as it works with only two values(0 and 1). Moreover, we can employ redundancy check and error detection mechanisms by digital logic codes which can detect and rectify errors introduced during transmission. This ensures reliable and accurate data processing.

Scalability and Modularity: Digital logic provides scalable framework by which we can develop complex system by using basic logic gates only. This enables a easy and cost effective way to develop a large-scale system with improved flexibility, maintainability, and ease of integration.

Noise Immunity: As digital logic follows the discrete nature of signal so it is less prone to have induced noise compared to analog signal. So it provides more robust communication and data processing by noise filtering and error mitigation.

Digital Logic refers to the system of rules and processes that electronic devices use to perform operations based on binary numbers (0s and 1s). It is the foundation of all digital circuits, including computers, calculators, and microcontrollers.

Easy Explanation:

Think of digital logic as a set of simple rules that tell a device how to process information using only two states:

0 (OFF or LOW voltage)

1 (ON or HIGH voltage)

These rules are implemented using logic gates, which are small circuits that take one or more inputs and produce an output based on logical operations

Example:

A light switch can be thought of as a basic digital logic system:

Switch OFF (0) → Light is OFF

Switch ON (1) → Light is ON

In digital electronics, we use logic gates like AND, OR, and NOT to process inputs and make decisions.

For example, an AND Gate works like this:

If both inputs are 1 (ON) → Output is 1 (ON)

If any input is 0 (OFF) → Output is 0 (OFF)

AND Gate Example:

Imagine a security system that only opens a door if both the fingerprint and the password are correct (both are 1). If one of them is wrong (0), the door remains locked.

Boolean laws and theorems

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)'

Boolean algebra is a branch of mathematics that deals with binary values (0 and 1) and logical operations. It is widely used in digital circuits and computer science.

Basic Boolean Laws and Theorems with Examples

1. **Identity Law-**
   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 (0 is the identity for OR)

A · 1 = A (1 is the identity for AND)

 Example:

If A = 1, then

1 + 0 = 1

1 · 1 = 1

If A = 0, then

0 + 0 = 0

0 · 1 = 0

2. Null Law (Dominance Law)

A + 1 = 1

A · 0 = 0

 Example:

If A = 0, then

0 + 1 = 1

0 · 0 = 0

If A = 1, then

1 + 1 = 1

1 · 0 = 0

3. Idempotent Law

A + A = A

$A \cdot A = A$

□ Example:

If $A = 1$, then

$1 + 1 = 1$

$1 \cdot 1 = 1$

If $A = 0$, then

$0 + 0 = 0$

$0 \cdot 0 = 0$

4. Complement Law

$A + A' = 1$ (A' means NOT A)

$A \cdot A' = 0$

□ Example:

If $A = 1$, then $A' = 0$

$1 + 0 = 1$

$1 \cdot 0 = 0$

If $A = 0$, then $A' = 1$

$0 + 1 = 1$

$0 \cdot 1 = 0$

5. Double Negation Law or Inversion Law-

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.

$(A')' = A$

□ **Example:**

If A = 1, then

(1')' = 1

If A = 0, then

(0')' = 0

6. Commutative Law-

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

□ **Example:**

If A = 1, B = 0, then

1 + 0 = 0 + 1 = 1

1 · 0 = 0 · 1 = 0

7. Associative Law-

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)

□ **Example:**

If A = 1, B = 0, C = 1, then

(1 + 0) + 1 = 1 + (0 + 1) = 1

(1 · 0) · 1 = 1 · (0 · 1) = 0

8. Distributive Law-

Boolean Variables also follow the distributive law and the expression for Distributive law is given as:

$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

$A + (B \cdot C) = (A + B) \cdot (A + C)$

□ **Example:**

If $A = 1, B = 0, C = 1$, **then**

$1 \cdot (0 + 1) = (1 \cdot 0) + (1 \cdot 1) = 0 + 1 = 1$

$1 + (0 \cdot 1) = (1 + 0) \cdot (1 + 1) = 1 \cdot 1 = 1$

9. Absorption Law

$A + (A \cdot B) = A$

$A \cdot (A + B) = A$

□ **Example:**

If $A = 1, B = 0$, **then**

$1 + (1 \cdot 0) = 1 + 0 = 1$

$1 \cdot (1 + 0) = 1 \cdot 1 = 1$

10. De Morgan’s Theorems

De Morgan’s Laws are also called De morgan’s Theorem. They are the most important laws in Boolean Algebra and these are added below under the heading Boolean Algebra Theorem

Boolean Algebra Theorems

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. Now let’s learn about both in detail.

1.De Morgan’s First laws

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**

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.

De Morgan’s Second laws

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)’

Proof:

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**

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.

(A · B)' = A' + B'

(A + B)' = A' · B'

□ **Example:**

If A = 1, B = 0, then

(1 · 0)' = 1' + 0' = 0 + 1 = 1

(1 + 0)' = 1' · 0' = 0 · 1 = 0

Introduction of K-Map (Karnaugh Map)

In numerous digital circuits and other practical problems, finding expressions that have minimum variables becomes a prerequisite. In such cases, minimisation of Boolean expressions is possible that have 3, 4 variables. It can be done using the Karnaugh 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.

A Karnaugh Map (K-Map) is a simple way to simplify Boolean algebra expressions. It helps reduce logic circuits, making them more efficient.

How It Works:

Grid Representation: A K-Map is a grid that represents all possible values of a Boolean function.

Placing Values: The values in the grid come from a truth table (0s and 1s).

Grouping 1s: You group adjacent 1s in powers of 2 (1, 2, 4, 8, etc.).

Simplifying Expression: Each group represents a simplified Boolean expression.

Solving an Expression Using K-Map

Here are the steps that are used to solve an expression using the K-map method:

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.

Working Principle of K-Map

We know that there are mainly two forms in which logical expressions can be represented namely:

- **Sum-of-products form (SOP)**
- **Product-of-sums form (POS)**

Advantages of K-Map

- **Simplifies Boolean expressions** – Reduces complex logic circuits easily.
- **Visual Representation** – Makes it easier to find patterns compared to algebraic simplification.
- **Faster than Boolean algebra** – Reduces manual calculation efforts.
- **Reduces logic gates** – Leads to cost-effective and efficient digital circuits.

Disadvantages of K-Map

- **Limited to small variables** – Becomes difficult for more than 5-6 variables.
- **Manual Errors Possible** – Mistakes in grouping may lead to incorrect expressions.
- **Not Practical for Computer-Based Design**

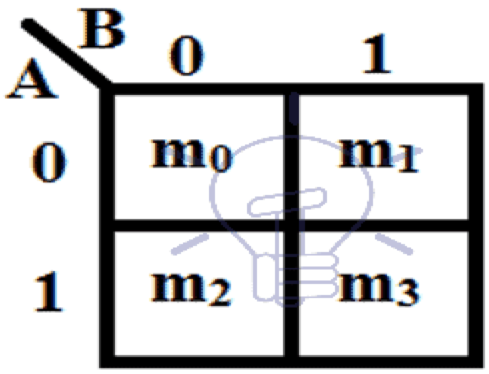
Truth Tables for Karnaugh Map (K-Map)

A **truth table** represents all possible input combinations and their corresponding output values in a Boolean function. The **Karnaugh Map (K-Map)** organizes these values into a structured grid, making it easier to simplify the function.

2 Variable K-Map

2 variables have $\mathbf{2^n = 2^2 = 4}$ minterms. Therefore there are 4 cells (squares) in 2 variable K-map for each minterm.

Consider variable A & B as two variables. The rows of the columns will be represented by variable B. The square facing the combination of the variable represents that min term as shown in fig below.



Grouping in 2 variables K-map is easy as there are few squares.

1. Truth Table for 2-Variable K-Map

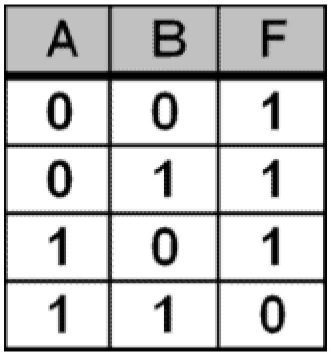
For a function **F(A, B):**

A	B	F(A,B)
0	0	X
0	1	X
1	0	X
1	1	X

2-Variable K-Map Layout:

A \ B	0	1
0	X	X
1	X	X

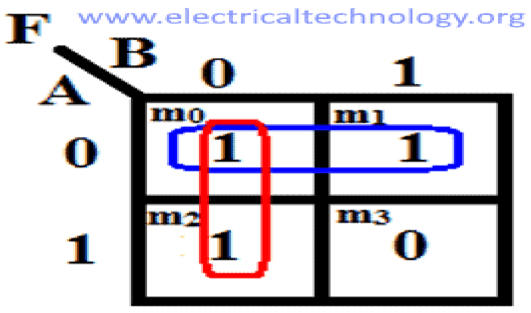
Example of 2 Variable K-Map
Function F (A, B)



A	B	F
0	0	1
0	1	1
1	0	1
1	1	0

F = $\sum (m_0, m_1, m_2) = \bar{A}\bar{B} + \bar{A}B + A\bar{B}$

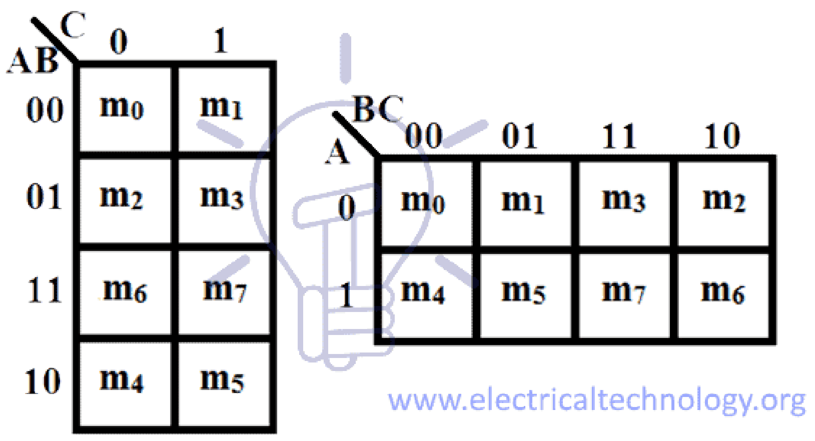
K-map from Truth table



- We made 2 groups of 1's. each group contains 2 minterms.
- In the first group, variable A is changing & B remains unchanged. So the first term of the output expression will be $\bar{B}$ (because **B = 0** in this group).
- In the 2nd group, Variable B is changing and variable A remains unchanged. So the second term will be of the output expression will be $\bar{A}$ (because A=0 in this group).
- Now the simplifies expression will be the sum of these two terms as given below,

3 Variable K-Map

3 variables make **$2^n=2^3=8$** min terms, so the Karnaugh map of 3 variables will have 8 squares(cells) as shown in the figure given below.

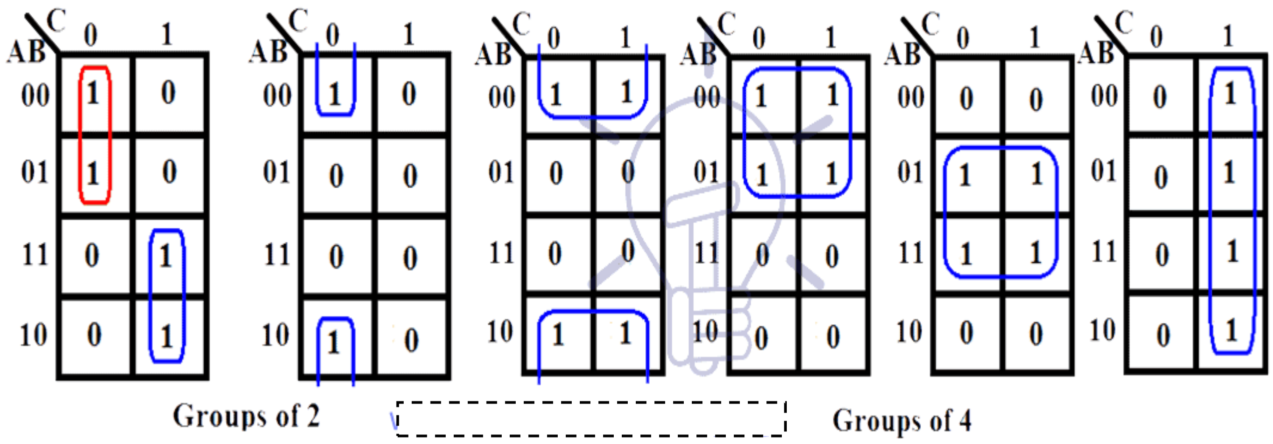


3 variable K-map can be in both forms. Note the combination of two variables in either form is written in Gray code. So the min terms will not be in a decimal order.

The uppermost & lowermost cells are adjacent in the first form of K-map, the leftmost and rightmost cells are also adjacent in the second form of K-map. So they can be made into groups.

Some examples of grouping:

You can make groups of 2, 4 & 8 cells having same 1s or 0s.



Notice the groups of the uppermost & lowermost cells. They are adjacent as there is only one-bit difference. That is why they can be grouped together. Don't make unnecessary groups. All 1s or 0s should be grouped, not all possible groups of 1s or 0s should be made.

Truth Table for 3-Variable K-Map

For a function **F(A, B, C):**

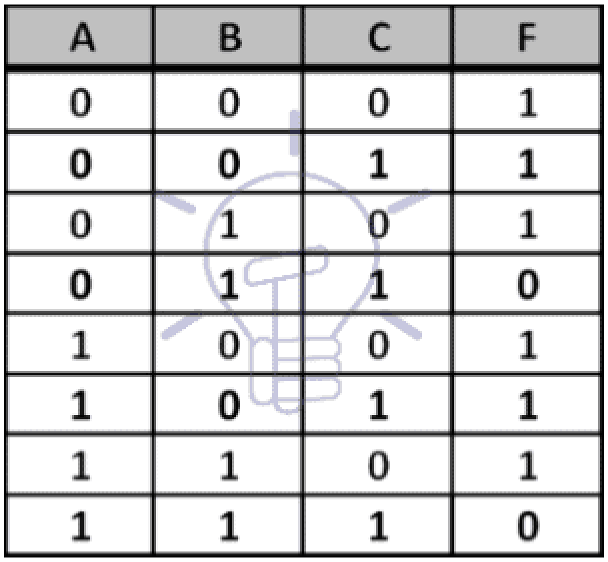
A	B	C	F(A,B,C)
0	0	0	X
0	0	1	X
0	1	0	X
0	1	1	X
1	0	0	X
1	0	1	X
1	1	0	X
1	1	1	X

3-Variable K-Map Layout:

AB \ C	**0**	**1**
00	X	X
01	X	X
11	X	X
10	X	X

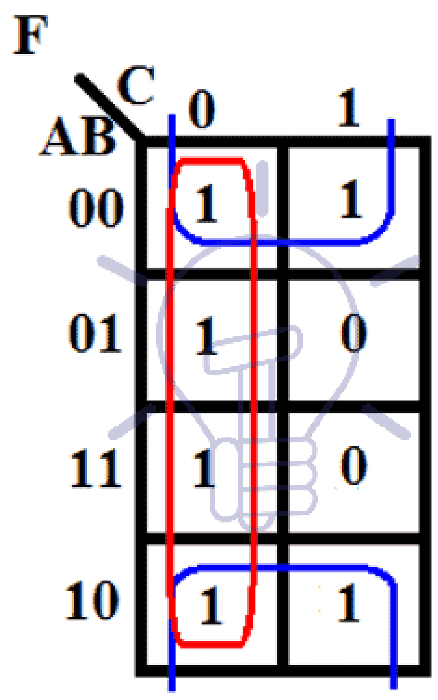
Example of 3 Variable K-Map

$$F(A,B,C) = \sum(m_0, m_1, m_2, m_4, m_5, m_6)$$

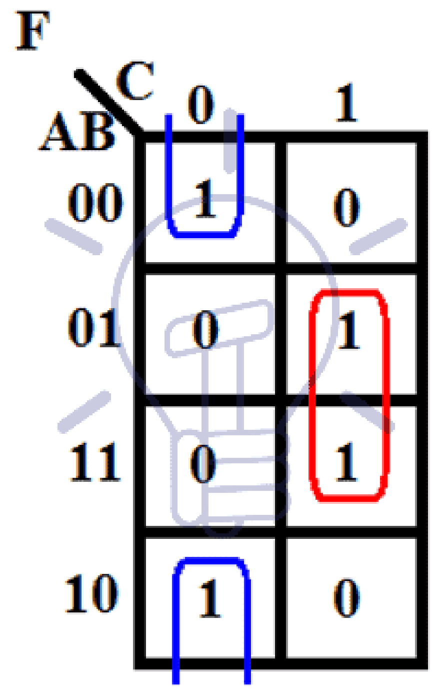


A	B	C	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

This example shows that you can make the groups overlap each other to make them as large as possible and cover all the 1s.



In this first group **(m_0, m_2, m_6, m_4)**, A &B are changing so we will eliminate it. However, C remains unchanged in this group. So the term this group produce will be $\overline{C}$ (because C=0 in this group).



In the 2nd group (m_0,m_1,m_4,m_5), A and C are changing so it will be eliminated from the term. However, B remains unchanged in this group. So the term this group produce will be $\overline{B}$ (because B=0 in this group).

The sum of these two terms will make the simplified expression of the function as given below.

$$\mathbf{F = \overline{B} + \overline{C}}$$

Another example of grouping of 2 is given below. It shows how the corner min terms are grouped.
In the **first group** (m_0,m_4), A is changing. B & C remains unchanged. So the term will be $\overline{B}\overline{C}$ (B=0,C=0 in this group).

In 2nd group (m3,m7), A is changing. B & C remains unchanged. BC will be the term because B=1,C=1 in this group.

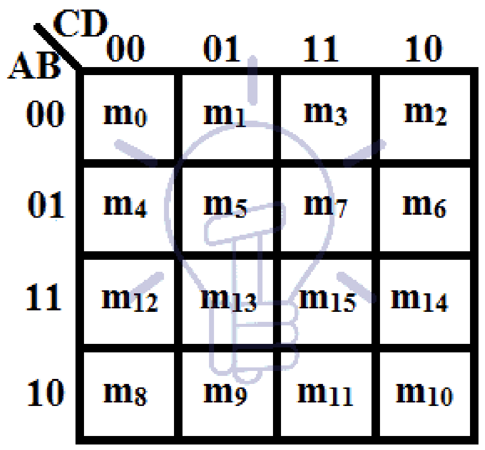
So This K-map leads to the expression

$F = \overline{B}\overline{C} + BC$

These two examples show that a group of 4 cells give a term of 1 literal and a group of 2 cells gives a term of 2 literals and a group of 1 cell gives a term of 3 literals. So the larger the group,the smaller and simple the term gets.

4-variable K-Map

4 variables have $\mathbf{2^n=2^4=16}$ minterms. So a 4-variable k-map will have 16 cells as shown in the figure given below.



Each cell (min term) represent the variables in front of the corresponding row & column.

Truth Table for 4-Variable K-Map

For a function **F (A, B, C, D):**

A	B	C	D	F(A,B,C,D)

A	B	C	D	F(A,B,C,D)
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	X
0	1	0	0	X
0	1	0	1	X
0	1	1	0	X
0	1	1	1	X
1	0	0	0	X
1	0	0	1	X
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

4-Variable K-Map Layout:

AB \ CD	00	01	11	10
00	X	X	X	X
01	X	X	X	X
11	X	X	X	X
10	X	X	X	X

How to Use a Truth Table in K-Map?

1. **Write the truth table** with all possible input values.
2. **Transfer 1s (or 0s for POS) to the K-Map** in their respective positions.
3. **Group adjacent 1s in powers of 2** (1, 2, 4, 8, etc.).
4. **Derive a simplified Boolean expression** from the groups.

K-Map Simplification

Karnaugh Map (K-Map) simplification is a method used to minimize Boolean expressions, reducing the number of logic gates in a circuit. It works by grouping **1s** (or **0s for POS**) in the K-Map and forming a simplified Boolean equation.

Steps for K-Map Simplification

1. **Create a Truth Table** – Identify the Boolean function values for all input combinations.
2. **Fill the K-Map** – Transfer the **1s (for SOP) or 0s (for POS)** from the truth table into the K-Map.
3. **Group the 1s (or 0s)** – Combine adjacent **1s** in powers of 2 (1, 2, 4, 8, etc.).
4. **Write the Simplified Expression** – Derive the Boolean equation based on the groups.

Example: 2-Variable K-Map Simplification

Consider the function:
F(A, B) = Σ(1, 3) → (minterms 1 and 3 are 1s in the K-Map)

Step 1: Truth Table

A	B	F(A, B)
0	0	0
0	1	1
1	0	0
1	1	1

Step 2: K-Map Representation

A \ B	0	1
0	0	1
1	0	1

Step 3: Grouping

- The two **1s** (minterms 1 and 3) are in the **same column**.
- Since **B = 1** in both cases, the simplified function is:
  F(A, B) = B

Example: 3-Variable K-Map Simplification

Consider **F(A, B, C) = Σ(1, 3, 5, 7)**.

Step 1: Truth Table

A	B	C	F(A, B, C)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Step 2: K-Map Representation

AB \ C	0	1
00	0	1
01	0	1
11	0	1
10	0	1

Step 3: Grouping

- The **four 1s** are all in **column C = 1**.
- Since all **A, B** combinations have C = 1, the simplified function is:
  F(A, B, C) = C

Example: 4-Variable K-Map Simplification

For **F(A, B, C, D) = Σ(0, 2, 8, 10, 5, 7, 13, 15)**.

Step 1: Truth Table

A	**B**	**C**	**D**	**F(A, B, C, D)**
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Step 2: K-Map Representation

AB \ CD	**00**	**01**	**11**	**10**
00	1	0	0	1

AB \ CD	00	01	11	10
01	0	1	1	0
11	0	1	1	0
10	1	0	0	1

Step 3: Grouping

- Four **1s** form a large group in **CD = 00 or 10**.
- Another group is in **CD = 01 or 11**.
- The simplified function:
  F(A, B, C, D) = A'C' + BD

Conclusion

- **2-variable K-Maps** simplify to a single variable term.
- **3-variable K-Maps** can simplify to expressions like A, B, or C.
- **4-variable K-Maps** form more complex groups but still result in simpler expressions.

Don't Care Condition

A **don't care condition** refers to a scenario in which the output of a function is **not relevant** for certain input combinations. This means we can assign either **0 or 1** to these conditions to **simplify Boolean expressions or logic circuits**.

Why Do We Use Don't Care Conditions?

1. **Unused Input Combinations:** Some input combinations may never occur in a system, so we don't care about their output.
2. **Simplification of Boolean Expressions:** It helps in reducing the complexity of logic circuits.
3. **Efficient Circuit Design:** It allows us to minimize the number of gates used.

Example 1: Digital Circuit Design (Truth Table)

Consider a **3-bit binary code** (000 to 111). If we are designing a **BCD (Binary-Coded Decimal)** system, we only use **0000 to 1001 (0 to 9)**, while the remaining inputs **(1010 to 1111)** are invalid.

A	B	C	Output (F)
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

Here, **X represents don't care conditions**, meaning we can assign **either 0 or 1** to simplify the logic circuit.

Example 2: Karnaugh Map (K-Map) Simplification

Suppose we have the function:

F(A, B, C) = Σ(0, 1, 4, 5) **(minterms where F = 1)**
Don't Care Conditions: d(A, B, C) = Σ(6, 7) **(minterms that we don’t care about)**

By treating **don't care conditions as either 0 or 1**; we can group them to form larger simplifications in K-Map.

Key Takeaways

- Don't care conditions help **reduce circuit complexity**.
- They are represented as **X** in truth tables and K-Maps.
- They can be assigned **either 0 or 1** to achieve the simplest expression.
- Commonly used in **BCD systems, memory design, and logic circuit simplification**.

Advantages of Using Don't Care Conditions

1. simplifies Boolean expressions.
2. Reduces the number of logic gates.
3. Minimizes hardware cost.
4. Enhances performance by reducing delay.

Real-Life Applications of Don't Care Conditions

BCD to 7-Segment Display: Certain numbers (10–15 in BCD) are unused, so they are marked as **don't care**.

Memory Addressing: Some addresses are never used, and their values can be **don't care** to optimize circuits.

State Machines (FSMs): Unreachable states in a state diagram are treated as **don’t care** to minimize logic design.

Multiplexers & Decoders: Some select lines might **never be active**, so their input values can be **don't care**.

Points-

- Don’t care conditions help simplify logic circuits**.**
- Represented as **X** in truth tables and K-Maps**.**
- Used in BCD systems, memory addressing, FSMs, and multiplexers**.**
- They allow flexible assignment (0 or 1) for better optimization**.**
- The major difference between SOP and POS is that the SOP represents a Boolean expression through minterms, while POS defines a Boolean expression through max terms.

- ### What is SOP?

- SOP stands for Sum of Product. SOP form is a set of product(AND) terms that are summed(OR) together. When an expression or term is represented in a sum of binary terms known as minterms and sum of products.

Definition of SOP

When we add two or multiple product terms by a boolean addition, the output expression is a **sum-of-products (SOP)**. For example, the expression a’bc’ + a’bd’ + a’bc’d shows a SOP expression. It can also have a single variable term within the expression like a + bc +a’b. These logical expressions are simplified in a way that they must not contain redundant information while creating the minimal version of it.

Domain of a boolean expression

The group of variables, either complimented or uncomplimented, comprised in a boolean expression, is known as the **domain**. Let’s suppose, we have an expression a’b + ab’c then the domain of this expression would be the set of the variables a, b, c.

Implementation of the SOP form

It is mainly implemented by an **AND-OR** logic where the product of the variables are first produced by AND gate and then added by the OR gates. For example, the expression “**ab’+bcd+ac**” can be expressed by the logic circuit shown in **figure 1.1** where the output P of the OR gate is the SOP expression.

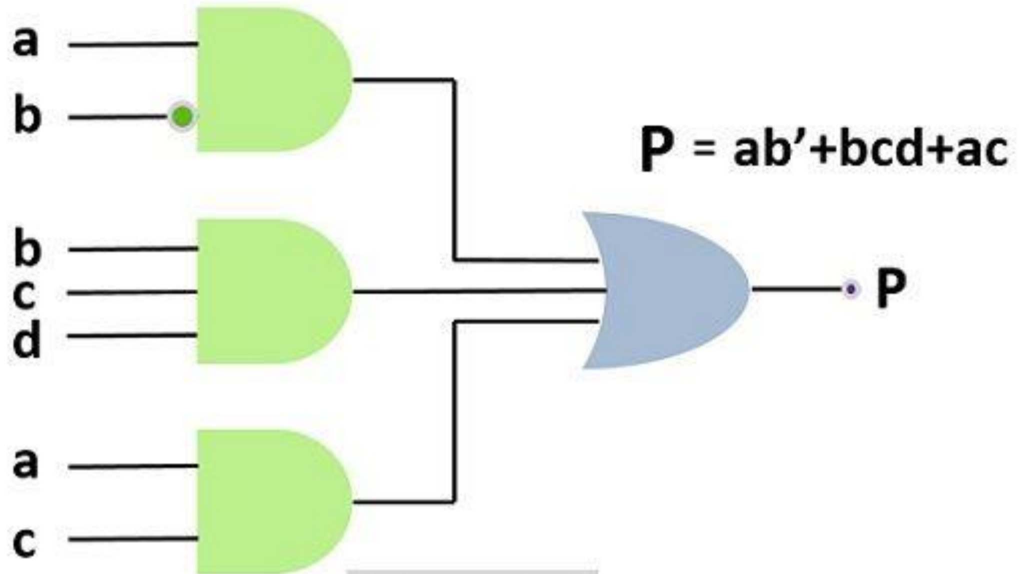


Figure 1.1

Steps for converting the product term into standard SOP

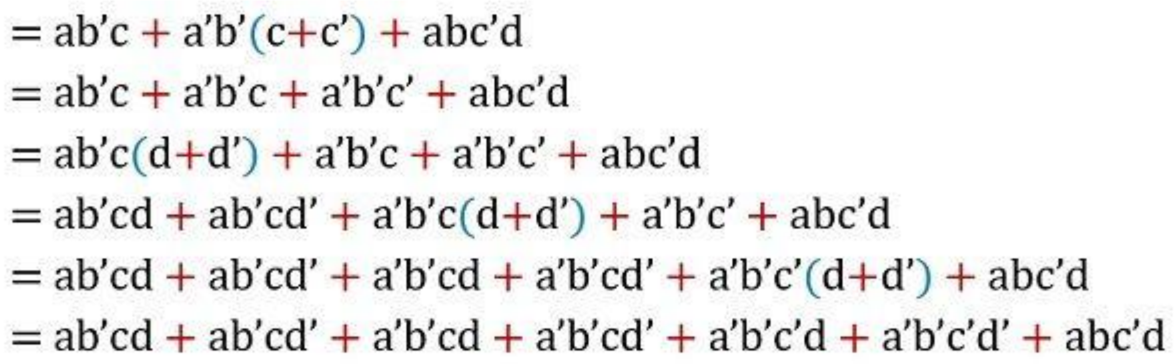
Here the standard SOP or canonical SOP refers to an expression in which all the variables of the domain are present. For generating, standard SOP from the product term the boolean rule “**A+A’=1**” (the output is ‘1’ when a variable added to its complement) is used and below given steps are followed.

- Each non-standard term is multiplied by a term constructed by the addition of the absent variable and its complement. This produces two product terms, as we know that anything can be multiplied by ‘1’ without changing its value.
- Repeat the step ‘1’ until all the domain variables are present in the expression R.

Example

The term **ab’c+a’b’+abc’d** converted into the standard SOP or canonical SOP by multiplying the part of the term by the missing term. Such as **a’b’** is multiplied with the **c+c’**.

Similarly, the whole expression is converted in its canonical form by the following given steps.

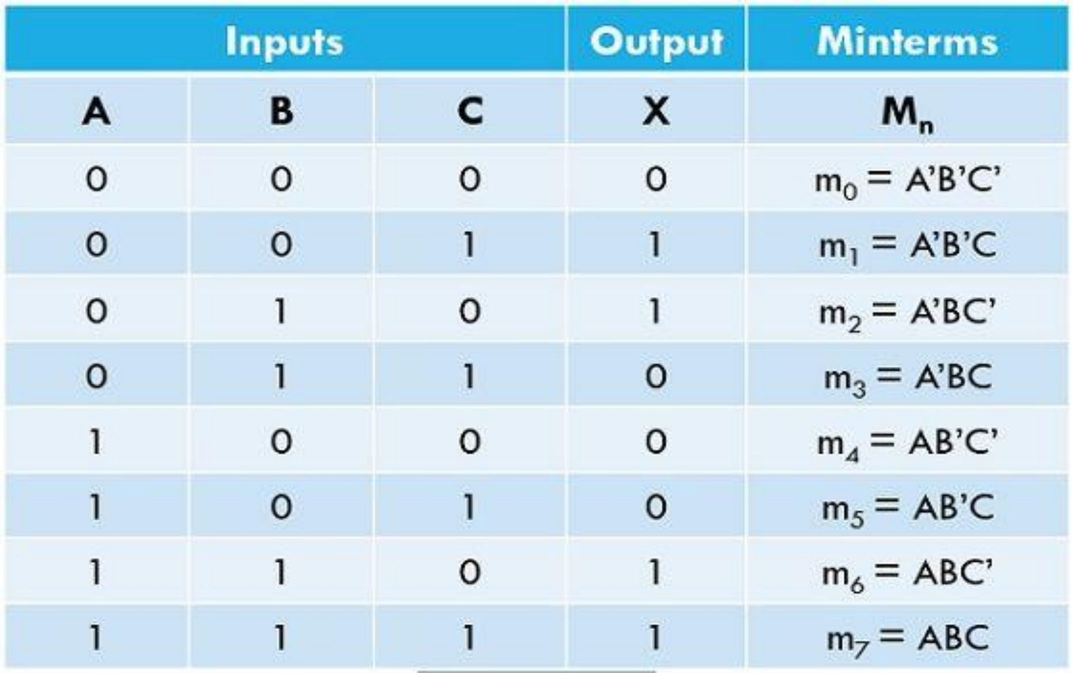


$$= ab'c + a'b'(c+c') + abc'd$$
$$= ab'c + a'b'c + a'b'c' + abc'd$$
$$= ab'c(d+d') + a'b'c + a'b'c' + abc'd$$
$$= ab'cd + ab'cd' + a'b'c(d+d') + a'b'c' + abc'd$$
$$= ab'cd + ab'cd' + a'b'cd + a'b'cd' + a'b'c'(d+d') + abc'd$$
$$= ab'cd + ab'cd' + a'b'cd + a'b'cd' + a'b'c'd + a'b'c'd' + abc'd$$

Now there are various terms which are used while generating the reduced logic function such as minterm, maxterm, k-map (Karnaugh Map), which we will elucidate further in the article.

Minterm

In these terms, the input variables making up a boolean expression is the dot product of each other, and it is also known as **minterm** or **product term**. M_n following table presents the minterms of the variables.

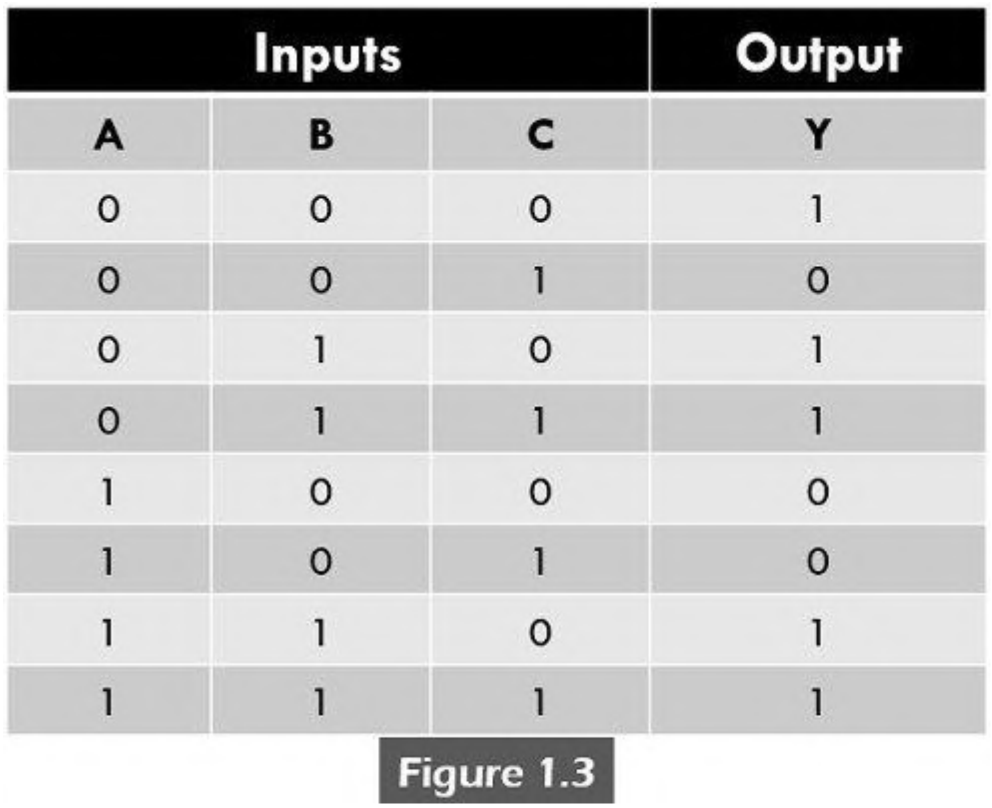


Inputs			Output	Minterms
A	B	C	X	M_n
0	0	0	0	m_0 = A'B'C'
0	0	1	1	m_1 = A'B'C
0	1	0	1	m_2 = A'BC'
0	1	1	0	m_3 = A'BC
1	0	0	0	m_4 = AB'C'
1	0	1	0	m_5 = AB'C
1	1	0	1	m_6 = ABC'
1	1	1	1	m_7 = ABC

Figure 1.2

SOP expression from a Truth table

Suppose, we have a truth table (as shown in figure 1.3) in which each term of the input variables are written as the product of all the terms. To determine the input combinations that exist, we need to select the output having value 1 and convert the binary into relevant product term. Here, we will consider '0' as the variable and '1' as the compliment of the variable.



Inputs			Output
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 1.3

Y(A,B,C,D) = ∑m (0,2,3,6,7)
Only those product terms are selected where the output value is 1.
$Y = m_0+m_2+m_3+m_6+m_7$
So, in accordance with the truth table, the boolean function **(Canonical SOP form)** in the minterm is :
Y = A'B'C'+A'BC'+A'BC+ABC'+ABC
0 is variable and 1 is the complement of the variable.

What is POS?

- POS stands for product of sum. A technique of explaining a Boolean expression through a set of max terms or sum terms, is known as POS(product of sum).

Definition of POS

POS (Product of Sums) is the representation of the boolean function in which the variables are first summed, and then the boolean product is applied in the sum terms. For example, (a'+b).(a+b'+c) is POS expression where we can see that the variables are added then each bigger term is the product of the other.

Implementation of the POS form

It just needs the variables to be inserted as the inputs to the OR gate. The terms generated by the OR gates are inserted in the AND gate. The sum term is formed by an OR operation, and product of two or multiple sum terms is created by an AND operation. To understand the POS implementation refer the below given **figure 2.1** of the expression (a+b).(b+c+d).(a+c).

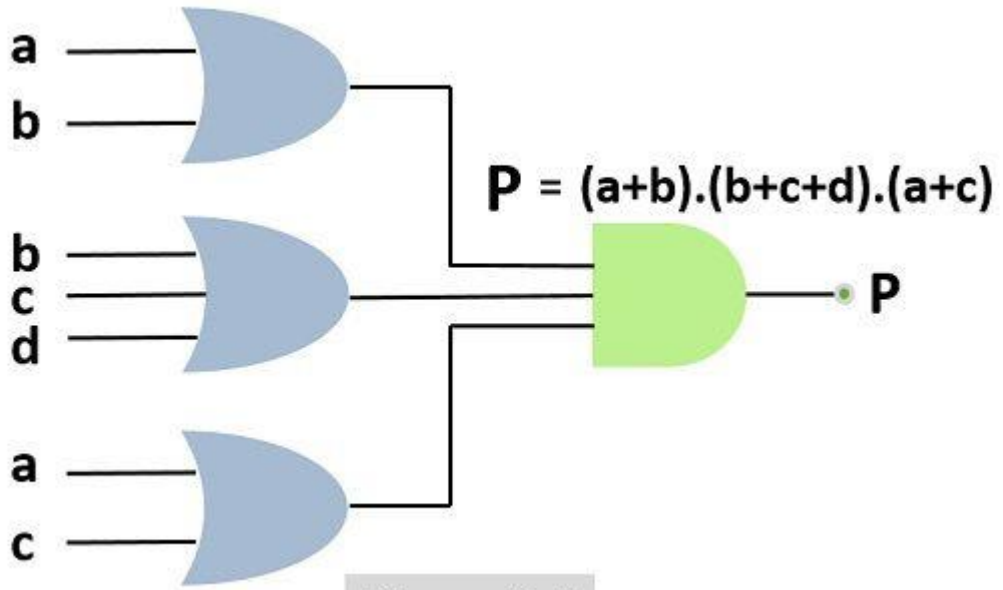


Figure 2.1

Steps for converting the product term into standard POS

Similar to the previous explanation, the standard or canonical POS is in which sum terms does not include all of the variables in the domain of the expression. Here, also we use the boolean algebra rule 8 "**A.A'=0**" (a variable multiplied by its complement is 0) to convert a term in a standard form, and the method is given below.

- In the very first step, each non-standard term added with a term comprised of the product of the absent variable and its complement. This will produce two sum term, and '0' can be added to any term without changing its value.
- Then, rule 12 (i.e.**A+BC=(A+B)(A+C)**) is applied to the terms.
- The first two steps are redone again and again until all the sum terms involve all the variables present in the domain either in the complemented and uncomplemented form.

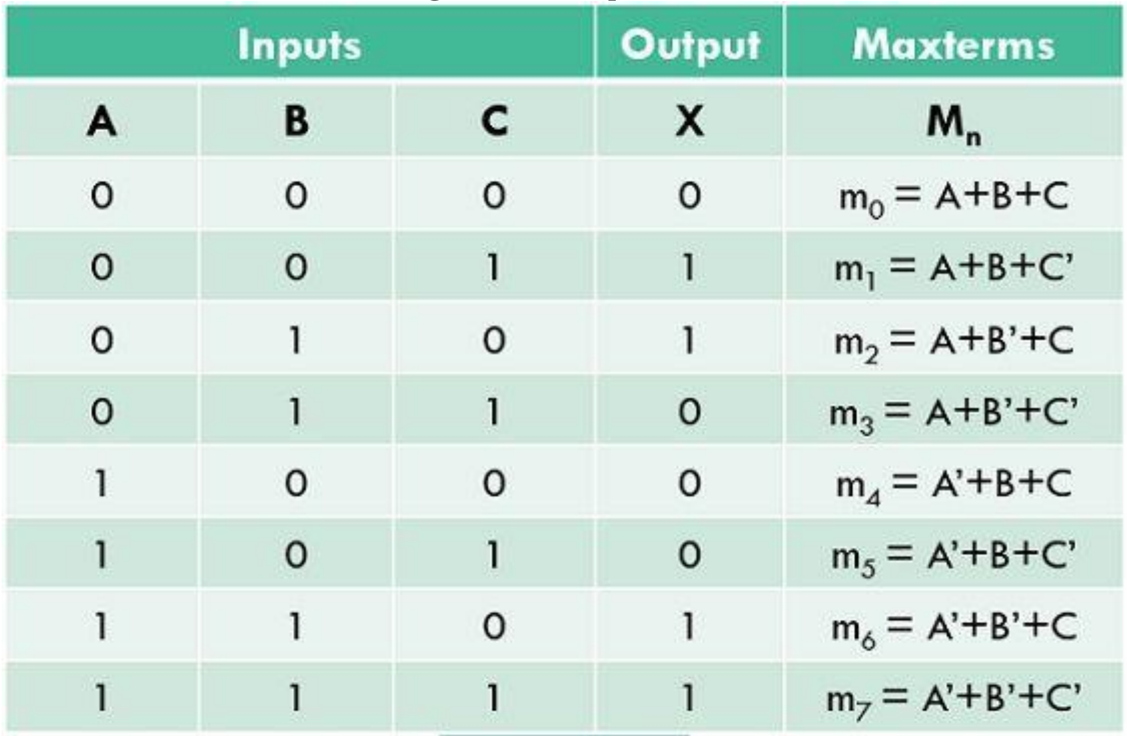
Example

The term (a+b’+c)(b’+c+d’)(a+b’+c’+d) is translated into the standard POS or canonical POS by adding each term with the missing term (which is a product of its complement). Such as “a+b’+c” is summed with the d.d’. In this way, the entire expression is converted in its canonical form by the following given steps.

$$= (a+b'+c).(b'+c+d').(a+b'+c'+d)$$
$$= (a+b'+c+d.d').(b'+c+d').(a+b'+c'+d)$$
$$= (a+b'+c+d).(a+b'+c+d').(b'+c+d'+a.a').(a+b'+c'+d)$$
$$= (a+b'+c+d).(a+b'+c+d').(b'+c+d'+a).(b'+c+d'+a').(a+b'+c'+d)$$
$$= (a+b'+c+d).(a+b'+c+d').(a'+b'+c+d').(a+b'+c'+d)$$

Maxterm

These are the terms in which the input variables are present in summation form, alternately called as **sum term**. The below-given table represents the M_n, **maxterms** of the variables.

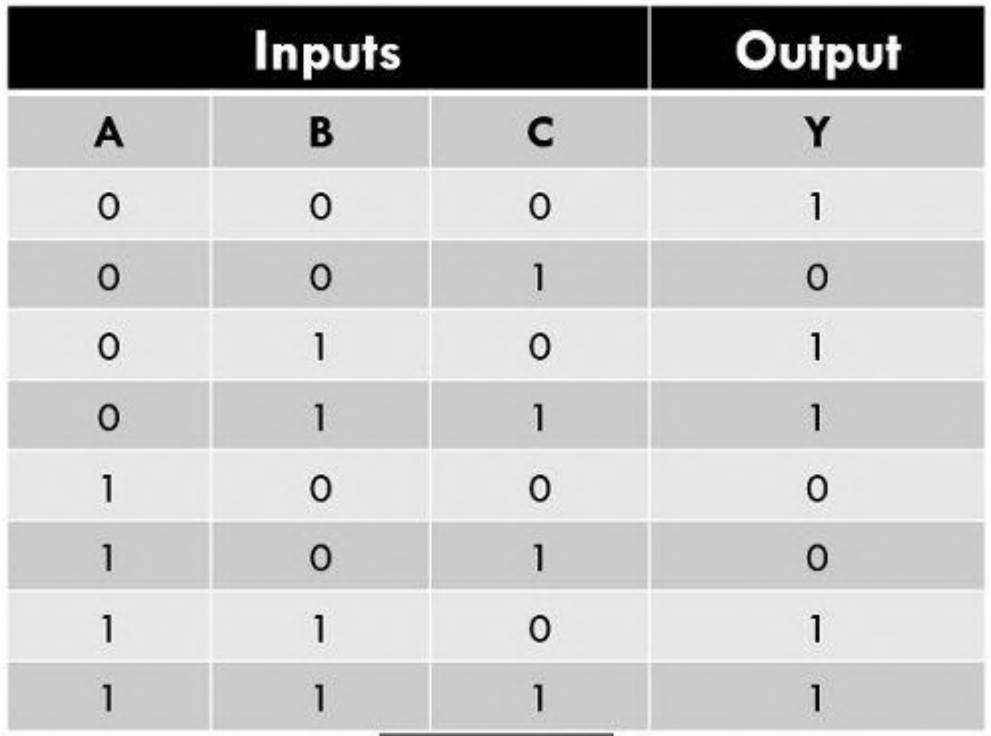


Inputs			Output	Maxterms
A	B	C	X	M_n
0	0	0	0	m_0 = A+B+C
0	0	1	1	m_1 = A+B+C’
0	1	0	1	m_2 = A+B’+C
0	1	1	0	m_3 = A+B’+C’
1	0	0	0	m_4 = A’+B+C
1	0	1	0	m_5 = A’+B+C’
1	1	0	1	m_6 = A’+B’+C
1	1	1	1	m_7 = A’+B’+C’

Figure 2.2

POS expression from a Truth table

To find the POS expression with the help of a truth table **(figure 2.3)**, record the binary values having the output 0. Translate each binary value to the related sum term where each value ‘1’ is substituted with the corresponding variable complement and each 0 is with the corresponding variable.



Inputs			Output
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.3

Y(A,B,C,D) = ∏m (1,4,5)
Only those sum terms are selected where the output value is 0.
$Y = m_1+m_4+m_5$
So, in accordance with the truth table, the boolean function **(Canonical POS form)** in the maxterm is :
$Y = (A+B+C').(A'+B+C).(A'+B+C')$
1 is variable and 0 is the complement of the variable.

Key Differences Between SOP and POS

1. SOP (Sum of product) generates expression in which all the variables in a domain are first multiplied then added. On the contrary, the POS (Product of Sum) represents the boolean expression having variables summed then multiplied with each other.
2. Minterms or product terms are mainly used in the SOP which associates with the high (1) value. Conversely, in POS, Maxterms or sum terms are employed, which produces a low (0) value.
3. In the SOP, method, the value ‘1’ is replaced by the variable and ‘0’ by its complement. In contrast, when it comes to POS a ‘0’ is substituted by the variable and ‘1’ by its complement.
4. At last, all the terms are added with each other in case of SOP. As against, in POS, the terms are multiplied with each other in the last step of the process.

Karnaugh Map (K-map) :

It is quite similar to a truth table where we have various probable values of the input variables and the outcome for each value. A karnaugh map provides an organized way for simplifying boolean expressions and helps in constructing the simplest minimized SOP and POS expression.

Conclusion

The SOP and POS, both forms are used for representing the expressions and also holds equal importance. In an effort for finding whether your reduced boolean expression is correct or not for designing the logical circuit, one can compare the SOP and POS form of expression and check whether they are equivalent or not. Additionally, for any binary value, the resultant of SOP and POS are either be both 1 or 0 based on the binary value.

Difference between SOP and POS in Digital Logic

S.No.	**SOP**	**POS**
1	SOP stands for Sum of Products.	POS stands for Product of Sums.
2	It is a technique of defining the boolean terms as the sum of product terms.	It is a technique of defining boolean terms as a product of sum terms.
3	It prefers minterms.	It prefers maxterms.
4	In the case of SOP, the minterms are defined as ‘m’.	In the case of POS, the Maxterms are defined as ‘M’
5	It gives HIGH(1) output.	It gives LOW(0) output.
6	In SOP, we can get the final term by adding the product terms.	In POS, we can get the final term by multiplying the sum terms.

Unit No 2
Computer Architecture

What is a Number System?

1. A number system is a writing system used to express numbers.

2. It is a set of rules or symbols for representing quantities and performing arithmetic operations.

3. A number system ornumeral system is defined as an elementary system to express numbers and figures.
   It is the unique way of representing of numbers in arithmetic and algebraic structure.
4. Thus, in simple words, the writing system for denoting numbers using digits or symbols in a logical manner is defined as a **Number system**.

Types of Number Systems

Based on the base value and the number of allowed digits, number systems are of many types. The four common types of Number systems are:

- Decimal Number System
- Binary Number System
- Octal Number System
- Hexadecimal Number System

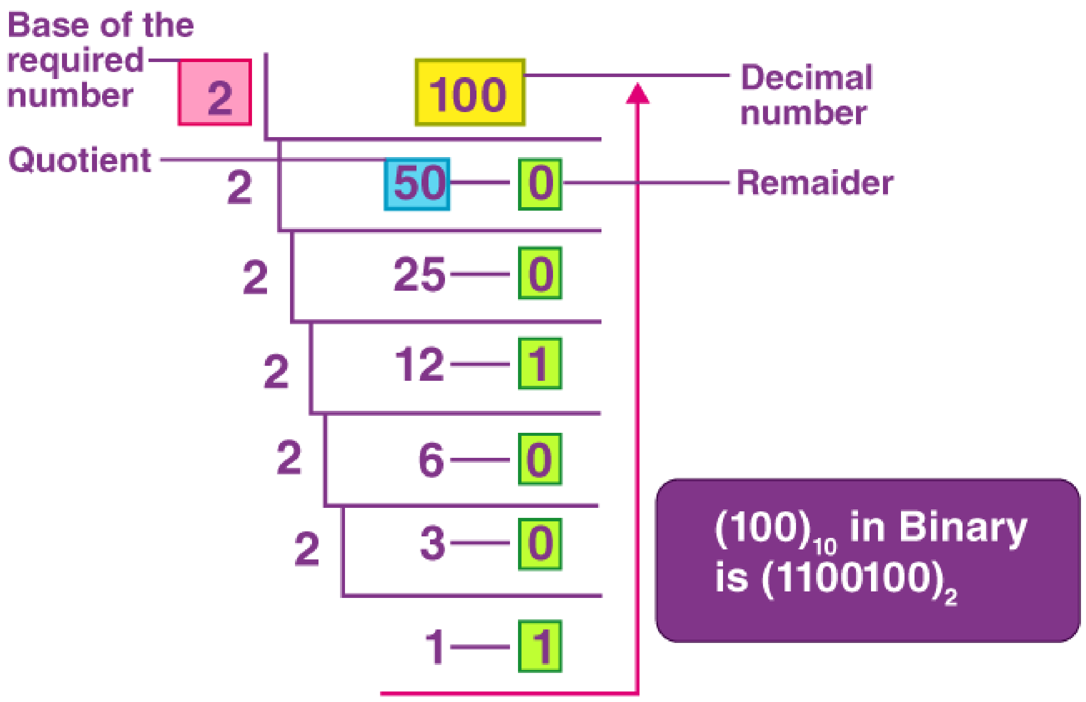
1.Decimal Number System

A number system with a base value of 10 is termed a Decimal number system. It uses 10 digits i.e. 0-9 for the creation of numbers. Here, each digit in the number is at a specific place with a place value of a product of different powers of 10. Here, the place value is termed from right to left as the first place value called units, second to the left as Tens, so on Hundreds, Thousands, etc. Here, units have a place value of 100, tens have a place value of 101, hundreds as 102, thousands as 103, and so on.

2.Binary Number System

A number System with a base value of 2 is termed a Binary number system. It uses 2 digits i.e. 0 and 1 for the creation of numbers. The numbers formed using these two digits are termed Binary Numbers. The binary number system is very useful in electronic devices and computer systems because it can be easily performed using just two states ON and OFF i.e. 0 and 1.

Decimal Numbers 0-9 are represented in binary as 0, 1, 10, 11, 100, 101, 110, 111, 1000, and 1001.



Base of the required number
2
100
Decimal number
Quotient
2
50
0
Remaider
2
25
0
2
12
1
2
6
0
2
3
0
1
1
$(100)_{10}$ in Binary is $(1100100)_2$

3.Octal Numbering System-

The octal number system is a base-8 system using digits 0-7, where each position represents a power of 8. It is commonly used in computing for easy conversion to binary.
'**OCTAL**' is derived from the Latin word 'OCT' which means Eight. The number system with base 8 and symbols ranging between 0-7 is known as the Octal Number System. Each digit of an octal number represents a power of 8.
It is widely used in computer programming and digital systems. Octal number system can be converted to other number systems and visa versa.

For example, an octal number (10)8 is equivalent to 8 in the decimal number system, 001000 in the binary number system and 8 in the hexadecimal number system.

Octal Numbers System Table

We use only 3 bits to represent Octal Numbers. Each group will have a distinct value between 000 and 111.

Octal Digital Value	*Binary Equivalent*
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

4.Hexadecimal Number System-

The **hexadecimal number system** is a type of number system, that has a base value equal to 16. It is also pronounced sometimes as **'hex'**. Hexadecimal numbers are represented by only 16 symbols. These symbols or values are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. Each digit represents a decimal value. For example, D is equal to base-10 13.

Hexadecimal number systems can be converted to other number systems such as binary number (base-2), octal number (base-8) and decimal number systems (base-10).

The **list of 16 hexadecimal digits** with their equivalent decimal, octal and binary representation is given here in the form of a table, which will help in number system conversion. This list can be used as a translator or converter also.

Hexadecimal Number System Table

Below is the table of hexadecimal number systems with equivalent values of the binary and decimal number systems.

Decimal Numbers	4-bit Binary Number	Hexadecimal Number
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D

14	1110	E
15	1111	F

Why Different Systems?

Binary is useful for digital computing because it directly corresponds to the on/off states of electrical circuits.

Decimal is used because it aligns with human's natural counting system.

Hexadecimal is used for compactly representing binary data.

The conversion of one base number to another base number considering all the base numbers such as decimal, binary, octal and hexadecimal with the help of examples. Here, the following number system conversion methods are explained.

- Binary to Decimal Number System
- Decimal to Binary Number System
- Octal to Binary Number System
- Binary to Octal Number System
- Binary to Hexadecimal Number System
- Hexadecimal to Binary Number System

The general representations of number systems are;

Decimal Number – Base 10 – N_{10}

Binary Number – Base 2 – N_2

Octal Number – Base 8 – N_8

Hexadecimal Number – Base 16 – N_{16}

Number System Conversion Table

Binary Numbers	Octal Numbers	Decimal Numbers	Hexadecimal Numbers
0000	0	0	0

0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

How to Convert Decimal Numbers to Binary Numbers?

We can convert the given decimal to binary using different methods such as formula, division method, and so on. In the section, you will learn how to convert decimal numbers to binary in the division method. To convert decimal to binary numbers, proceed with the steps given below:

Step 1: Divide the given decimal number by "2" where it gives the result along with the remainder.

Step 2: If the given decimal number is even, then the result will be whole and it gives the remainder "0"

Step 3: If the given decimal number is odd, then the result is not divided properly and it gives the remainder “1”.

Step 4: By placing all the remainders in order in such a way, the Least Significant Bit (LSB) at the top and Most Significant Bit (MSB) at the bottom, the required binary number will be obtained.

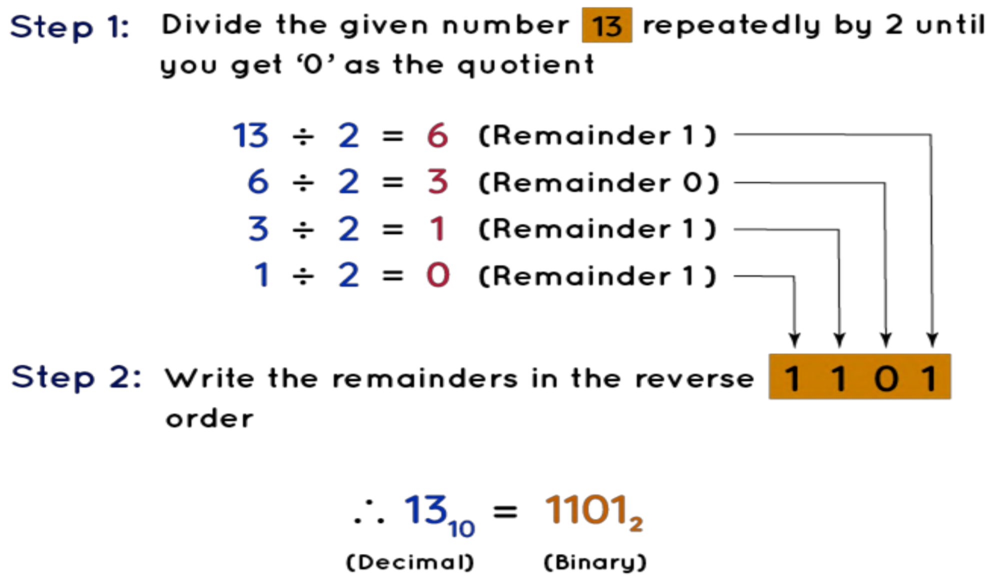
Now, let us convert the given decimal number 294 into a binary number.
Therefore, the binary equivalent for the given decimal number 294_{10} is 100100110_2

Divide by 2	**Result**	**Remainder**	**Binary Value**
294 ÷ 2	147	0	0 (LSB)
147 ÷ 2	73	1	1
73 ÷ 2	36	1	1
36 ÷ 2	18	0	0
18 ÷ 2	9	0	0
9 ÷ 2	4	1	1
4 ÷ 2	2	0	0
2 ÷ 2	1	0	0
1 ÷ 2	0	1	1 (MSB)

294_{10} =100100110_2

Example: Convert the decimal number 13_{10} to binary.

Solution: We will start dividing the given number (13) repeatedly by 2 until we get the quotient as 0. We will note the remainders in order.



Step 1: Divide the given number 13 repeatedly by 2 until you get '0' as the quotient
13 ÷ 2 = 6 (Remainder 1)
6 ÷ 2 = 3 (Remainder 0)
3 ÷ 2 = 1 (Remainder 1)
1 ÷ 2 = 0 (Remainder 1)
Step 2: Write the remainders in the reverse order
1 1 0 1
∴ 13₁₀ = 1101₂
(Decimal)
(Binary)

Division by 2	Quotient	Remainder
13 ÷ 2	6	1 (LSB)
6 ÷ 2	3	0
3 ÷ 2	1	1
1 ÷ 2	0	1 (MSB)

After noting the remainders, we will write them in such a way that the Most Significant Bit (MSB) of the binary number is written first, followed by the rest. Therefore, the binary equivalent for the given decimal number 13_{10} is 1101_2. This means that $13_{10} = 1101_2$.

Decimal to Binary Table

There are different methods of converting numbers from decimal to binary. When we convert numbers from decimal to binary, the base of the number changes from 10 to 2. It should be noted that all decimal numbers have their equivalent binary numbers. The following table shows the decimal to binary chart of the first 20 whole numbers.

Decimal Numbers	Binary Numbers

Decimal Numbers	**Binary Numbers**
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111
16	10000

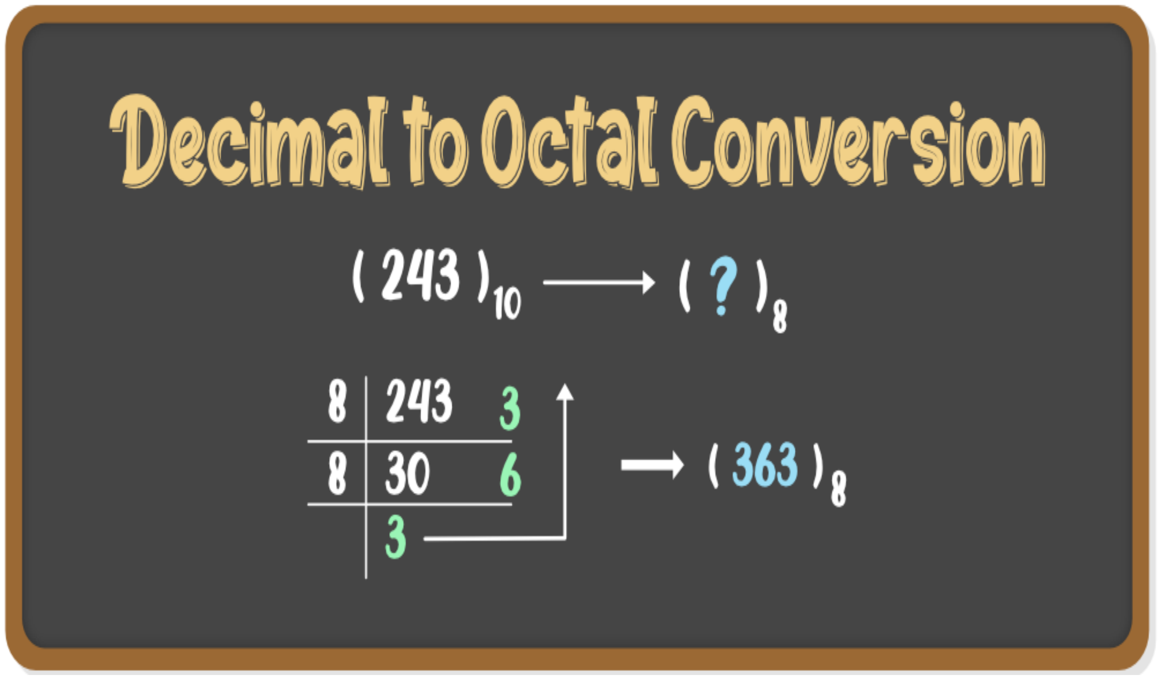
Decimal Numbers	**Binary Numbers**
17	10001
18	10010
19	10011
20	10100

Decimal to Octal Conversion

To convert a decimal number to an octal number follow these simple steps:
Step 1: Divide the given decimal number by 8.
Step 2: Write down the quotient and remainder obtained.
Step 3: Divide the quotient obtained by 8.
Step 4: Repeat step 2 and step 3 until the quotient becomes 0.
Step 5: Write the obtained remainder in reverse order.

Example: Represent $164_{(10)}$ as Octal Number.
Solution:

164/8 , Quotient = 20 and Remainder = 4
20/8 , Quotient = 2 and Remainder = 4
2/8 , Quotient = 0 and Remainder = 2
Now, By writing obtained remainders in reverse order we get, 244.
Hence 2448 is octal representation of 16410
The image added below shows binary to octal conversion.



Steps to Convert Decimal to Octal

The steps to convert a decimal number to its equivalent octal number are as follows:

Step 1: Note down the given decimal number.

Step 2: If the specified decimal number is less than 8, its octal equivalent is the same number.

Step 1: Note down the given decimal number.

Step 2: If the specified decimal number is less than 8, its octal equivalent is the same number.

$0_{(10)}$ =08

$1_{(10)}$ =18

$2_{(10)}$ =28

$3_{(10)} = 3_8$

$4_{(10)} = 4_8$

$5_{(10)} = 5_8$

$6_{(10)} = 6_8$

$7_{(10)} = 7_8$

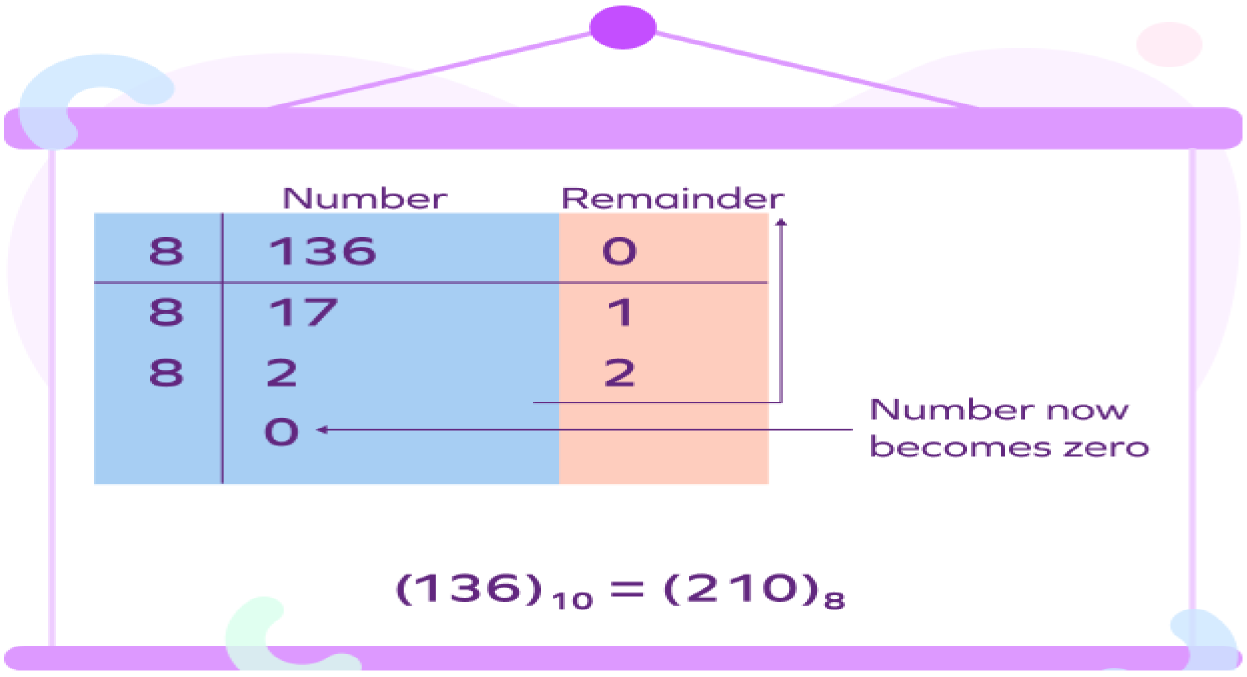
Step 3: If the number is greater than 7, divide it by 8.

Step 4: Note the remainder we obtain after division.

Step 5: Steps 3 and 4 should be repeated until the quotient is less than 8.

Step 6: Now, write the remainders in reverse order (bottom to top). The outcome is the octal number that corresponds to the given decimal number.

Let's understand the steps with an example.



How to Convert Decimal to Hexadecimal

Step 1: Divide the number by 16. Note down the quotient and remainder.

If the quotient is 0, the remainder is the equivalent hexadecimal number.

If the quotient is not 0, go to step 2.

Step 2: Divide the quotient in step 1 by 16. Again, note down the quotient and remainder.

If the quotient is 0, write the remainders in reverse order to find the hexadecimal number.

If the quotient is not 0, repeat the process until we get 0 as a quotient.

(Note that when the remainder is greater than 9, we refer to the decimal to hexadecimal table mentioned above to write its hexadecimal equivalent. Replace 10, 11, 12, 13, 14, 15 by A, B, C, D, E, F respectively.)

Example 1: Find the hexadecimal equivalent of (152)10.

Division	Quotient	Remainder
152÷16	9	8
9÷16	0	9

Write the remainders in reverse order.

$(152)_{10} = (98)_{16}$

Example 2: Convert from decimal to hexadecimal: 450_{10}

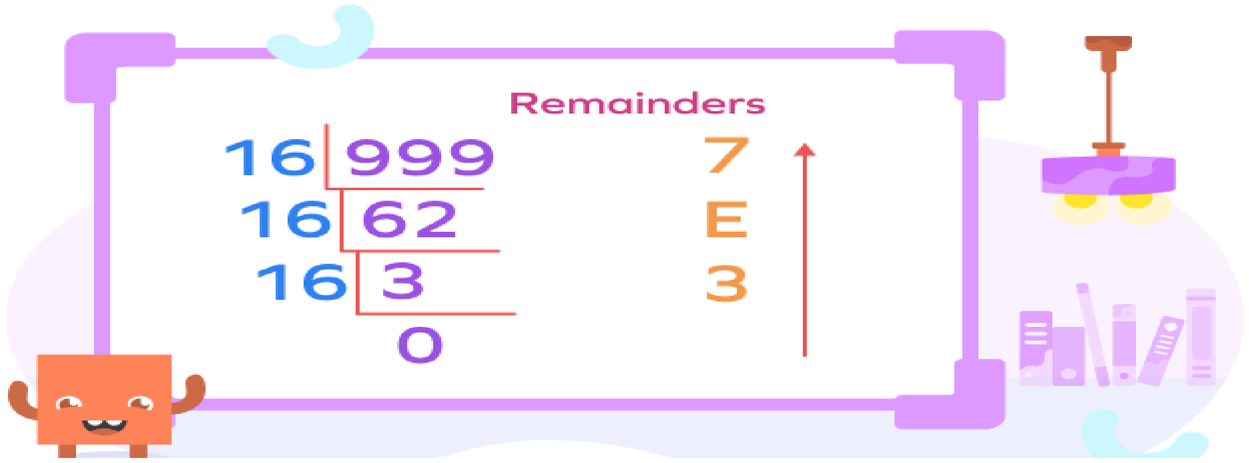
Division	Quotient	Remainder (decimal value)	Remainder(Hexadecimal value)
450÷16	28	2	2
28÷16	1	12	C
1÷16	0	1	1

Writing the remainders in the reverse order, we get

$450_{10} = (1C2)_{16}$

Example 3: Convert to hexadecimal: 999_{10}

You can also show the division using columns.



$999_{10} = 3E7_{16}$

How to Convert Binary to Decimal Numbers?

To convert the binary number to a decimal number, we use the multiplication method. In this conversion process, if a number with base n has to be converted into a number with base 10, then each digit of the given number is multiplied from the Most

Significant Bit (MSB) to the Least Significant Bit (LSB) with reducing the power of the base.

Binary to Decimal Conversion Steps

- First, write the given binary number and count the powers of 2 from right to left (powers starting from 0)
- Now, write each binary digit (right to left) with the corresponding powers of 2 from (right to left), such that first binary digit (MSB) will be multiplied with the greatest power of 2.
- Add all the products in the above step
- The final answer will be the required decimal number

Let us understand this conversion with the help of an example.

Example of Binary to Decimal Conversion:

Convert the binary number $(1101)_2$ into a decimal number.

Solution:

Given binary number = $(1101)_2$

Now, multiplying each digit from MSB to LSB with reducing the power of the base number 2.

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 8 + 4 + 0 + 1$

$= 13$

Solved Examples

Q.1: Convert the binary number 1001 to a decimal number.

Solution: Given, binary number = 1001_2

Hence, using the binary to decimal conversion formula, we have:

$1001_2 = (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

$= 8 + 0 + 0 + 1$

$= (9)_{10}$

Q.2: Convert 1101001_2 into an equivalent decimal number.

Solution: Using binary to decimal conversion method, we get;

$(1101001)_2 = (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

$= 64 + 32 + 0 + 8 + 0 + 0 + 1$

$= (105)_{10}$

Q.3: Convert $(11110111)_2$ into base-10 number system.

Solution: Using binary to decimal conversion method, we get;

$(11110111)_2 = (1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$

$= 128 + 64 + 32 + 16 + 0 + 4 + 2 + 1$

$= (247)_{10}$

Thus, the equivalent decimal number for the given binary number $(1101)_2$ is $(13)_{10}$

Conversion of Binary to Octal

Since binary numbers are used in computers in the form of bits or bytes and octal numbers are used in electronics, direct conversion from binary to octal is not a method. There are two kinds of methods that are used in the binary to octal conversion.

Method 1: Converting Binary to Decimal **then from Decimal to Octal**

Here are the steps that need to be followed for this method.

- Step 1: Identify the binary number
- Step 2: Convert binary to decimal by multiplying each digit by 2^{n-1} where 'n' is the position of the digit from the right.
- Step 3: The derived answer is the decimal number for the given binary number
- Step 4: Divide the decimal number by 8
- Step 5: Note the remainder
- Step 6: Continue the above two steps with the quotient till the quotient is zero
- Step 7: Write the remainder in the reverse order
- Step 8: The answer is the required octal number for the binary number

For example: Convert the binary number $(1011101)_2$(1011101)2 to an octal

number.

Solution: According to method 1, first convert the binary number to decimal number

$(1011101)_2$(1011101)2 =

$(1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$

= 64 + 0 + 16 + 8 + 4 + 0 + 1

= 93

$(1011101)_2$(1011101)2 = $(93)_{10}$

The next step is to convert the decimal number to an octal number by dividing 93 by 8.

93 divided by 8 will give 5 as remainder and 11 as the quotient

11 divided by 8 will give 3 as remainder and 1 as the quotient

1 divided by 8 will give 1 as remainder and 0 as the quotient

Collect the remainders in reverse order we get 1 3 5

Therefore, binary number \((1011101)_{2}\) = \((135)_{8}\)

Example 1: Convert 1010101_2 to octal

Solution:

Given binary number is 1010101_2

First, we convert given binary to decimal

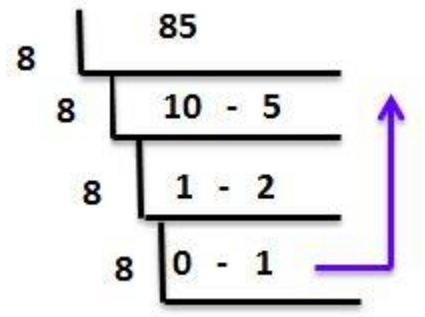
$1010101_2 = (1 * 2^6) + (0 * 2^5) + (1 * 2^4) + (0 * 2^3) + (1 * 2^2) + (0 * 2^1) + (1 * 2^0)$

= 64 + 0 + 16 + 0 + 4 + 0 + 1

= 64 + 21

010101_2= 85 (Decimal form)

Now we will convert this decimal to octal form

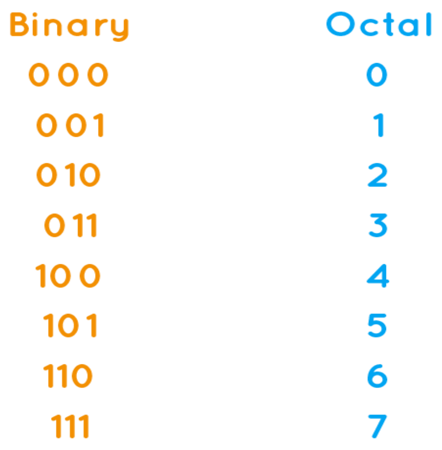


Therefore, the equivalent octal number is 125_8.

Method 2: Converting Binary to Octal by grouping

Here are the steps that need to be followed for this method.

- Step 1: Identify the binary number i.e. the digits should be either 0 or 1 with base 2.
- Step 2: Group all the 0 to 1 in a set of three starting from the right side.
- Step 3: Add 0's to the left if it does not form a group of three. Each group must have three digits.
- Step 4: Look at the binary to octal conversion table to get the accurate numbers.
- Step 5: Once obtained, that number is the octal number



Binary	Octal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

For example: Convert the binary number $(01110101)_2$(01110101)2 to an octal number.

Solution: Using the grouping method, set the binary number into three numbers in each group.

$(01110101)_2$= 001 110 101 = 1 6 5

$(01110101)_2 = (165)_8$

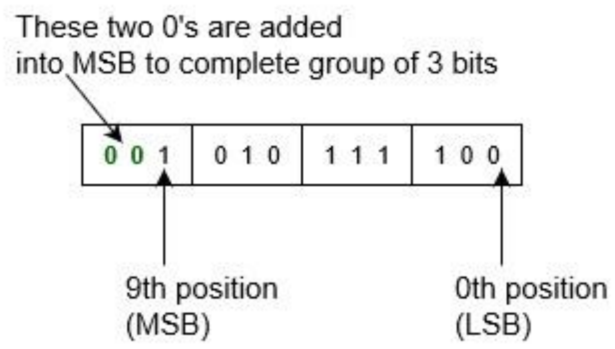
So, if you make each group of 3 bit of binary input number, then replace each group of binary number from its equivalent octal digits. That will be octal number of given number number. Note that you can add any number of 0's in leftmost bit (or in most significant bit) for integer part and add any number of 0's in rightmost bit (or in least significant bit) for fraction part for completing the group of 3 bit, this does not change value of input binary number.

So, these are following steps to convert a binary number into octal number.

- Take binary number
- Divide the binary digits into groups of three (starting from right) for integer part and start from left for fraction part.
- Convert each group of three binary digits to one octal digit.

This is simple algorithm where you have to grouped binary number and replace their equivalent octal digit.

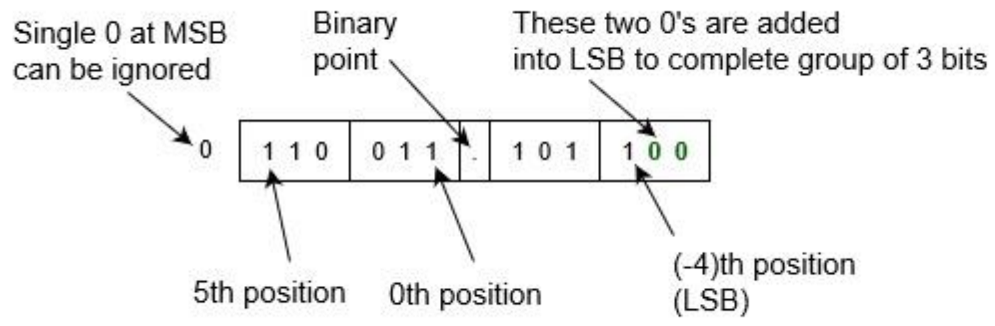
Example-1 − Convert binary number 1010111100 into octal number. Since there is no binary point here and no fractional part. So,



Therefore, Binary to octal is.

= $(1010111100)_2$
= $(001\ 010\ 111\ 100)_2$
= $(1\ 2\ 7\ 4)_8$
= $(1274)_8$

Example-2 Convert binary number 0110 011.1011 into octal number. Since there is binary point here and fractional part. So,



Therefore, Binary to octal is.= $(0110\ 011.1011)_2$

= $(0\ 110\ 011\ .\ 101\ 1)_2$
= $(110\ 011\ .\ 101\ 100)_2$
= $(6\ 3\ .\ 5\ 4)_8$

= $(63.54)_8$

Binary to Hexadecimal

A binary-to-hexadecimal conversion is done to convert a binary number (base 2) to its equivalent hexadecimal number (base 16). It is done by the given methods.

Direct Method: Using Table
In this method, we directly represent a group of binary digits (of 4 bits) to its hexadecimal value using the conversion table.

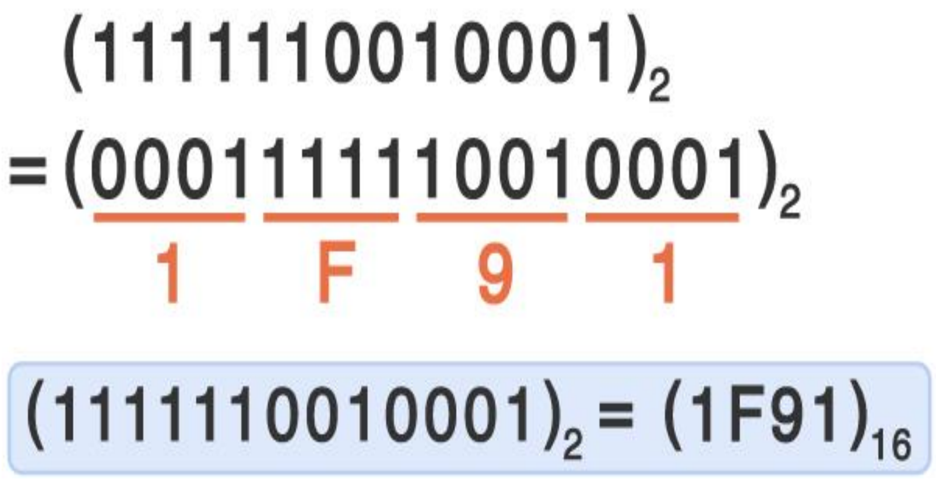
Let us convert $(11010)_2$ into its corresponding hexadecimal number.
Step 1: Grouping 11010 into 4 bits starting from the right, we have (1) and (1010).
Step 2: Since the first group is not of four bits, we add zeros to the front. Now, the groups (0001) and (1010) are of four bits.
Step 3: We find their corresponding hexadecimal values using the conversion table.

By converting each group into its corresponding hexadecimal values, we get

$(0001)_2 = (1)_{16}$ and $(1010)_2 = (A)_{16}$
Step 4: Taking the values based on the order of the groups, we get
$(11010)_2 = (1A)_{16}$

Convert $(1111110010001)_2$ into its equivalent hexadecimal number.

Solution:


$(1111110010001)_2$
$=(\underbrace{0001}_{1}\underbrace{1111}_{F}\underbrace{1001}_{9}\underbrace{0001}_{1})_2$
$(1111110010001)_2 = (1F91)_{16}$

(0001) → 1
(1111) → F
(1001) → 9
(0001) → 1

$(1111110010001)_2 \rightarrow (1F91)_{16}$

What is Octal to Decimal Conversion?

Octal to decimal conversion takes place when we want to know the equivalent of a number in the number system. The number system is of four types - Binary number system, Octal number system, Decimal number system, and Hexadecimal number system. Each number system has its own base numbers that help in identifying which type of number it is. These base numbers also help in the octal to decimal conversion. The base number for octal numbers is 8 and the base number for decimal numbers is 10.

Octal Number System

A number system with its base as 8 and uses digits from 0 to 7 is called Octal Number System. The word octal is used to represent the numbers that have eight as the base. The octal numbers have many applications and importance such as it is used in computers and digital numbering systems. In the number system, octal numbers can be converted to binary numbers, decimal numbers, and to hexadecimal numbers. Some of the examples of octal numbers are $(47)_8$(47)8, $(120)_8$(120)8. The octal numbers are represented with a power of 8. For example: $(547)_8$(547)8 = $5 \times 8^2 + 4 \times 8^1 + 7 \times 8^0$.

Decimal Number System

The number system with its base as 10 and uses ten digits: 0,1,2,3,4,5,6,7,8 and 9 are called decimal number system. The decimal number system is the system that we generally use to represent numbers in real life. If any number is represented without a base, it means that its base is 10. For example: 65_{10}, 687_{10}, 4198_{10} 6510, 68710, 419810 are some examples of numbers in the decimal number system.

Steps to Convert Octal to Decimal

As with any other conversion in the number system, octal to decimal conversion is also done by using its base number. To convert octal to decimal, we need to multiply the octal digits with the power of 8 starting from the right-hand side and gradually decreasing to zero to sum up, all the products. Here are the steps to convert a number from octal to decimal:

- Step 1: Since an octal number only uses digits from 0 to 7, we first arrange the octal number with the power of 8.
- Step 2: We evaluate all the power of 8 values such as 8^0 is 1, 8^1 is 8, etc., and write down the value of each octal number.
- Step 3: Once the value is obtained, we multiply each number.
- Step 4: Final step is to add the product of all the numbers to obtain the decimal number.

Let us look at an example, convert $(140)_8$(140)8 into a decimal number.

Step 1: Write 140 with the power of 8. Start from the right-hand side.

$1 \times 8^2 + 4 \times 8^1 + 0 \times 8^0$

Step 2: Evaluate the power of 8 values for each octal number.

$8^2 = 64$, $8^1 = 8$, $8^0 = 1$

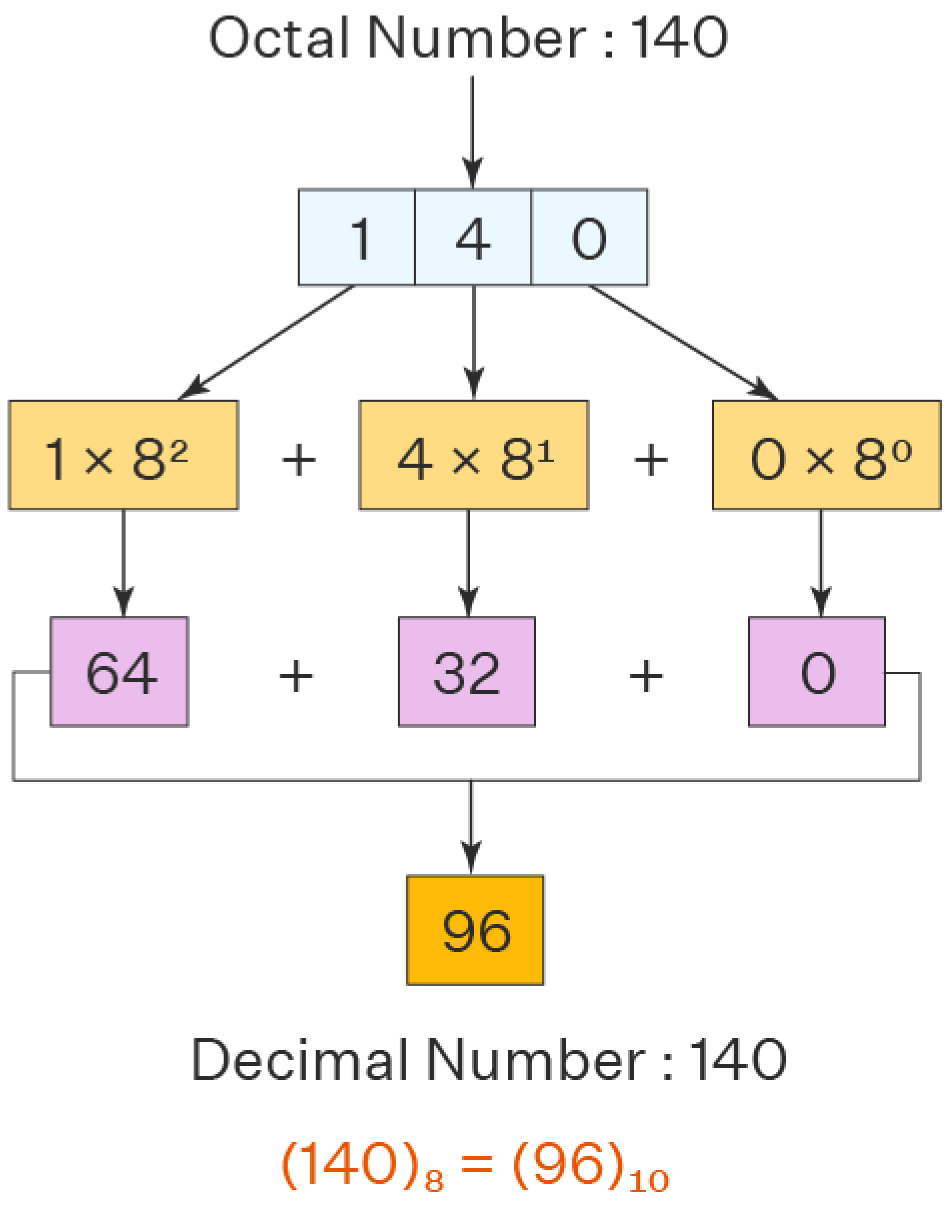
Step 3: Multiply each of the power of 8 numbers with the respective numbers.

$1 \times 64 + 4 \times 8 + 0 \times 1 = 64 + 32 + 0$

Step 4: Add the values to obtain the decimal number.

$64 + 32 + 0 = 96$.

Therefore, $(140)_8$(140)8 = $(96)_{10}$(96)10.



Convert Octal to Decimal with Decimal Point

To convert octal to decimal number with a decimal point, we need to follow the same procedure as did in the previous section. However, the power or the exponents of 8 will vary after the decimal point. Since we are moving towards the right-hand side with the exponents increasing, the exponents after the decimal point will decrease or be negative. Let us look at an example.

Convert $(246.28)_8$(246.28)8 into a decimal number. We will follow the same steps as before.

Step 1: Write 140 with the power of 8. Start from the right-hand side. Here, the power of 8 will be negative after the decimal point.

$2 \times 8^2 + 4 \times 8^1 + 6 \times 8^0 + 2 \times 8^{-1} + 8 \times 8^{-2}$

Step 2: Evaluate the power of 8 values for each octal number.

$8^2 = 64$, $8^1 = 8$, $8^0 = 1$, $8^{-1} = 1/8$, $8^{-2} = 1/8^2$ or $1/64$

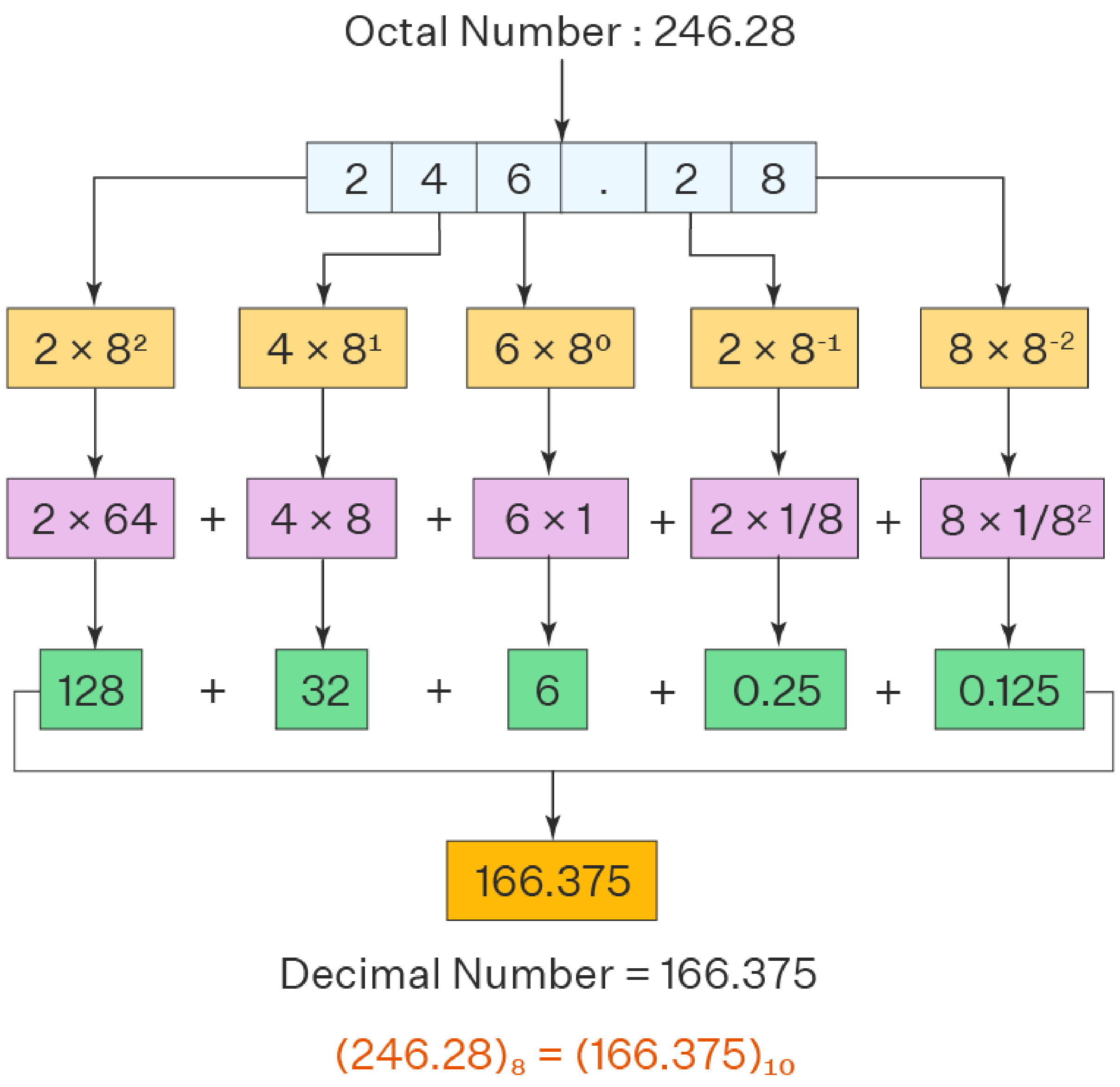
Step 3: Multiply each of the power of 8 numbers with the respective numbers.

$2 \times 64 + 4 \times 8 + 6 \times 1 + 2 \times 1/8 + 8 \times 1/64 = 128 + 32 + 6 + 0.25 + 0.125$

Step 4: Add the values to obtain the decimal number.

$128 + 32 + 6 + 0.25 + 0.125 = 166.375$.

Therefore, $(246.28)_8$(246.28)8 = $(166.375)_{10}$(166.375)10.



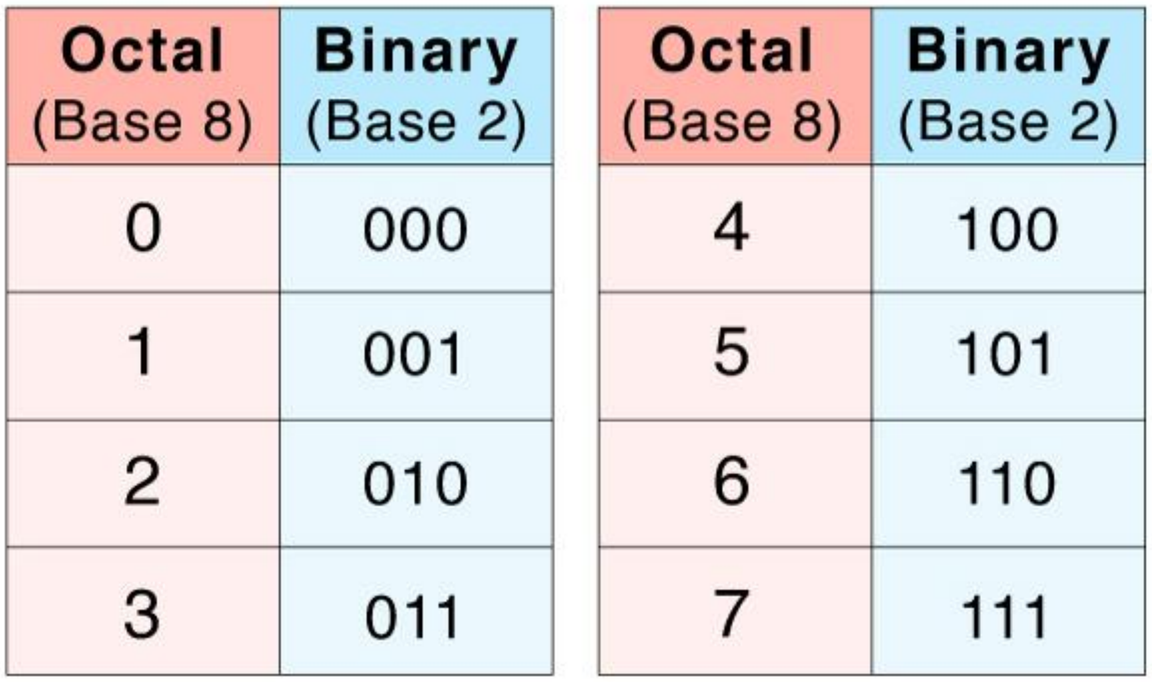
Octal to Binary

An octal-to-binary conversion is done to convert an octal number (base 8) to its equivalent binary number (base 2). Here are the methods to convert an octal number to its binary counterpart.

Direct Method: Using Table

The octal number system has 8 digits from 0 to 7, represented in their equivalent binary using 3 bits.

Let us convert $(363)_8$ into its binary number.
Step 1: Grouping 363 into individual digits, we have 3, 6, and 3.
Step 2: Now, we find their corresponding binary numbers using the conversion table.



Octal (Base 8)	Binary (Base 2)	Octal (Base 8)	Binary (Base 2)
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

Octal Number	**Binary Number**
0	000
1	001

Octal Number	Binary Number
2	010
3	011
4	100
5	101
6	110
7	111

By converting each octal digit of $(363)_8$ to their binary numbers, we get

Octal Value	3	6	3
Binary Value	011	110	011

Here, we observe that each octal digit gives us a 3-bit binary number. Otherwise, we add zeros to maintain the correct number of digits.

Step 3: Taking the values from the first to last, $(363)_8$ equals $(011110011)_2$ or $(11110011)_2$.

For Fractional Octal Numbers

Similarly, we convert the fractional octal numbers into their corresponding binary.

Now, converting $(452.01)_8$ to its equivalent binary, we get

For the Integral Part:

4 → 100, 5 → 101, and 2 → 010

For the Fractional Part:

0 → 000 and 1 → 001

Thus, $(452.01)_8 = (100101010.000001)_2$

Convert $(53)_8$ into binary using the conversion table.

Solution:

Grouping 53 into individual digits, we have 5 and 3.
By converting each octal digit of $(53)_8$ to the binary, we get

$(5)_8 = (101)_2$

(3)8 = (011)2

Placing the values from the first to last, (53)8 equals (101011)2.

Octal Value 53   Binary Value 101011

Indirect Method: Without Using Table

There is another way by which each digit in any octal number is represented to its corresponding binary number without using the octal-to-binary conversion table.

Let us convert an octal number $(73)_8$ into its corresponding binary.
First, we convert 73 into a decimal number and then decimal to binary.

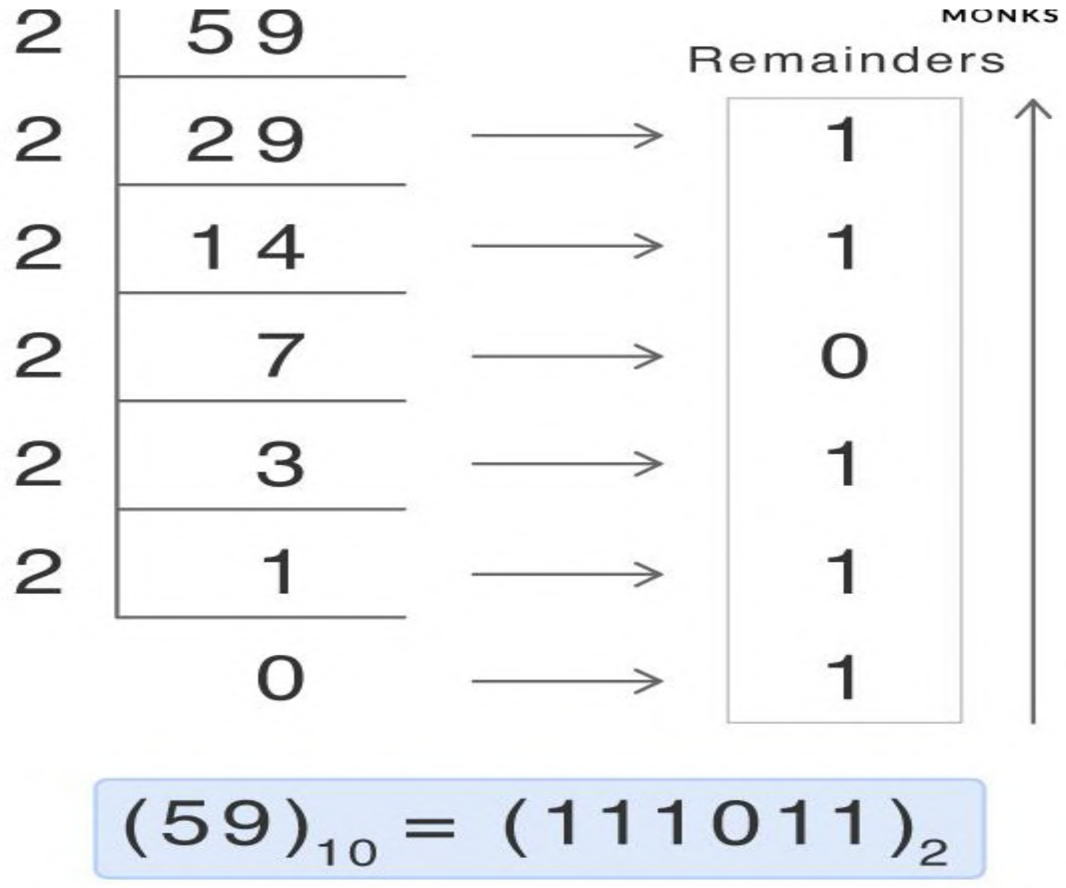
Step 1: Octal to Decimal
While converting $(73)_8$ to its decimal number, we multiply each digit from the right by the corresponding powers of 8, as shown.

Octal Value	7	3
Decimal Value	7×8^1	3×8^0

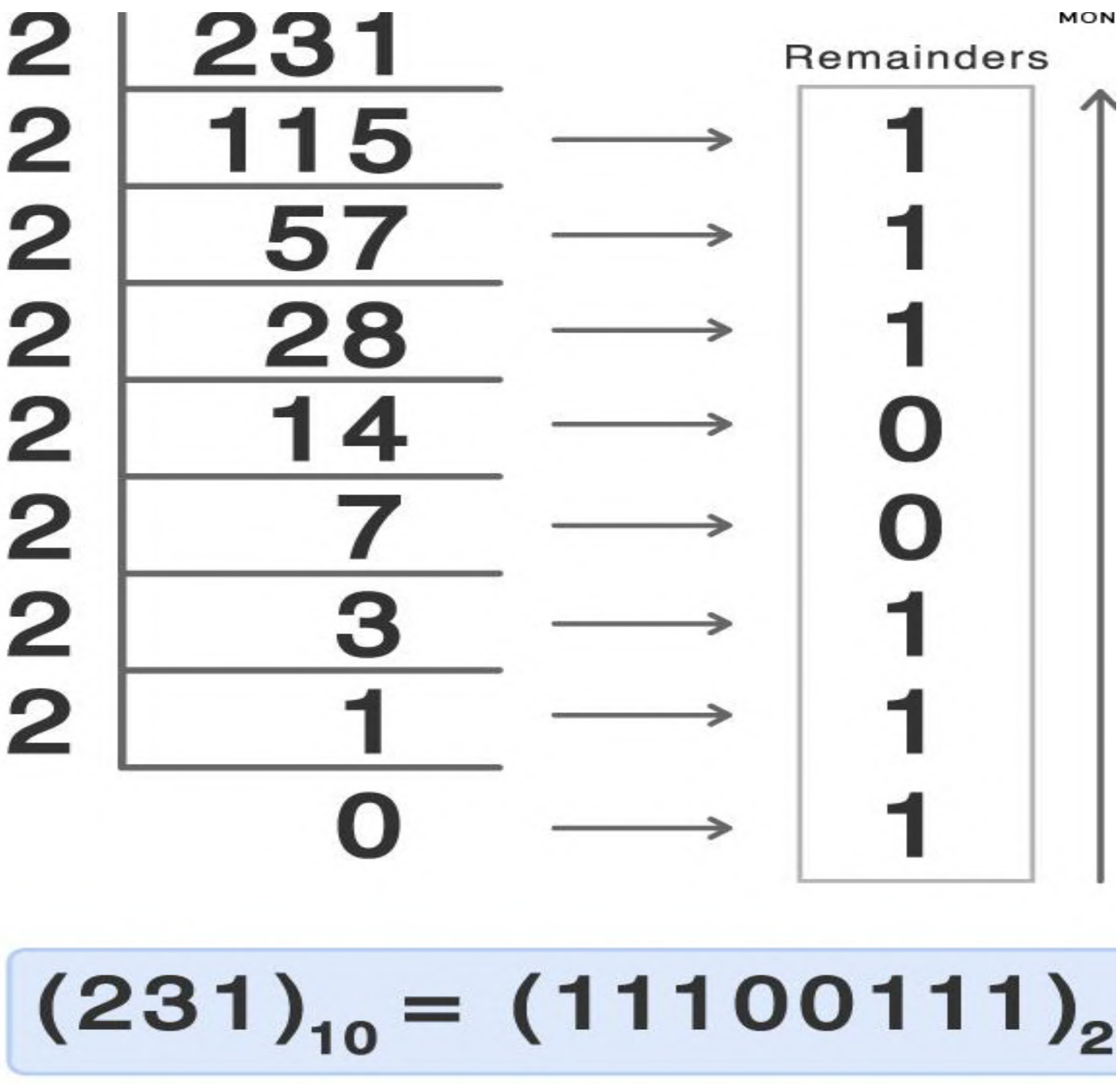
Now, on adding the values, we get the decimal number

$(7 \times 8^1) + (3 \times 8^0) = 56 + 3 = 59$

Step 2: Decimal to Binary
Now, converting $(59)_{10}$ into its corresponding binary, we get



Thus, $(73)_8 = (111011)_2$.



How to convert octal to hexadecimal ?

Using the below two methods, we can convert the octal number system into the hexadecimal number system.

1. Convert the octal number into **binary** and then convert the binary into hexadecimal.

2. Convert the octal number into **decimal** and then convert the decimal into hexadecimal.

Let's convert the octal number into the hexadecimal number system.

Octal to Binary to Hexadecimal

Let's convert $(56)_8$ into hexadecimal

Step 1 : Convert $(56)_8$ into Binary

In order to convert the octal number into binary, we need to express every octal value using 3 binary bits.

Binary equivalent of **5** is **$(101)_2$**.

Binary equivalent of **6** is **$(110)_2$**.

= $(56)_8$

= (101) (110)

= $(101110)_2$

Step 2 : Convert $(101110)_2$ into Hexadecimal

In order to convert the binary number into hexadecimal, we need to group every 4 binary bits and calculate the value [From left to right].

$(101110)_2$ in hexadecimal

= $(101110)_2$

= (10)(1110)

= (2) (14)

= $(2e)_{16}$

14 equivalent hexadecimal is **e**.

This method is relatively easy compared to the below method.

Octal to Decimal to Hexadecimal

Step 1: Convert $(56)_8$ into Decimal

= $5*8^1+6*8^0$

= 40+6

= $(46)_{10}$

Step 2: Convert $(46)_{10}$ into hexadecimal
16|46

16|**214**

= $(2e)_{16}$

Conversion from Hex to Decimal

As we know, number systems can be converted from one base to another. Thus, we can convert hexadecimal numbers to decimal easily. This number system conversion can be done as explained in the example given below:

Example:

Convert 7CF (hex) to decimal.

Solution:

Given hexadecimal number is 7CF.

In hexadecimal system,

7 = 7

C = 12

F = 15

To convert this into a decimal number system, multiply each digit with the powers of 16 starting from units place of the number.

7CF = $(7 \times 16^2) + (12 \times 16^1) + (15 \times 16^0)$

= $(7 \times 256) + (12 \times 16) + (15 \times 1)$

= 1792 + 192 + 15

= 1999

Example 1:

Convert $(1DA6)_{16}$ to decimal.

Solution:

$(1DA6)_{16}$

Here,

1 = 1

D = 13

A = 10

6 = 6

Thus,

$(1DA6)_{16} = (1 \times 16^3) + (13 \times 16^2) + (10 \times 16^1) + (6 \times 16^0)$

$= (1 \times 4096) + (13 \times 256) + (10 \times 16) + (6 \times 1)$

= 4096 + 3328 + 160 + 6

= 7590

Therefore, $(1DA6)_{16} = (7590)_{10}$

Example 2:

Convert $(E8B)_{16}$ to decimal system.

Solution:

$(E8B)_{16}$

Here,

E = 14

8 = 8

B = 11

Thus,

$(E8B)_{16} = (14 \times 16^2) + (8 \times 16^1) + (11 \times 16^0)$

$= (14 \times 256) + (8 \times 16) + (11 \times 1)$

$= 3584 + 128 + 11$

$= 3723$

Therefore, $(E8B)_{16} = (3723)_{10}$

Convert Hexadecimal to Binary

To convert hexadecimal to a binary number we need to first convert the hexadecimal number to a decimal number to finally convert it to a binary number. One of the most important aspects to remember here is every hexadecimal number will produce 4 binary digits. The hexadecimal to binary conversion can occur in two methods - First, after the hexadecimal is converted to a decimal number, we convert the decimal number by using the division process to obtain the binary number. Second, we can directly use the hexadecimal to decimal to binary conversion table. Let us look at the steps of both methods.

Method 1: Convert Hexadecimal to Decimal to Binary (without conversion table)

This method requires both multiplication and division of numbers using the respective base numbers. The hexadecimal base number is 16, the base number of a decimal number is 10, and the base of a binary number is 2. Let us look at the steps:

- Step 1: Write the hexadecimal number and find its equivalent decimal number.
- Step 2: To find the decimal equivalent, we multiply each digit with 16^{n-1}, where the digit is in its nth position.
- Step 3: After multiplying the numbers, add the product of those numbers to obtain the decimal number.
- Step 4: To convert decimal to binary, we divide the decimal number by 2 by keeping the remainder aside and dividing the quotient by 2 until we arrive at zero.
- Step 5: Once the quotient is zero, we arrange the remainder from bottom to top i.e. reverse order to obtain the binary number.

Let us look at an example for a better understanding. Convert hexadecimal (100)16

to binary.

Step 1 + 2: Convert (100)16

to decimal by multiplying each digit with 16^{n-1}. Multiply it

(100)16

$= 1 \times 16^{(3-1)} + 0 \times 16^{(2-1)} + 0 \times 16^{(1-1)}$

(100)16

$= 1 \times 16^2 + 0 \times 16^1 + 0 \times 16^0$

Step 3: Multiply the numbers and add the product to obtain the decimal number.

(100)16

= 1 × 256 + 0 × 16 + 0 × 1

(100)16

= 256 + 0 + 0

(100)16

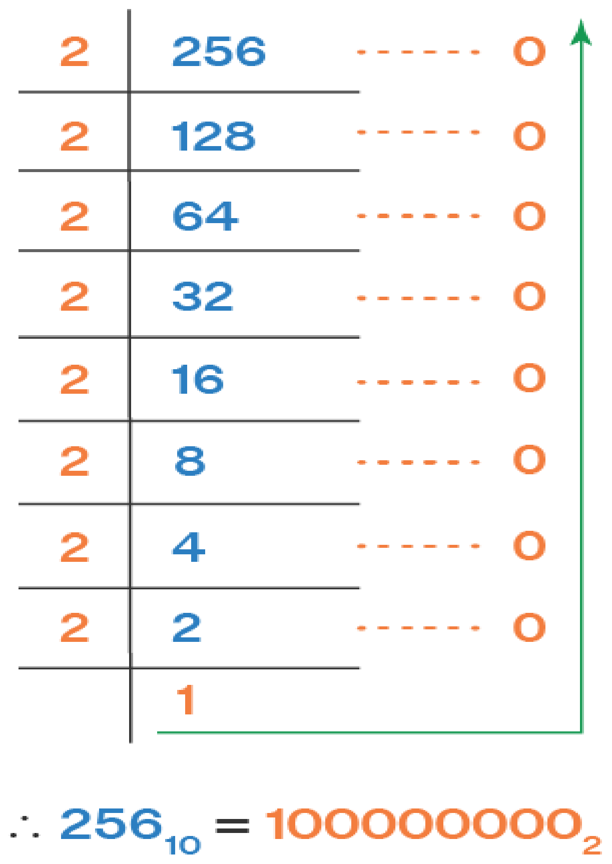
= 256

Therefore, (100)16

= (256)10

Step 4: Convert the decimal number (256)10

to a binary number by dividing the number by 2 until the quotient is zero.



Therefore, (256)10

= (100000000)2

Step 5: Once the binary is obtained, the conversion is done.

Hence, (100)16

= (100000000)2

.

Method 2: Convert Hexadecimal to Decimal to Binary (with conversion table)

This method is a direct procedure by just looking at the conversation table we can convert hexadecimal to binary. The steps are fairly simple, lets look at them:

- Step 1: Write the hexadecimal
- Step 2: Find the equivalent decimal of each of the digits by looking at the conversion table.

- Step 2: Once the decimal number is obtained, looking at the same table we can convert it to a binary.
- Step 3: Combine all the binary numbers together to obtain the final binary number.

Let us look at an example for a better understanding. Convert hexadecimal

$(E5B)_{16}$ to binary.

Step 1: We have the hexadecimal as $(E5B)_{16}$

.

Step 2: Looking at the conversion table, find the equivalent of each digit.

E = (14)10

, 5 = $(5)_{10}$, B = (11)10

Step 3: Once the decimal of each digit is obtained, looking at the conversion table convert each decimal number to binary.

(14)10

= (1110)2

(5)10

= (0101)2

(11)10

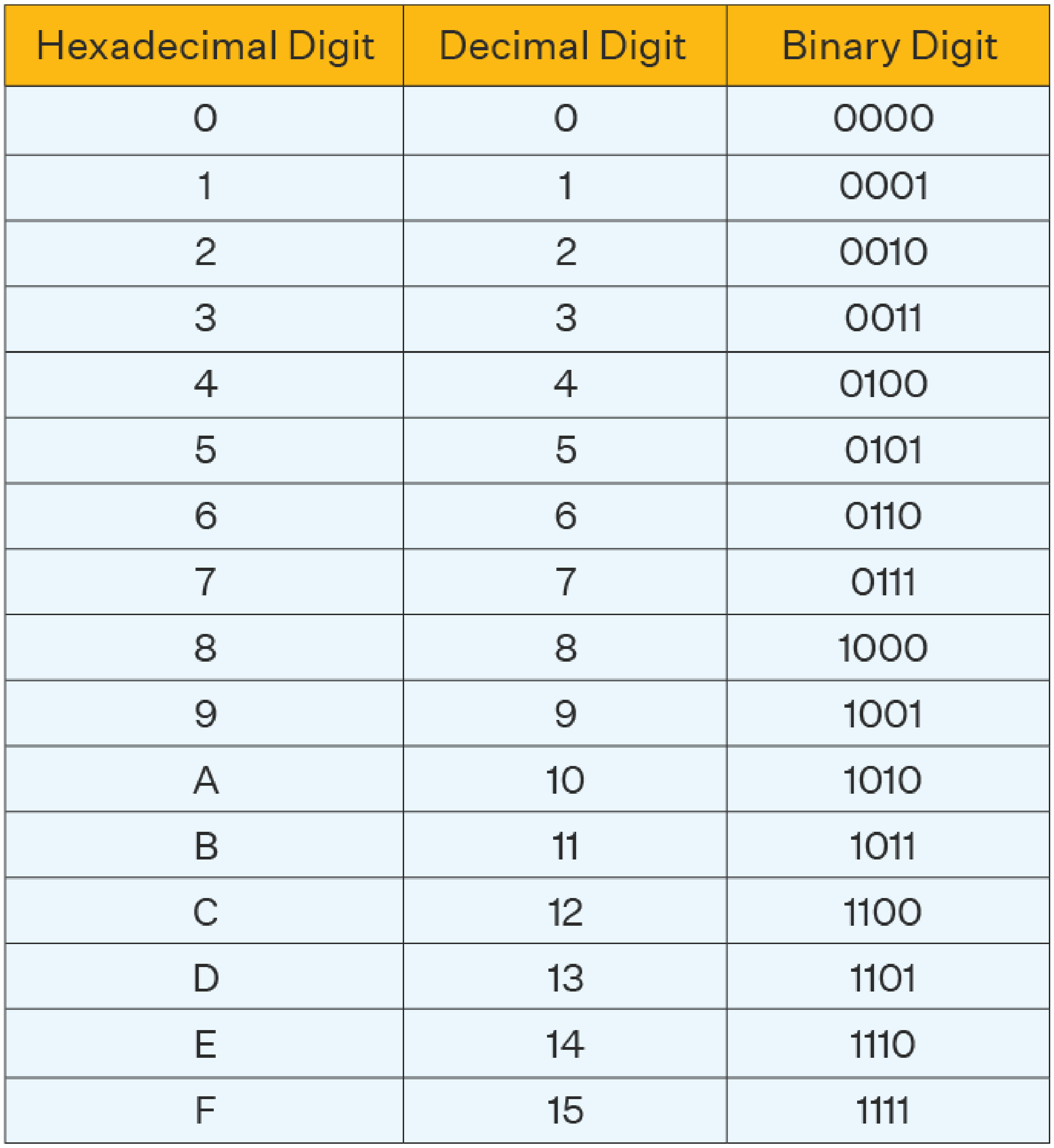
= (1011)2

Step 4: Combine all the binary numbers together to obtain the final one.

Therefore, $(E5B)_{16}$

= (111001011011)2

.



Hexadecimal Digit	Decimal Digit	Binary Digit
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Convert Hexadecimal to Binary With Decimal Point

To convert the hexadecimal digit to binary, we use a similar method as used in the previous section. We use the conversion table to convert hexadecimal to binary. While converting with the decimal point, we use the same steps but do not take into consideration the zero placed on the rightmost side since they are called trailing zeros. Let us look at an example, convert $(0.C48)_{16}$

to binary.

Step 1: We have the hexadecimal as $(0.C48)_{16}$

.Step 2: Looking at the conversion table, find the equivalent of each digit. We do not take the zero into consideration.

C = $(12)_{10}$

, 4 = $(4)_{10}$, 8 = $(8)_{10}$

Step 3: Once the decimal of each digit is obtained, looking at the conversion table convert each decimal number to binary.

$(12)_{10}$

= (1100)2

$(4)_{10}$

= $(0100)_2$

$(8)_{10}$

= $(1000)_2$

Step 4: Combine all the binary numbers together to obtain the final one. The zero before the decimal will be written along with the final binary number.

Therefore, $(0.C48)_{16}$

= $(110001001000)_2$.

How to convert hexadecimal to octal?

Using the below two methods, we can convert the hexadecimal number system into the octal number system.

1. Convert the hexadecimal number into **binary** and then convert the binary into octal.

2. Convert the hexadecimal number into **decimal** and then convert the decimal into octal.

Let's convert the hexadecimal number into the octal number system.

Hexadecimal to Binary to Octal

Let's convert $(ff)_{16}$ into Octal.

Step 1: Convert $(ff)_{16}$ into Binary

In order to convert the hexadecimal number into binary, we need to express every hexadecimal value using 4 binary bits.

Binary equivalent of **f** is **$(1111)_2$**

= $(ff)_{16}$

= (1111)(1111)

= $(11111111)_2$

Step 2 : Convert $(11111111)_2$ into Octal

In order to convert the binary number into octal, we need to group every 3 binary bits and calculate the value[From left to right].

$(11111111)_2$ in Octal

= $(11111111)_2$

= (11)(111)(111)

= $(377)_8$

Hexadecimal Decimal Octal

Step 1: Convert $(ff)_{16}$ into Decimal

f equivalent decimal is 15.

= $15*16^1+15*16^0$

= 240+15

= $(255)_{10}$

Step 2 : Convert $(255)_{10}$ into Octal

```
8|255
 ----
8|31 7
 ----
8|37
```

= $(377)_8$

BCD or Binary Coded Decimal

Binary Coded Decimal, or **BCD**, is another process for converting decimal numbers into their binary equivalents.

- It is a form of binary encoding where each digit in a decimal number is represented in the form of bits.
- This encoding can be done in either 4-bit or 8-bit (usually 4-bit is preferred).
- It is a fast and efficient system that converts the decimal numbers into binary numbers as compared to the existing binary system.
- These are generally used in digital displays where is the manipulation of data is quite a task.
- Thus BCD plays an important role here because the manipulation is done treating each digit as a separate single sub-circuit.

The BCD equivalent of a decimal number is written by replacing each decimal digit in the integer and fractional parts with its four bit binary equivalent.the BCD code is more precisely known as 8421 BCD code , with 8,4,2 and 1 representing the weights of different bits in the four-bit groups, Starting from MSB and proceeding towards LSB. This feature makes it a weighted code , which means that each bit in the four bit group representing a given decimal digit has an assigned weight.
Many decimal values have an infinite place-value representation in binary but have a finite place-value in binary-coded decimal. For example, 0.2 in binary is .001100… and in BCD is 0.0010. It avoids fractional errors and is also used in huge financial calculations.
Consider the following truth table and focus on how are these represented.

Truth Table for Binary Coded Decimal

DECIMAL NUMBER	**BCD**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

In the **BCD numbering system**, the given decimal number is segregated into chunks of four bits for each decimal digit within the number. Each decimal digit is converted into its direct binary form (usually represented in 4-bits).

For example:

1. Convert (123)10 in BCD

From the truth table above,
1 -> 0001
2 -> 0010
3 -> 0011
thus, BCD becomes -> 0001 0010 0011

2. Convert (324)10 in BCD

(324)10 -> 0011 0010 0100 (BCD)

Again from the truth table above,
3 -> 0011
2 -> 0010
4 -> 0100
thus, BCD becomes -> 0011 0010 0100

This is how decimal numbers are converted to their equivalent BCDs.

- It is noticeable that the BCD is nothing more than a binary representation of each digit of a decimal number.
- It cannot be ignored that the BCD representation of the given decimal number uses extra bits, which makes it heavy-weighted.

Gray Code

Definition: Gray Code is the minimum-change code category of coding in which, the two consecutive values changes by only a single bit. More specifically we can say, it is a binary number system where while moving from one step to the next, only a single bit shows variation.

This coding technique was invented by **Frank Gray**, thus it is named so.

It is also termed as **reflected binary code** or **cyclic code**. It is an unweighted code, as here like other number systems, no particular weight is provided to various bit positions.

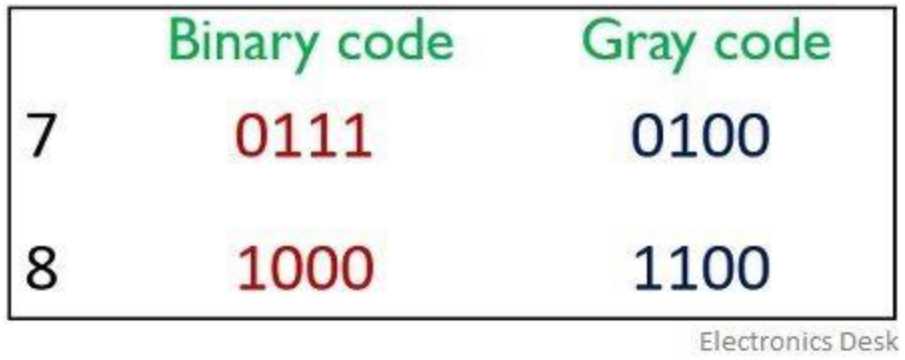
Basically, binary code is changed to gray equivalent in order to lessen the switching operations. As only a single bit is changed at a particular time duration this leads to a reduction in switching from one bit to another.

Let us have a look at the tabular representation, showing gray value for different binary values:

Decimal Value	**Binary Code**	**Gray Code**
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
A	1010	1111
B	1011	1110
C	1100	1010
D	1101	1011
E	1110	1001
F	1111	1000

Consider decimal value **7 and 8**, to understand the switching of bits.

We know in binary, 7 is written as 0111 while 8 is written as 1000. So, we can see that



	Binary code	Gray code
7	0111	0100
8	1000	1100

Electronics Desk

Thus we can conclude that in a binary system, all 4 bits are getting changed simultaneously. Hence we can say multiple bits are changing at the same time.

In gray code, 7 is written as 0100, as against 8 is written as 1100.

So, here we can see that only a single bit i.e., MSB is changing from 0 to 1 rest other bits are the same.

This shows that in binary coding, multiple bits are changing simultaneously, while in gray coding only a single bit is getting change at a time to move from one value to another.

Therefore, we can say switching is easy in gray code than in binary code.

Now, the question arises, how can we achieve, a gray equivalent from a binary code or vice versa. For this, we will separately discuss the conversion process using examples.

Binary to Gray Code Conversion

The conversion process from binary code to a gray involves the following steps:

1. Firstly, record the most significant bit or MSB or the leftmost bit of the given binary data as it is, to have MSB of gray equivalent.
2. Now, proceed towards adding the adjacent bits of the binary data starting from MSB with its adjacent bit to LSB. While adding, put the summation obtained in place of next bit and ignoring the carry.
3. Repeat the same process for all the bits in the sequence till LSB.

This is how the binary code is converted into gray equivalent.

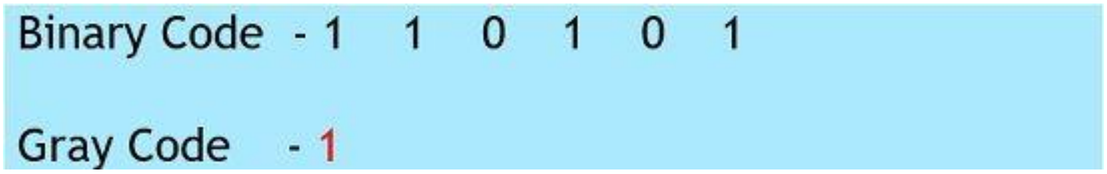
- Let us take some examples to understand the above-discussed steps clearly.

$(110101)_2$

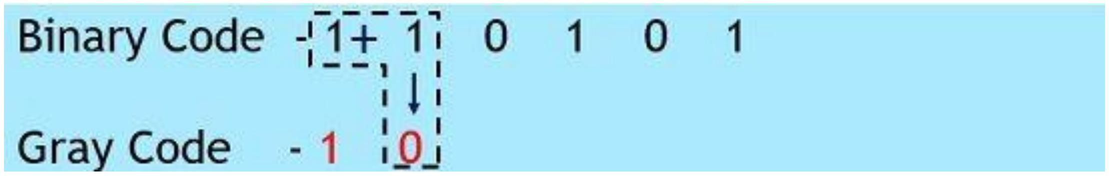
Suppose this is the binary value which is to be converted into equivalent gray value.

As we have discussed while mentioning the steps, that the first bit or MSB of the gray equivalent will be the same as the MSB of the binary value.

Thus

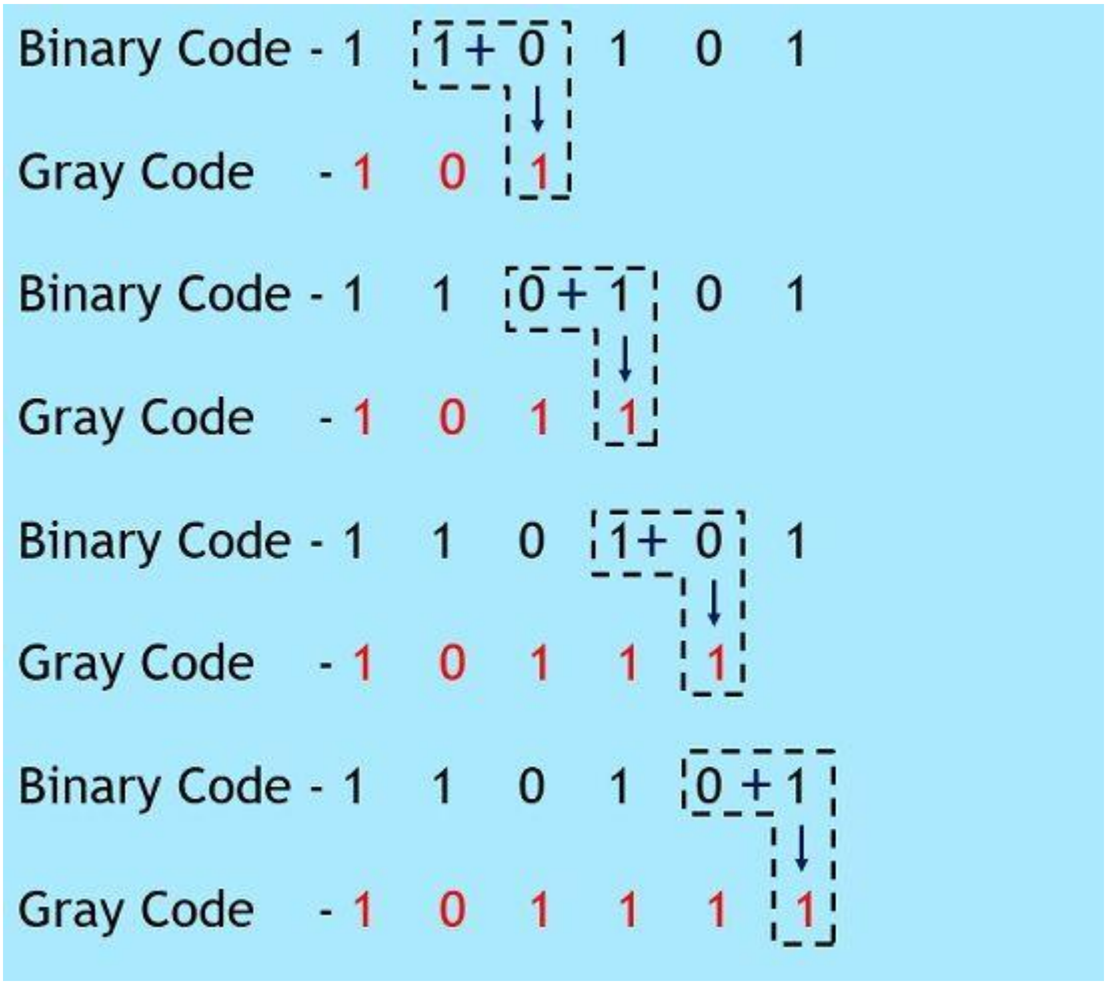


Now add the two adjacent bits starting from MSB to LSB and writing the result obtained as the next bit.



We know the addition of binary 1 and 1 will give 0 as the sum and 1 as the carry. And we have already discussed that the carry bit must be ignored, while the sum bit achieved will be put as the next bit in the gray value.

Further, repeating the same process,



Binary Code - 1 1+ 0 1 0 1
Gray Code - 1 0 1
Binary Code - 1 1 0 + 1 0 1
Gray Code - 1 0 1 1
Binary Code - 1 1 0 1+ 0 1
Gray Code - 1 0 1 1 1
Binary Code - 1 1 0 1 0 + 1
Gray Code - 1 0 1 1 1 1

So, the equivalent gray value for the given binary code is given as

(101111)$_{\text{Gray}}$

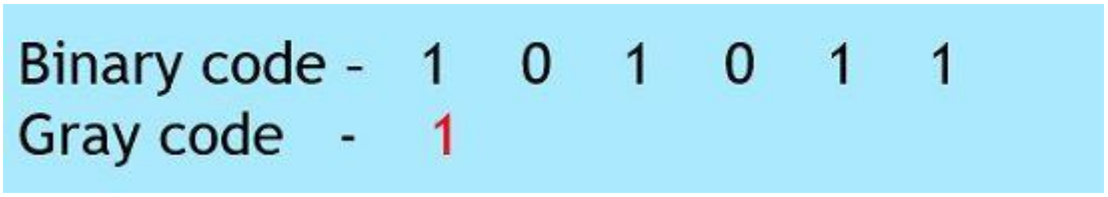
- Let us take another example to have a better understanding of the same.

(101011)$_2$

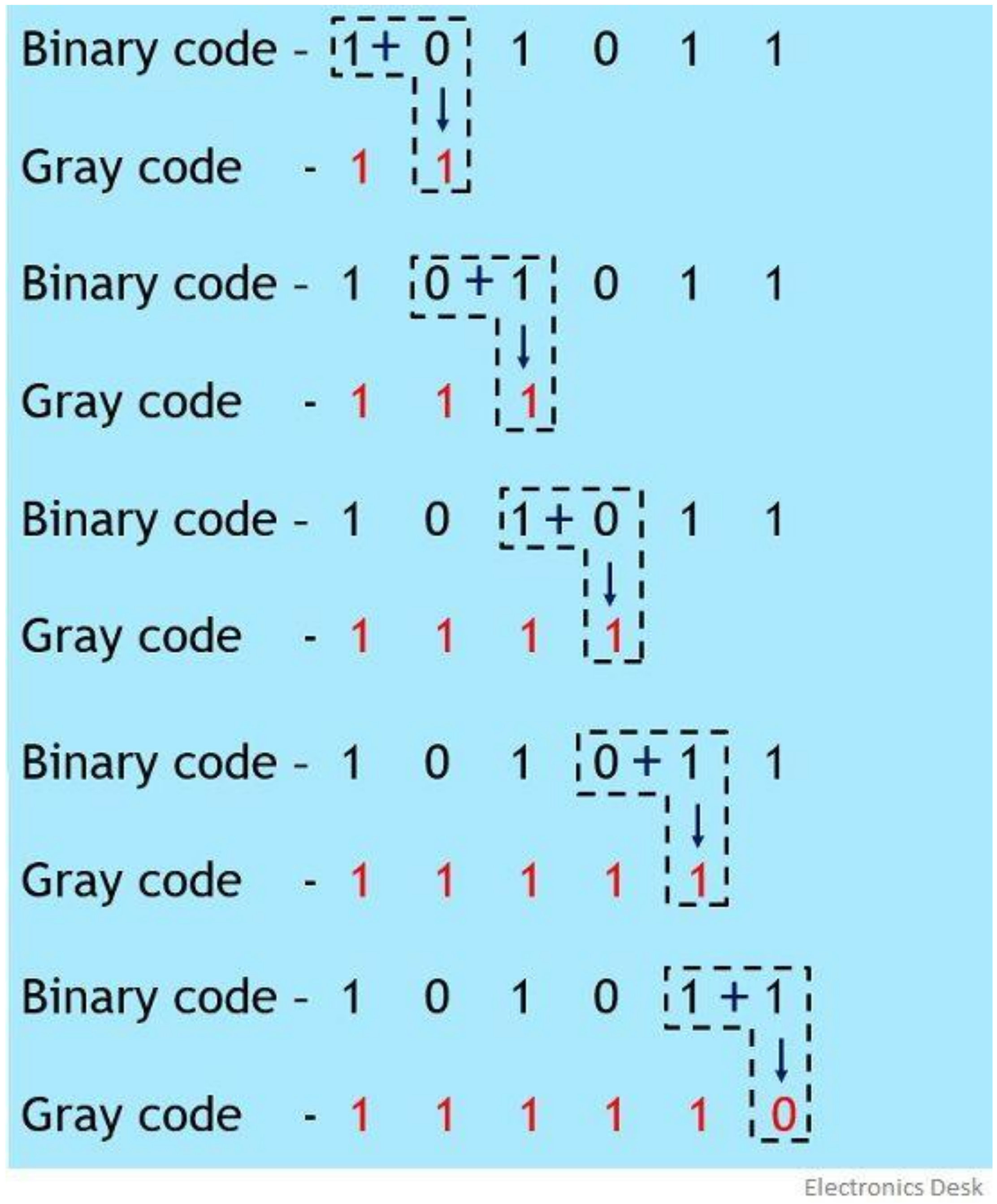
Consider the above given binary value which is to be converted into gray equivalent.

So, first, we will write the MSB of the binary digit as the MSB for the gray equivalent.

Thus



Now start adding the adjacent terms of the binary code in order to get the gray code



Binary code - 1 + 0 1 0 1 1
Gray code - 1 1
Binary code - 1 0 + 1 0 1 1
Gray code - 1 1 1
Binary code - 1 0 1 + 0 1 1
Gray code - 1 1 1 1
Binary code - 1 0 1 0 + 1 1
Gray code - 1 1 1 1 1
Binary code - 1 0 1 0 1 + 1
Gray code - 1 1 1 1 1 0
Electronics Desk

Hence the gray code for the above-given binary code will be

$(111110)_{Gray}$

So, in this way a binary code is converted into gray equivalent.

Let us now understand how gray code is converted into binary code.

Gray to Binary Code Conversion

The steps given below are required to be followed in order to convert a given gray value into its binary equivalent:

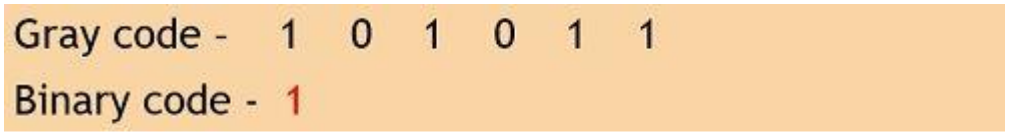
1. Like in case of binary to gray conversion, here also while writing binary code from gray code, the MSB must remain the same. So, write the leftmost bit of gray code as the MSB of binary code.
2. Now, add the recently achieved binary digit with the next adjacent gray code bit. The sum must be written as the next bit of binary equivalent, while the carry must be neglected.
3. The above-discussed step must be followed for all the bits present in the sequence.

In this way, a gray code is converted into the binary equivalent.

- Let us proceed towards some examples:

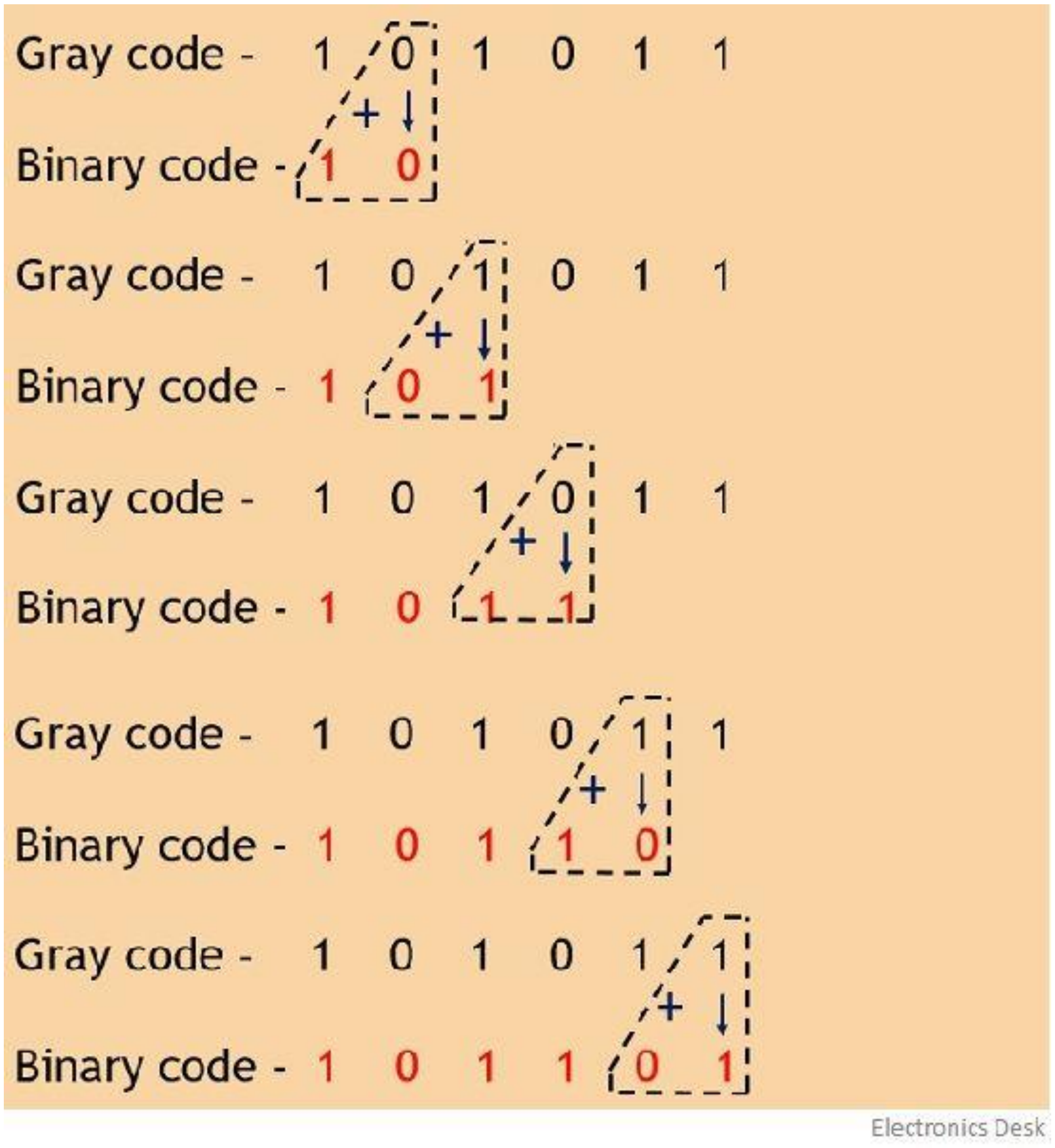
(101011)$_{\text{Gray}}$

So, in the first step, writing the MSB of gray value as the MSB of binary equivalent.



Gray code -	1	0	1	0	1	1
Binary code -	1					

Now on adding the achieved binary bit and the next adjacent gray bit till the LSB of the sequence,

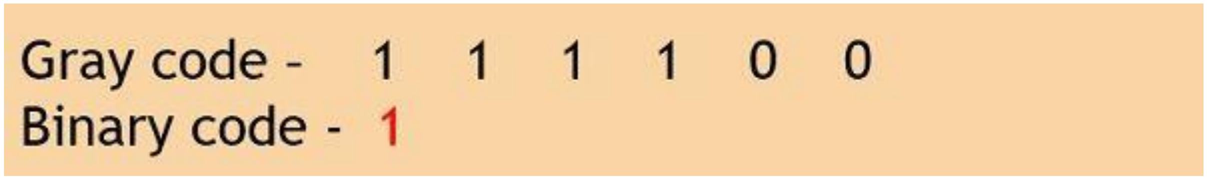


$(101101)_2$

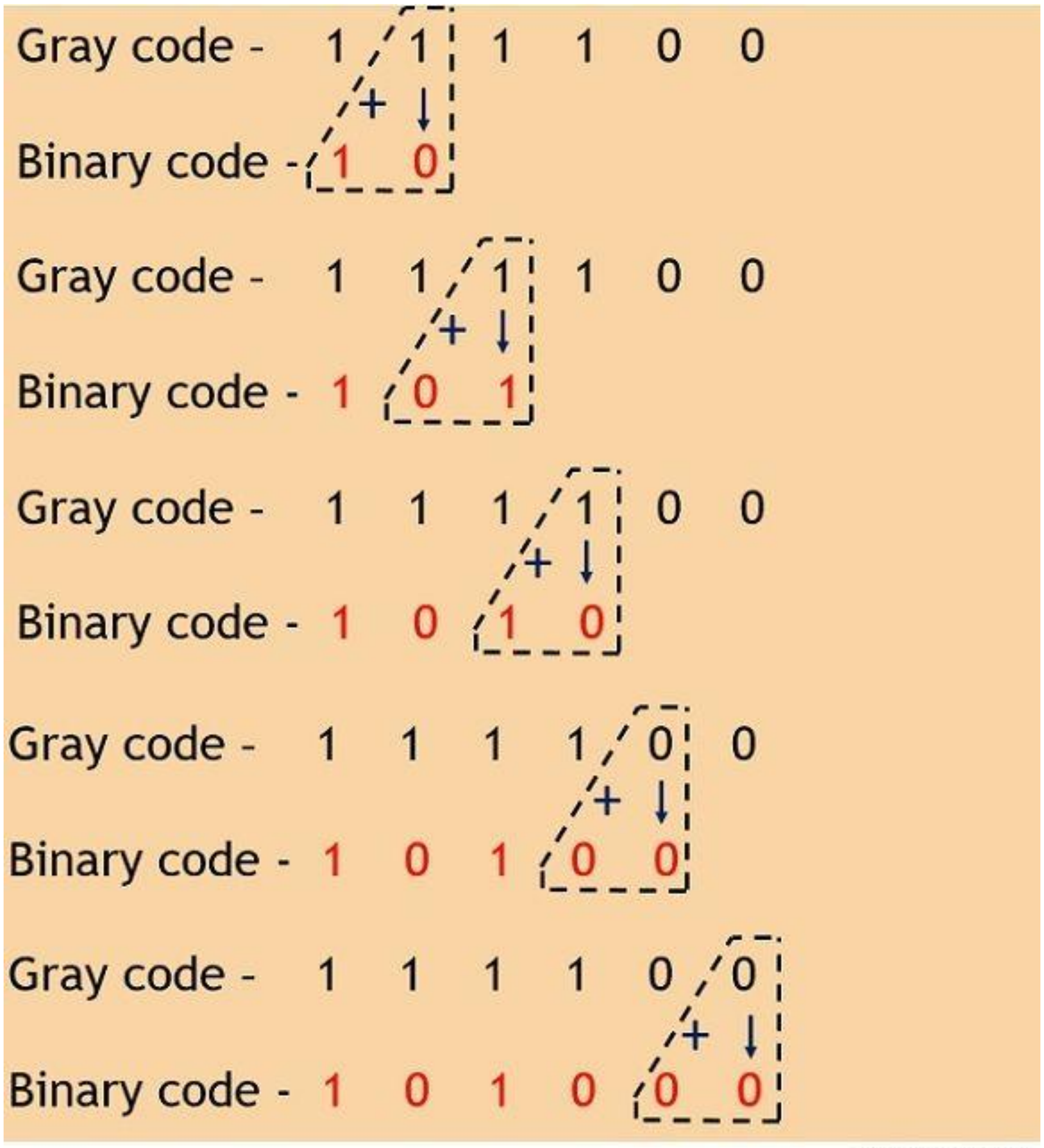
- Now have a look at one more example,

$(111100)_{Gray}$

Again writing the first bit of binary equivalent as the first digit of the gray code



Now on adding the binary code with the gray value, we will get,



Electronics Desk

Thus the equivalent binary code will be

$(101000)_2$

Applications of Gray Code

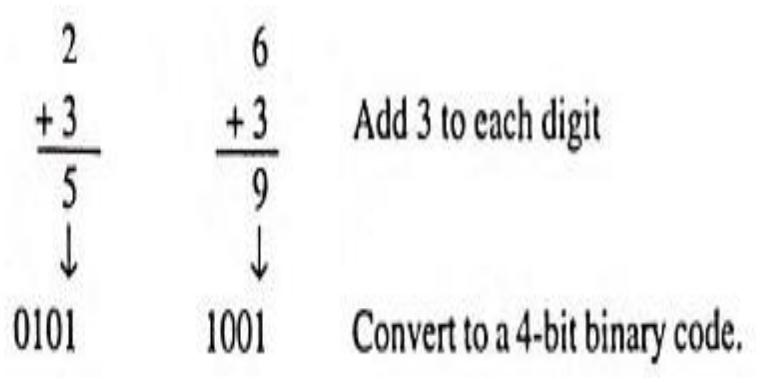
- Due to switching of a single bit, error correction can be easily achieved. Thus used in digital communication schemes such as cable TV etc.
- Also finds applications in shaft encoders, as in this possibility of errors increases with variation in the number of bits.

In this way binary to gray code and gray to binary code conversion is performed.

What is Excess-3 Code? (Definition and Examples)

The excess-3 code, abbreviated as XS-3, is an important 4-bit code sometimes used with binary-coded decimal (BCD) numbers. It possesses advantages in certain arithmetic operations.

The excess-3 code for a decimal number can be obtained in the same manner as BCD except that 3 is added to each decimal digit before encoding it in binary. For example, to encode the decimal digit 5 into excess-3 code, we must first add 3 to obtain 8. The digit 8 is encoded in its equivalent 4-bit binary code 1000. As another example, let us convert 26 into its excess-3 code representation.



Since no definite weights can be assigned to the four digit positions, excess-3 is an **unweighted code**. Excess-3 codes for decimal digits 0 through 9 are given in Table 44.7. The noteworthy point from the Table 44.6 is that both codes (BCD and excess-3) use only 10 of the 16 possible 4-bit code groups. The excess-3 codes, however, does not use the same code groups. For excess-3 codes, the invalid code groups are 0000, 0001, 0010, 1101, 1110 and 1111.

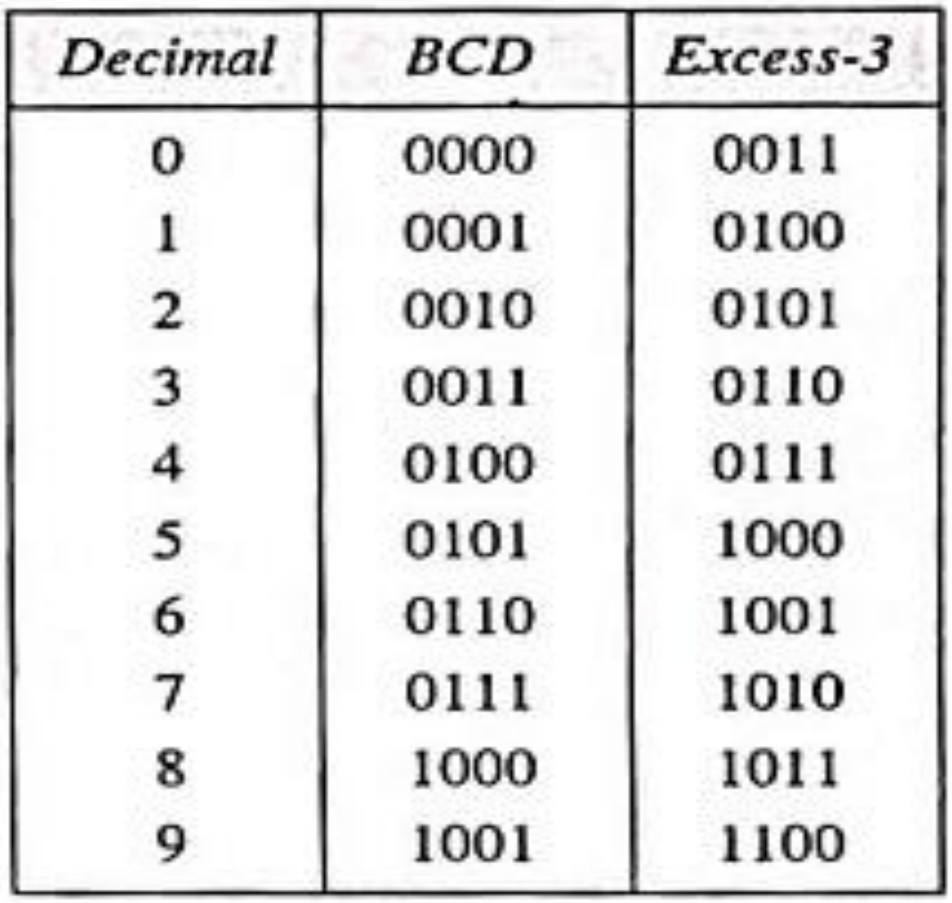


TABLE 44.7 Excess-3 Code

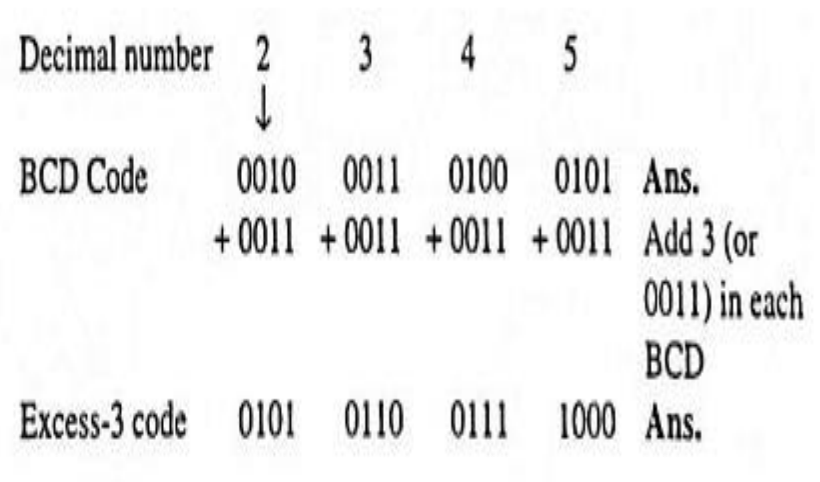
Decimal	*BCD*	*Excess-3*
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

The key feature of the excess-3 codes is that it is self-complementing code. It means that 1’s complement of the coded number yields 9’s complement of the number itself. For example, excess-3 code of decimal 5 is 1000, its 1’s complement is 0111, which is excess-3 code for decimal 4, which is 9’s complement of 5.

It should be noted that the 1’s complement is easily produced with digital logic circuits by simply inverting each bit. The self-complementing property makes the excess-3 code useful in some arithmetic operations, because subtraction can be performed using the 9’s complement method.

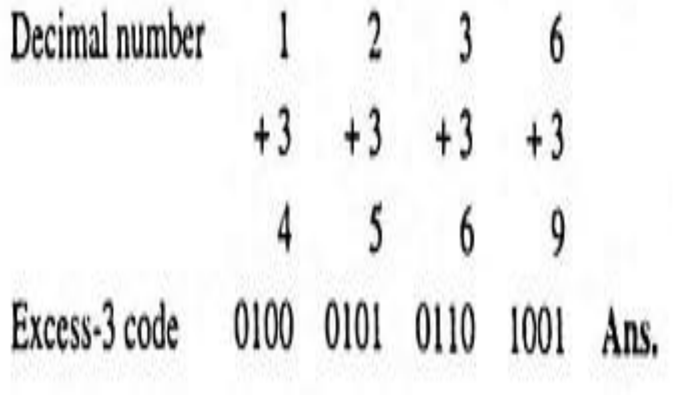
Example 1: Encode the decimal number 2345 in BCD and excess-3 codes.

Solution:



Decimal number	2	3	4	5	
	↓				
BCD Code	0010	0011	0100	0101	Ans.
	+ 0011	+ 0011	+ 0011	+ 0011	Add 3 (or 0011) in each BCD
Excess-3 code	0101	0110	0111	1000	Ans.

Example 2: Encode $(1236)_{10}$ In excess-3 code.
Solution:



Decimal number	1	2	3	6	
	+3	+3	+3	+3	
	4	5	6	9	
Excess-3 code	0100	0101	0110	1001	Ans.

Error Detection

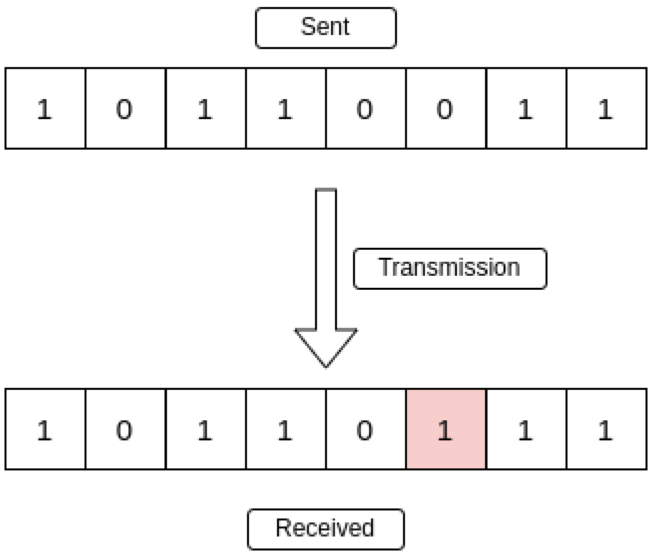
Error is a condition when the receiver's information does not match the sender's. Digital signals suffer from noise during transmission that can introduce errors in the binary bits traveling from sender to receiver. That means a 0 bit may change to 1 or a 1 bit may change to 0.

Data (Implemented either at the Data link layer or Transport Layer of the OSI Model) may get scrambled by noise or get corrupted whenever a message is transmitted. To prevent such errors, error-detection codes are added as extra data to digital messages. This helps in detecting any errors that may have occurred during message transmission.

Types of Errors

1. Single-Bit Error

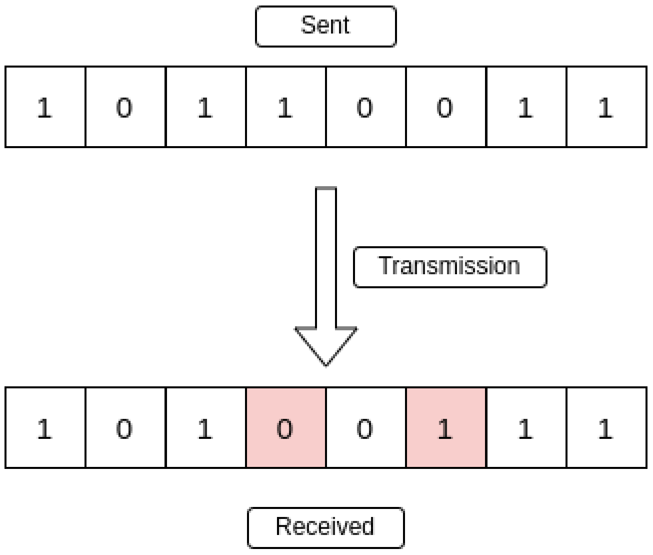
A single-bit error refers to a type of data transmission error that occurs when one bit (i.e., a single binary digit) of a transmitted data unit is altered during transmission, resulting in an incorrect or corrupted data unit.



Single-Bit Error

2. Multiple-Bit Error

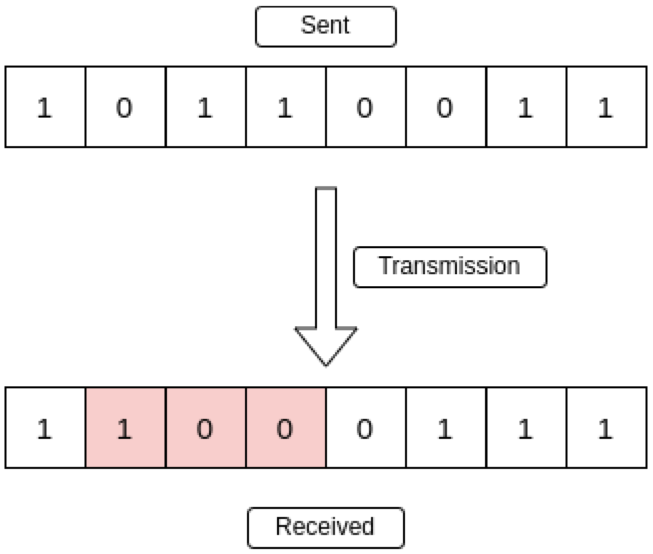
A multiple-bit error is an error type that arises when more than one bit in a data transmission is affected. Although multiple-bit errors are relatively rare when compared to single-bit errors, they can still occur, particularly in high-noise or high-interference digital environments.



Multiple-Bit Error

3. Burst Error

When several consecutive bits are flipped mistakenly in digital transmission, it creates a burst error. This error causes a sequence of consecutive incorrect values.



Burst Error

Error Correction

When the data is sent from the **sender side** to the receiver’s side it needs to be detected and corrected. So an error correction method is used for this purpose. Following are the two ways through which error correction can be handled:

Types of Error Correction

Here are the types of error correction in computer networks:

1. Backward Error Correction

The receiver detects an error and requests the sender to retransmit the entire data unit.

It is commonly used in applications where data integrity is critical and retransmission is feasible, such as file transfers.

2. Forward Error Correction (FEC)

The receiver corrects errors on its own using error-correcting codes, without needing retransmission. It is useful in real-time communications (e.g., video streaming, voice-over IP) where retransmission is impractical.

Here are the error correction techniques in computer networks:

1. Single-bit Error Detection

A single additional bit can detect errors but cannot correct them.

2. Hamming Code

It was developed by R.W. Hamming, it identifies and corrects single-bit errors by adding redundant bits.

3. Parity Bits

Parity bits are added to binary data to make the total number of 1s either even or odd.

Even Parity

- If the total number of 1s is even, the parity bit is set to 0.
- If the total number of 1s is odd, the parity bit is set to 1.

Odd Parity

- If the total number of 1s is even, the parity bit is set to 1.
- If the total number of 1s is odd, the parity bit is set to 0.

Comparison of Error Detection and Correction

Here is a detailed comparison of error detection and error correction

Error Detection	Error Correction
The purpose of error detection is to identify the presence of errors	The purpose of error correction is to correct the errors without retransmission
It is generally more efficient (lower overhead)	This can introduce higher overhead and complexity
It is much simpler to implement	It is more complex due to additional coding

Error Detection	Error Correction
	schemes
It has lower latency (only requires checking)	It contains higher latency (requires decoding and correction)
The error detection is used in networking (e.g., TCP, UDP)	The error correction is used in storage systems, error-prone environments (e.g., CDs, DVDs)
Examples of Error detection are Parity Check, CRC, Checksum	Examples of Error correction are Hamming Code, Reed-Solomon, Turbo Codes
This cannot fix errors, only detects them	It is limited to specific types and numbers of errors
It ensures data integrity during transmission	It ensures reliable data retrieval and storage

Advantages and Disadvantages of Error Detection and Error Correction

Here are the advantages and disadvantages of error detection and correction in computer networks:

Advantages of Error Detection

Here are the advantages of error detection in computer networks:

- Easier to implement with lower computational requirements.
- Faster processing since it only checks for errors rather than correcting them.
- Generally requires less additional data compared to error correction methods.
- Can identify errors quickly during data transmission.

Disadvantages of Error Detection

Here are the disadvantages of error detection in computer networks:

- Only detects errors but does not fix them, necessitating retransmission.
- May fail to detect certain types of errors, especially if multiple errors occur.
- Relies on the assumption that retransmission will resolve issues.

Advantages of Error Correction

Here are the advantages of error correction in computer networks:

- Can correct errors to improve data integrity and reliability.
- Reduces the need for retransmission, which is beneficial in bandwidth-limited environments.
- Provides a higher level of error resilience, especially in noisy environments.

Disadvantages of Error Correction

Here are the disadvantages of error correction in computer networks:

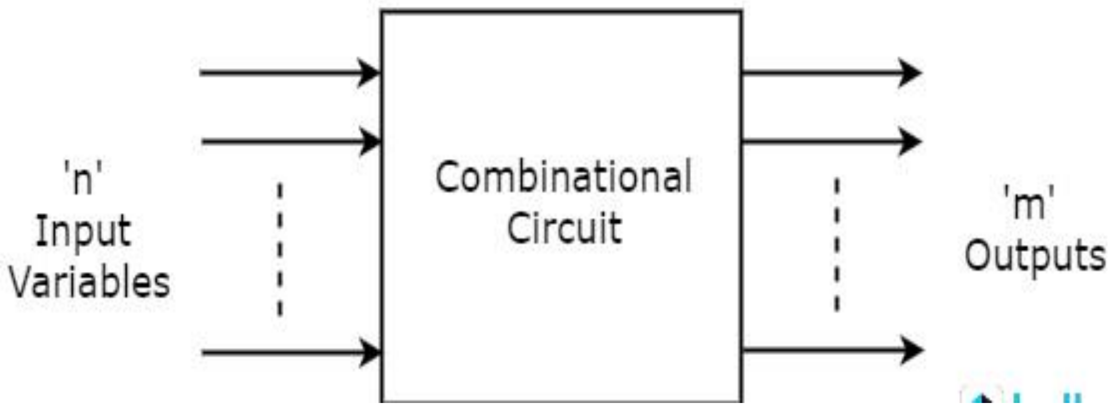
- More complex to implement, requiring advanced algorithms and coding schemes.
- Involves additional bits for correction, which can increase the overall data size.
- Increased processing time due to the need for decoding and correcting errors.
- Can only correct a predetermined number of errors, beyond which data integrity may be compromised.

Unit No 3

Combinational Circuits

Combinational Logic Circuits are built up of basic logic NAND, NOR or NOT gates that are linked or connected to compose more complicated switching circuits. These logic gates signify the building blocks of combinational logic circuits. Examples of combinational logic circuits are adders, subtractors, decoder, encoder, multiplexer, and demultiplexer.

In the combinational circuits for the X input binary variable, there are Y output variables. In these circuits for every possible combinational circuit, there is one possible output combination. So if one of the conditions of its inputs changes state, from 0-1 or 1-0, so too will the resulting output as combinational logic circuits by default have no memory/feedback loops within their design.



Addition is one of the most basic operations performed by different electronic devices like computers, calculators, etc. The electronic circuit that performs the addition of two or more numbers, more specifically binary numbers, is called as **adder**. Since, the logic circuits use binary number system to perform the operations, hence the adder is referred to as **binary adder**

Depending on the number of bits that the circuit can add, adders (or binary adders) are of two types −

- Half Adder
- Full Adder

What is a Half-Adder?

A combinational logic circuit which is designed to add two binary digits is called as a **half adder**. The half adder provides the output along with a carry value (if any). The half adder circuit is designed by connecting an EX-OR gate and one AND gate. It has two input terminals and two output terminals for sum and carry. The block diagram and circuit diagram of a half adder are shown in Figure-1.

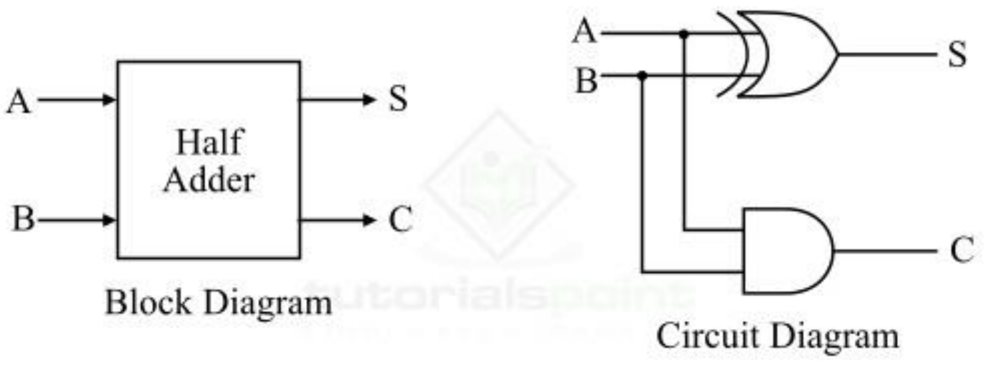
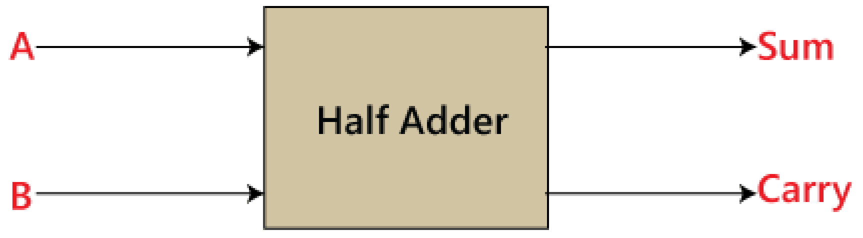


Figure 1 - Half Adder



From the logic circuit diagram of half adder, it is clear that A and B are the two input bits, S is the output sum, and C is the output carry bit.

In the case of a half adder, the output of the EX-OR gate is the sum of two bits and the output of the AND gate is the carry. Although, the carry obtained in one addition will not be forwarded in the next addition because of this it is known as half adder.

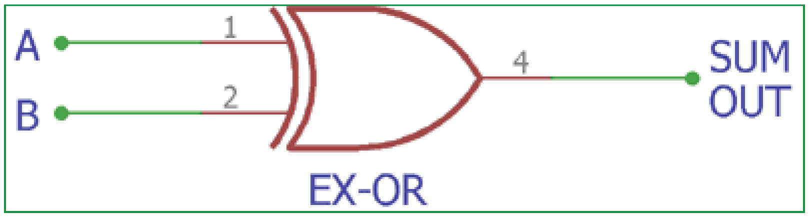
Construction of Half Adder Circuit:

We have seen the Block Diagram of Half Adder circuit above with two inputs A,B and two outputs- Sum, Carry Out. We can make this circuit using two basic gates

1. **2-input Exclusive-OR Gate or Ex-OR Gate**
2. **input AND Gate**

2-input Exclusive-OR Gate or Ex-OR Gate

The Ex-OR gate is used to produce the **SUM** bit and **AND** Gate produce the carry bit of the same input A and B.



This is the symbol of two inputs **EX-OR** gate. **A,** and **B** is the two binary input and **SUMOUT** is the final output after adding two numbers.

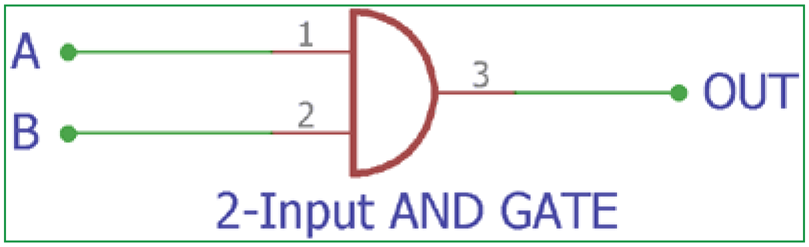
The truth table of EX-OR gate is –

Input A	*Input B*	*SUM OUT*
0	0	0
0	1	1
1	0	1
1	1	0

In the above table we can see the total sum output of the EX-OR gate. When any one of the bits **A** and **B** is **1** the output of the gate becomes **1**. On the two other cases when both inputs are **0** or **1** the Ex-OR gate produce **0** outputs.

2-input AND Gate:

X-OR gate only provides the sum and unable to provide carry bit on 1 + 1, we need another gate for Carry. **AND** gate is perfectly fits in this application.



This is the basic circuit of two input **AND** gate. Same as like **EX-OR** gate it has two **inputs**. If we provide **A** and **B** bit in the input it will produce an Output.

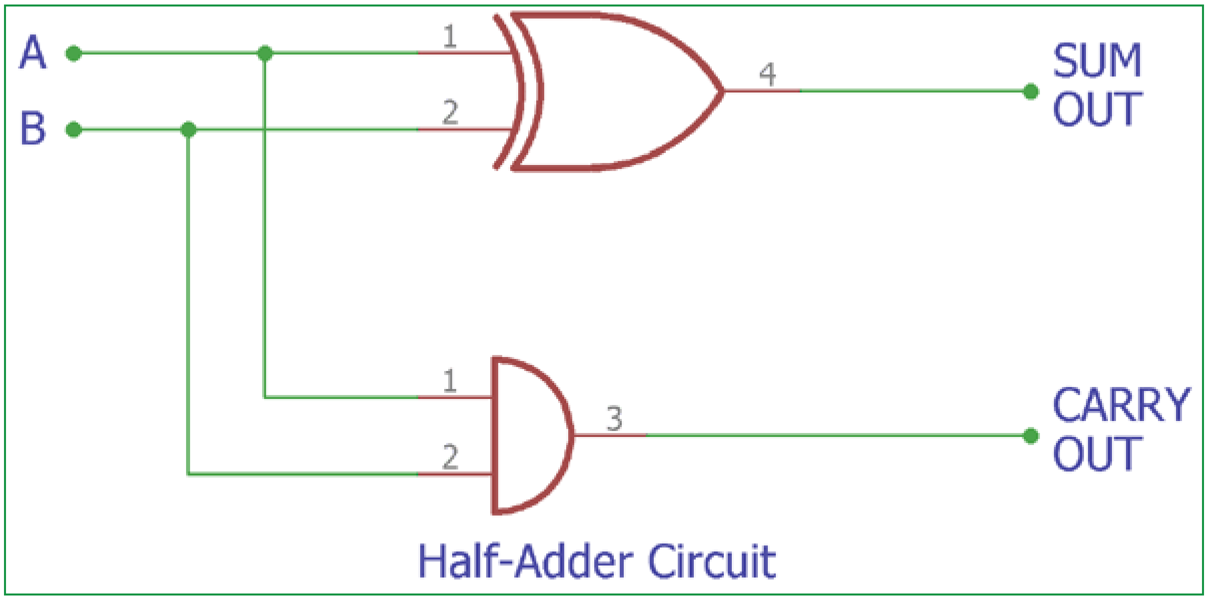
The output is depends on the **AND gate truth table-**

Input A	*Input B*	*Carry Output*
0	0	0
0	1	0
1	0	0
1	1	1

In the above, the truth table of AND gate is shown where it will only produce the output when both inputs are **1**, Otherwise it will not provide an output if both inputs are **0** or any of the inputs is **1**. Learn more about AND gate here.

Half-Adder logical circuit:

So the **Half-Adder logical circuit** can be made by combining this two gates and providing the same input in both gates.



This is the construction of Half-Adder circuit, as we **can** see two gates are combined and the same input A and B are provided in both gates and we get the **SUM output across EX-OR gate and the Carry Out bit across AND gate**.

The **Boolean expression of Half Adder circuit** is-

```
SUM = A XOR B (A+B)

CARRY = A AND B (A.B)
```

Operation of Half Adder

Half adder adds two binary digits according to the rules of binary addition. These rules are as follows −

0+0=00+0=0

0+1=10+1=1

1+0=11+0=1

1+1=10(Sum=0&Carry=1)1+1=10(Sum=0&Carry=1)

According to these rules of binary addition, we can see that the first three operations produce a sum whose length is one digit, whereas in the case of last operation (1 and 1), the sum consists of two digits. Here, the MSB (most significant bit) of this result is called a carry (which is 1) and the LSB (least significant bit) is called the sum (which is 0).

Truth Table of Half Adder

Truth table is one that gives the relationship between inputs and outputs of a logic circuit and explains the operation of the circuit. The following is the truth table of the half-adder −

Inputs		**Outputs**	
A	**B**	**S (Sum)**	**C (Carry)**
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

K-Map for Half Adder

We can use the K-Map (Karnaugh Map), a method for simplifying Boolean algebra, to determine equations of the sum bit (S) and the output carry bit (C) of the half adder circuit.

The k-map for half adder circuit is shown in Figure-2.

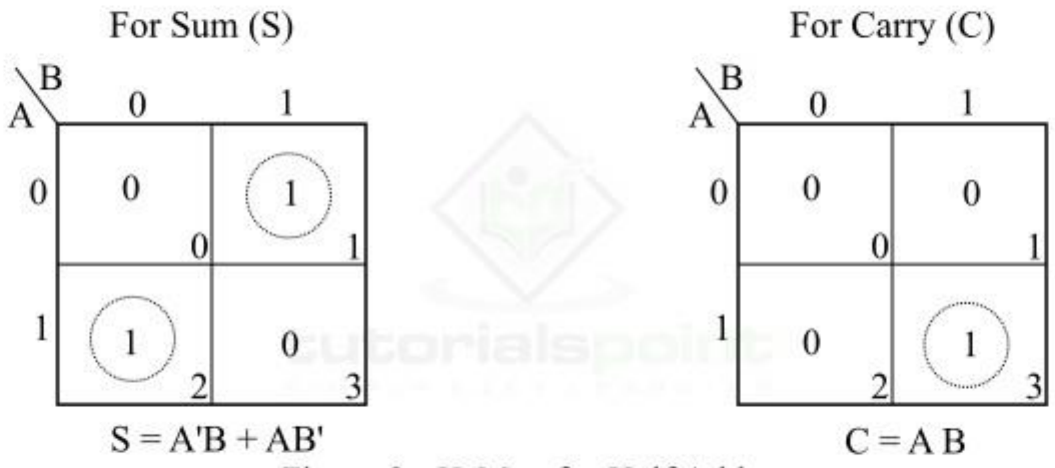


Figure 2 - K-Map for Half Adder

Characteristic Equations of Half-Adder

The characteristic equations of half adder, i.e., equations of sum (S) and carry (C) are obtained according to the rules of binary addition. These equations are given below −

The sum (S) of the half-adder is the XOR of A and B. Thus,

$$\text{Sum}, S=A \oplus B=AB'+A'B\text{Sum}, S=A \oplus B=AB'+A'B$$

The carry (C) of the half-adder is the AND of A and B. Therefore,

$$\text{Carry}, C=A \cdot B\text{Carry}, C=A \cdot B$$

Applications of Half Adder

The following are some important applications of half adder −

- Half adder is used in ALU (Arithmetic Logic Unit) of computer processors to add binary bits.
- Half adder is used to realize full adder circuit.
- Half adder is used in calculators.
- Half adder is used to calculate addresses and tables.
- A Half Adder does not consider any previous carry input, which is why it is called a "half" adder. To add multiple-bit numbers, a Full Adder is required.

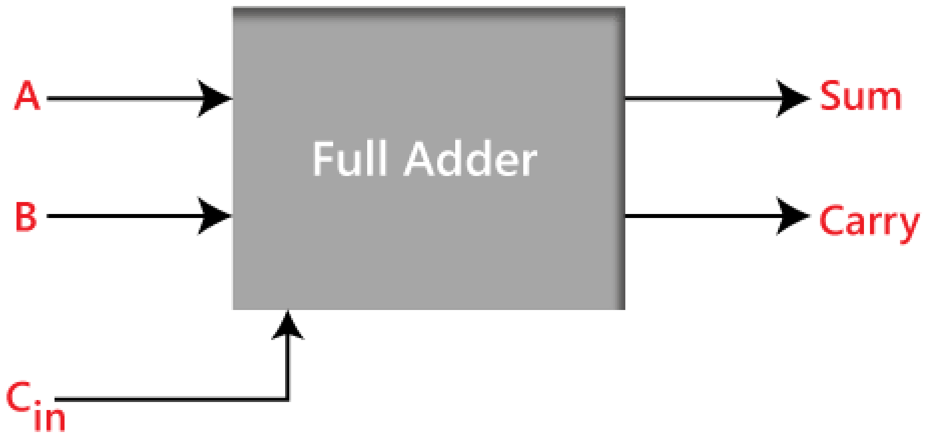
Conclusion

From the above discussion, we can conclude that half adders are one of the basic arithmetic circuits used in different electronic devices to perform addition of two binary digits. The major drawback of a half adder is that it cannot add the carry obtained from the addition of the previous stage. To overcome this drawback, full adders are used in electronic systems.

What is a Full Adder?

A combinational logic circuit that can add two binary digits (bits) and a carry bit, and produces a sum bit and a carry bit as output is known as a **full-adder**.

In other words, a combinational circuit which is designed to add three binary digits and produces two outputs (sum and carry) is known as a full adder. Thus, a full adder circuit adds three binary digits, where two are the inputs and one is the carry forwarded from the previous addition. The block diagram and circuit diagram of the full adder are shown in Figure-1.



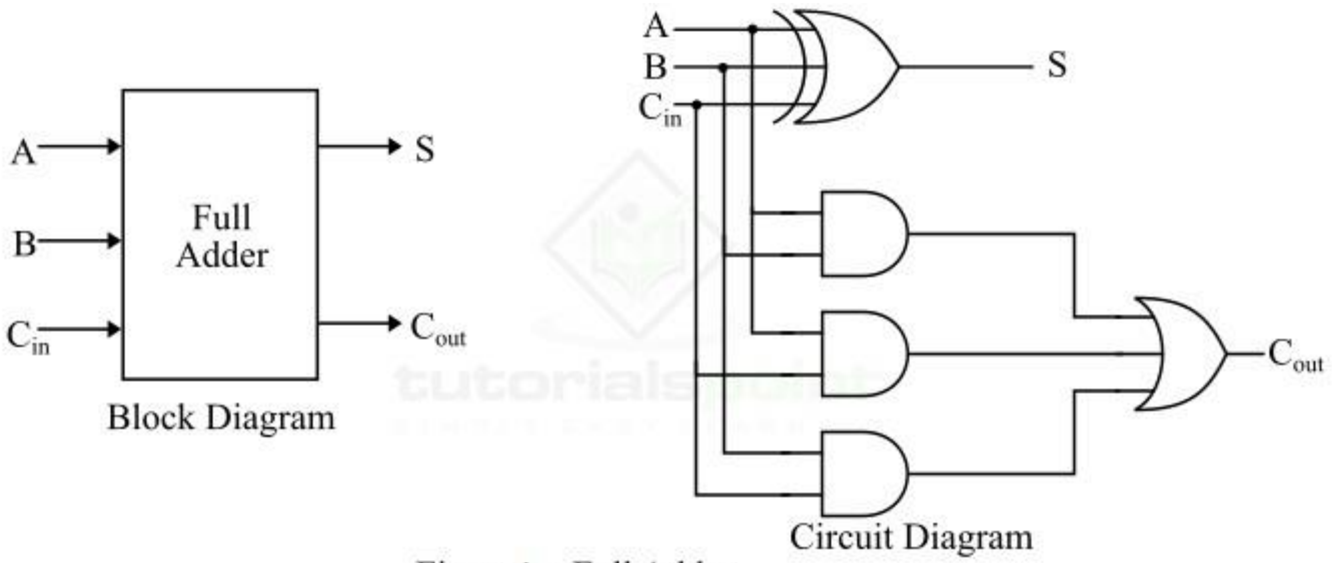


Figure 1 - Full Adder

Hence, the circuit of the full adder consists of one EX-OR gate, three AND gates and one OR gate, which are connected together as shown in the full adder circuit in Figure-1.

Operation of Full Adder

Full adder takes three inputs namely A, B, and C_{in}. Where, A and B are the two binary digits, and C_{in} is the carry bit from the previous stage of binary addition. The sum output of the full adder is obtained by XORing the bits A, B, and C_{in}. While the carry output bit (C_{out}) is obtained using AND and OR operations.

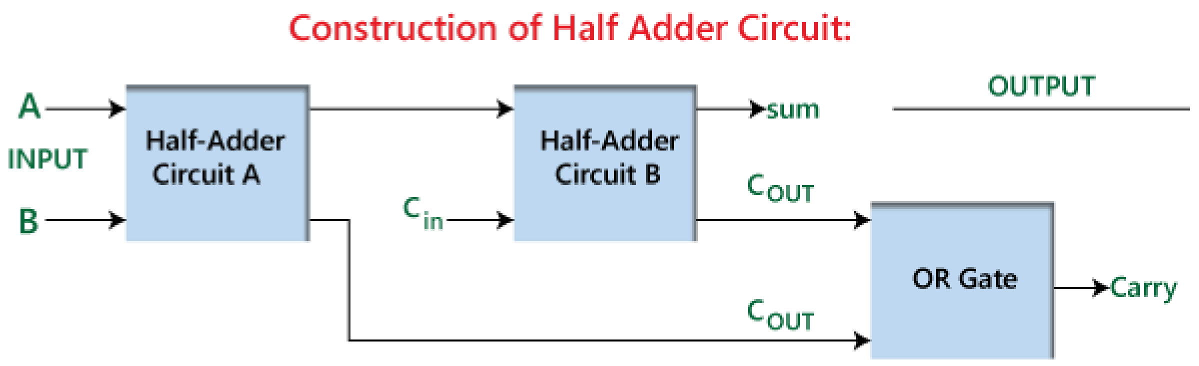
Truth Table of Full Adder

Truth table is one that indicates the relationship between input and output variables of a logic circuit and explains the operation of the logic circuit. The following is the truth table of the full-adder circuit –

Inputs			Outputs	
A	**B**	**C_{in}**	**S (Sum)**	**C_{out} (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

Hence, from the truth table, it is clear that the sum output of the full adder is equal to 1 when only 1 input is equal to 1 or when all the inputs are equal to 1. While the carry output has a carry of 1 if two or three inputs are equal to 1.

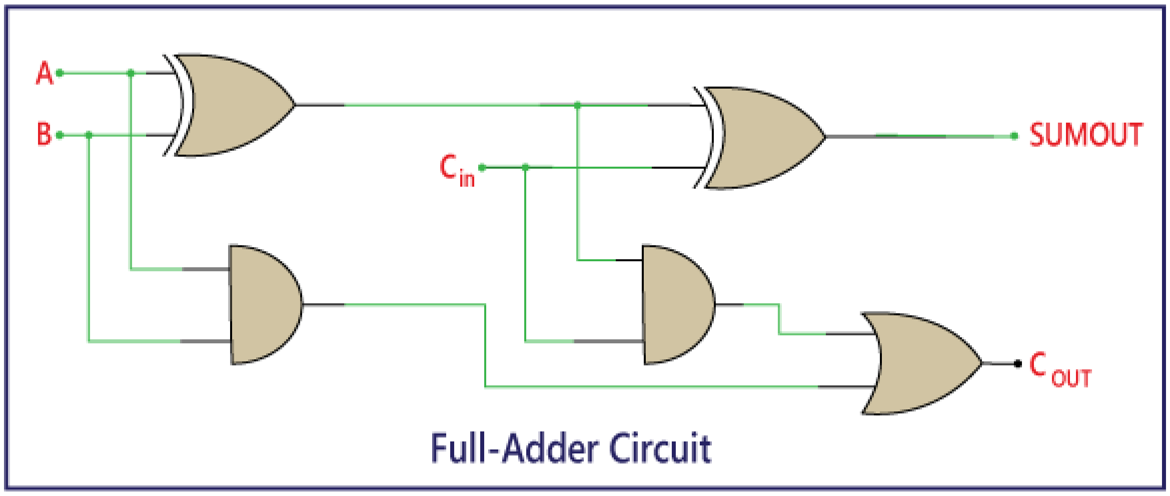
Construction of Half Adder Circuit:



The above block diagram describes the construction of the Full adder circuit. In the above circuit, there are two half adder circuits that are combined using the OR gate. The first half adder has two single-bit binary inputs A and B. As we know that, the half adder produces two outputs, i.e., Sum and Carry. The 'Sum' output of the first adder will be the first input of the second half adder, and the 'Carry' output of the first adder will be the second input of the second half adder. The second half adder will again provide 'Sum' and 'Carry'. The final outcome of the Full adder circuit is the 'Sum' bit. In order to find the final output of the 'Carry', we provide the 'Carry' output of the first and the second adder into the OR gate. The outcome of the OR gate will be the final carry out of the full adder circuit.

The MSB is represented by the final 'Carry' bit.

The full adder logic circuit can be constructed using the **'AND'** and **the 'XOR' gate** with an **OR gate**.



The actual logic circuit of the full adder is shown in the above diagram. The full adder circuit construction can also be represented in a Boolean expression.

Sum:

- Perform the XOR operation of input A and B.
- Perform the XOR operation of the outcome with carry. So, the sum is (A XOR B) XOR Cin which is also represented as:
- $(A \oplus B) \oplus Cin$

Carry:

1. Perform the 'AND' operation of input A and B.
2. Perform the 'XOR' operation of input A and B.
3. Perform the 'OR' operations of both the outputs that come from the previous two steps. So the 'Carry' can be represented as $A.B + (A \oplus B)$

K-Map for Full Adder

K-Map (Karnaugh Map) is a tool for simplifying binary complex Boolean algebraic expressions. The K-Map for full adder is shown in Figure-2.

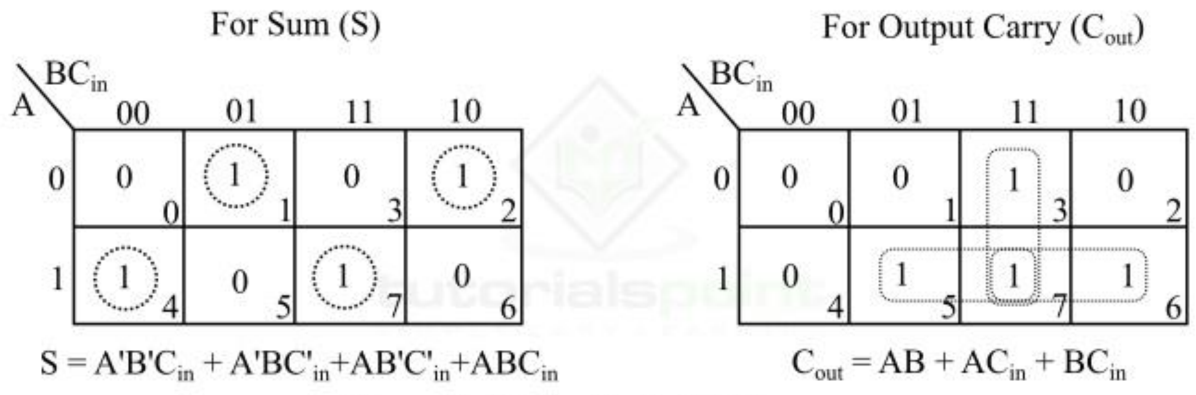

For Sum (S)
For Output Carry (C_{out})
BC_{in}
A
$S = A'B'C_{in} + A'BC'_{in} + AB'C'_{in} + ABC_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

Figure 2 - K Map for Full Adder

Advantages of Full Adder

The following are the important advantages of full adder over half adder −

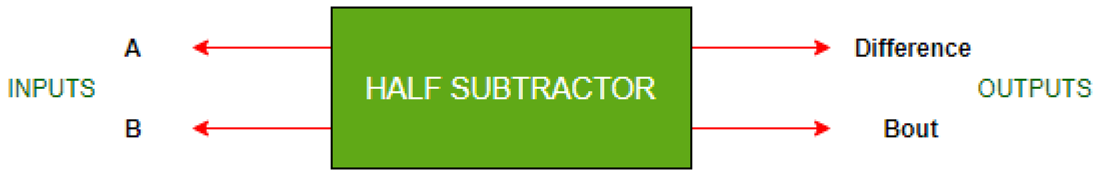
- Full adder provides facility to add the carry from the previous stage.
- The power consumed by the full adder is relatively less as compared to half adder.
- Full adder can be easily converted into a half subtractor just by adding a NOT gate in the circuit.
- Full adder produces higher output that half adder.
- Full adder is one of the essential part of critic digital circuits like multiplexers.
- Full adder performs operation at higher speed.

Applications of Full Adder

The following are the important applications of full adder −

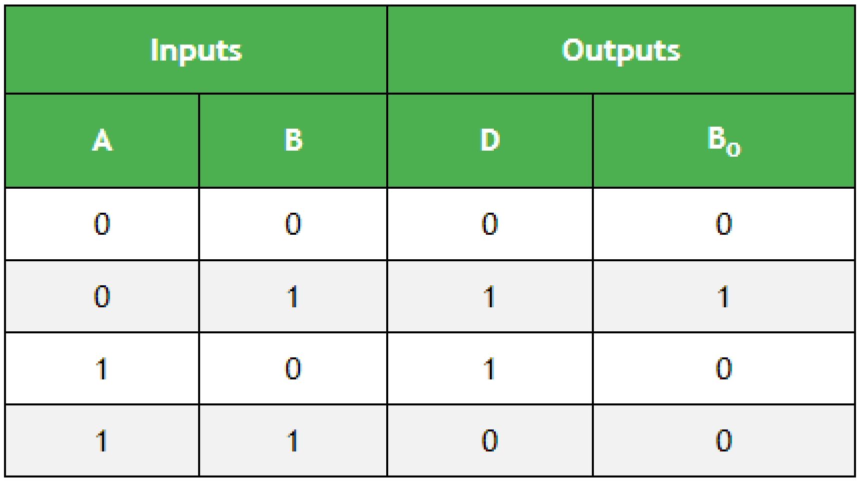
- Full adders are used in ALUs (arithmetic logic units) of CPUs of computers.
- Full adders are used in calculators.
- Full adders also help in carrying out multiplication of binary numbers.
- Full adders are also used to realize critic digital circuits like multiplexers.
- Full adders are used to generate memory addresses.
- Full adders are also used in generation of program counterpoints.
- Full adders are also used in GPU (Graphical Processing Unit).

. Half Subtractor:



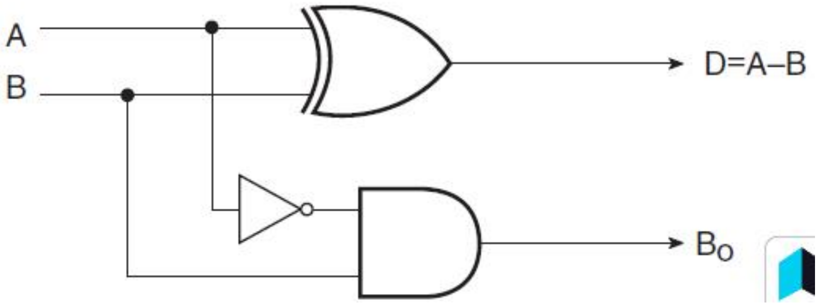
- It is a combinational logic circuit designed to perform the subtraction of two single bits.
- It contains two inputs (A and B) and produces two outputs (Difference and Borrow-output).
- Half subtractor is a combinational logic circuit intended to perform the subtraction of two single bits and generate the output. A subtractor circuit with two input variables as A and B displays two outputs i.e Difference and Borrow. The block diagram of a Half subtractor is as shown below:

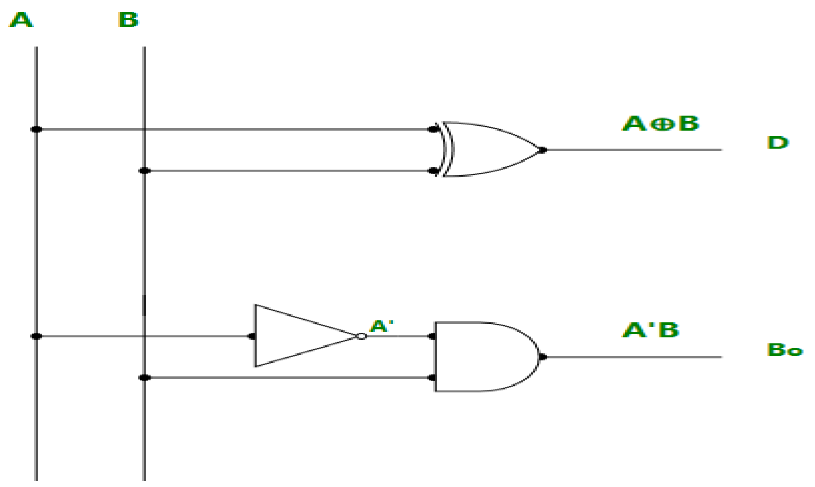
Truth Table of Half Subtractor:



Inputs		Outputs	
A	B	D	B_o
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

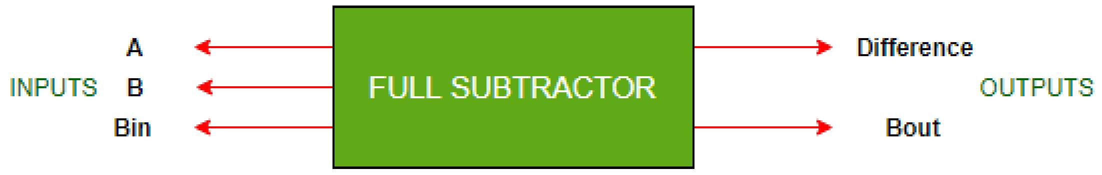
Logic Diagram of Half Subtractor:





Full Subtractor:

A full subtractor is again a combinational circuit that delivers subtraction of two bits, one is minuend and the other is subtrahend, taking into account the borrow of the earlier adjacent lower minuend bit. The block diagram of a full subtractor is as shown below:



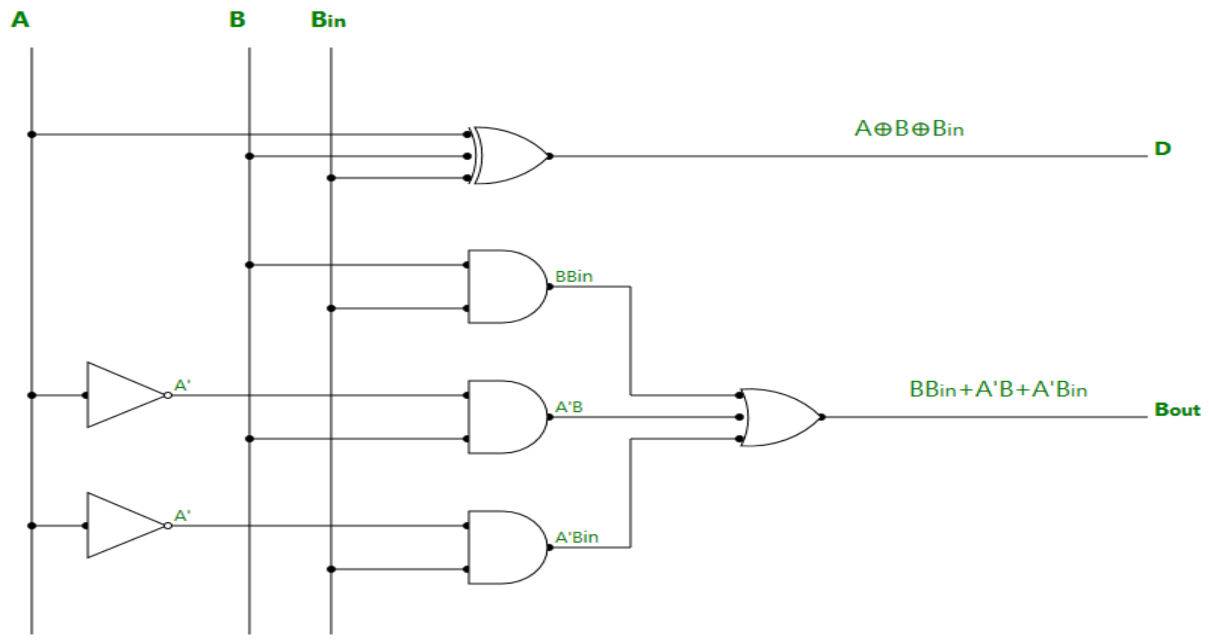
- It is a Combinational logic circuit designed to perform subtraction of three single bits.
- It contains three inputs(A, B, B_{in}) and produces two outputs (D, B_{out}).
- Where, A and B are called **Minuend** and **Subtrahend** bits.
- And, B_{in} -> Borrow-In and B_{out} -> Borrow-Out
-

Truth Table of Full Subtractor:

The full subtractor circuit includes three input variables and two output variables. The three inputs; Consider as A, B and Bin. The two outputs, D and Bout, outline the difference and output borrow, respectively. The full subtractor truth table is as shown:

Inputs			Outputs	
A	B	B_{in}	D	B_{out}
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

Logic Diagram of Full Subtractor:



Decoder-

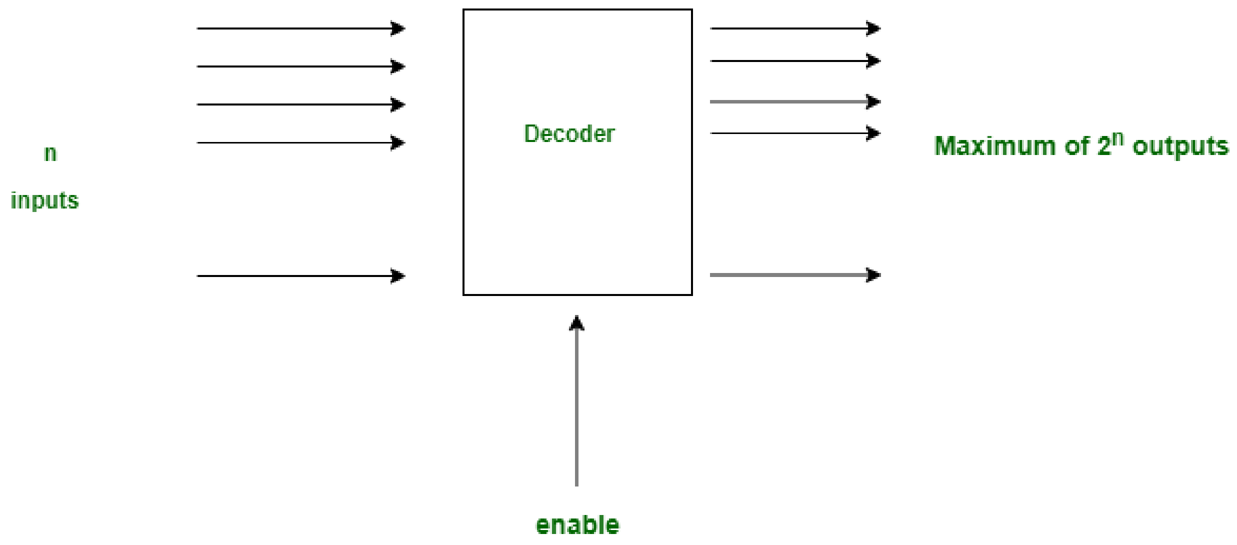
Following are the definitions of Decoder in Computer Architecture.

1. In computer architecture, a decoder is a combinational circuit that converts binary-coded inputs into a specific output. It takes an n-bit input and activates one of its 2^n output lines, making it useful for selecting components, memory addressing, or instruction decoding.

2. A decoder in digital electronics is a combinational circuit that converts binary input data into a unique output signal. Decoders play an essential role in various digital systems, enabling efficient data management and communication by transforming encoded data into a readable format. With applications ranging from memory addressing to data routing, decoders are essential components in modern electronic devices. The combinational circuit that converts binary information into 2^N output lines is known as a decoder. This binary information is input through N lines. The output lines represent a 2^N-bit code corresponding to the binary information.

3. In simple terms, a decoder performs the reverse function of an encoder. Typically, only one input line is activated at a time for simplicity, and the resulting 2^N-bit output code is equivalent to the original binary information.

4. The combinational circuit that change the binary information into 2N output lines is known as Decoders. The binary information is passed in the form of N input lines. The output lines define the 2N-bit code for the binary information. In simple words, the Decoder performs the reverse operation of the Encoder. At a time, only one input line is activated for simplicity. The produced 2N-bit output code is equivalent to the binary information.



Key Features of a Decoder

- Converts binary input data into distinct output signals.
- Activates one output line at a time based on the input binary value.
- Supports a range of applications from memory addressing to communication systems.
- Performs the opposite function of an encoder.

General Structure of a Decoder

- **Inputs:** Decoders typically take binary inputs, and the number of inputs determines the number of possible outputs.
- **Outputs:** The outputs represent a decoded signal, and only one output line is activated at any given time.
- **Enable Signal:** Many decoders have an enable signal, which ensures that the decoder is only active when necessary, avoiding unnecessary operation when disabled.

Types of Decoders

There are various types of decoders depending on the number of input and output lines.

The most common types include:

- 2 to 4 Decoder
- 3 to 8 Decoder
- 4 to 16 Decoder

Applications of Decoders

- **Memory Addressing:** Decoders are used in computers to select specific memory locations.
- **Multiplexing:** Decoders help in selecting data from multiple sources.
- **Control Units:** Used in control circuits to decode instruction signals for proper operation.

Encoder:

An encoder is a combinational circuit that is designed to perform the inverse operation of the decoder.

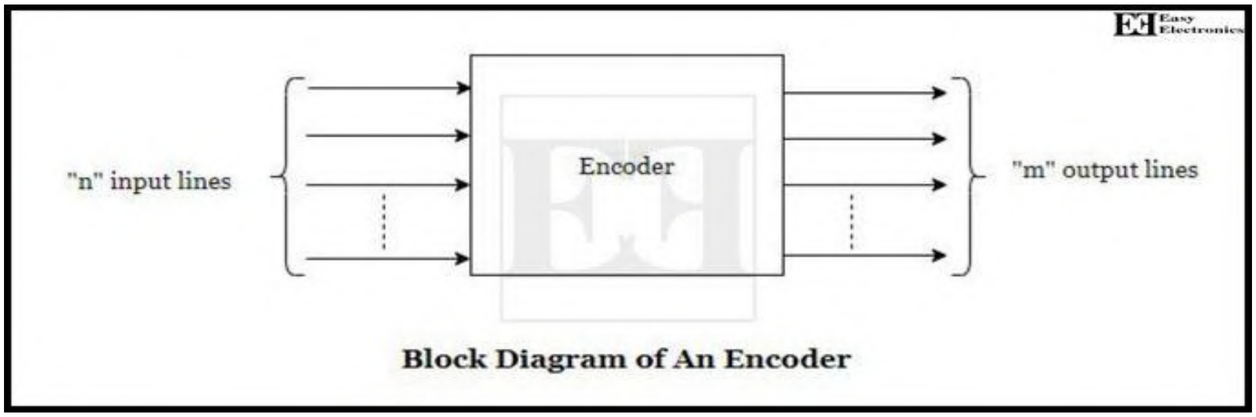
An encoder has “n” number of input lines and “m” number of output lines.

An encoder produces an m-bit binary code corresponding to the digital input number.

The encoder accepts an n-input digital word and converts it into an m-bit another digital word.

The internal combinational circuit of the encoder is designed accordingly.

The block diagram of the encoder is shown in the figure below.



Types of Encoder

The types of encoders that are going to be discussed in this lecture are as follows:

Types Of Encoder	
1.	Priority Encoder
2.	Decimal to BCD Encoder
3.	Octal to Binary Encoder
4.	Hexadecimal to Binary Encoder

Difference between Encoder and Decoder

Encoder	Decoder
Encoder circuit basically converts the applied information signal into a coded digital bit stream.	Decoder performs reverse operation and recovers the original information signal from the coded bits.
In case of encoder, the applied signal is the active signal input.	Decoder accepts coded binary data as its input.
The number of inputs accepted by an encoder is 2n.	The number of input accepted by decoder is only n inputs.

Encoder	Decoder
The output lines for an encoder is n.	The output lines of an decoder is 2n.
The encoder generates coded data bits as its output.	The decoder generates an active output signal in response to the coded data bits.
The operation performed is simple.	The operation performed is complex.
The encoder circuit is installed at the transmitting end.	The decoder circuit is installed at the receiving side.
OR gate is the basic logic element used in it.	AND gate along with NOT gate is the basic logic element used in it.
It is used in E-mail, video encoders etc.	It is used in Microprocessors, memory chips etc.

Applications of Encoder and Decoder

Applications of Encoders

- Encoders change data into a form that can be sent over long distances. They help phones, computers, and other devices share information across the world by turning messages into special codes that travel easily.
- In robots and machines, encoders turn physical movement into electrical signals. These signals tell the robot or machine its exact position, speed, and direction, helping it move accurately and do its job well.
- Encoders help computers find specific information in their memory quickly. They work like a librarian, turning a request into a code that points directly to where the information is stored.
- Encoders in sensors change real-world measurements into digital signals. This helps measure things like how far something has moved or how fast it's turning, which is useful in many machines and devices.
- In keyboards and other input devices, encoders change our actions (like pressing keys) into a language computers understand. This lets us type, click, and give commands to our devices easily.

Applications of Decoders

- Decoders change computer code into visible numbers, letters, or pictures. They're used in digital clocks, electronic signs, and screens to show information we can read and understand.
- Decoders in devices like TV boxes or internet routers turn incoming signals back into pictures, sound, or data. This is how we can watch TV shows or browse websites sent from far away.

- In computer systems, decoders help find and read the right information from memory. They're like a guide that takes a code and uses it to find and bring back the exact data needed.
- Decoders figure out what different signals mean, like the beeps when you press phone buttons. They turn these signals into instructions that devices can follow or understand.

Conclusion

The Encoders and decoders are essential in the modern technology. The Encoders convert information into the machine friendly codes while the decoders translate these codes back into a usable data. They work behind the scenes in our phones, computers and many other devices. These tools make it possible for the humans and machines to communicate effectively and enabling everything from digital displays to robot control. The encoders and decoders help bridge the gap between the human understanding and machine processing playing the crucial role in world.

What is Multiplexer?

A digital logic circuit which is capable of accepting several inputs and generating a single output is known as **multiplexer** or **MUX**. Thus, the multiplexer is a type of **data selector** which takes many inputs and gives a selected output. In a multiplexer, there are 2^n input lines and 1 output line, where n is the number of select lines.

Therefore, a multiplexer is a combinational circuit which is designed to switch one of the many input lines to a single output line by the use of a control signal. For this reason, the multiplexer is also referred to as a **many to one circuit**. The block diagram of a multiplexer is shown in Figure-1.

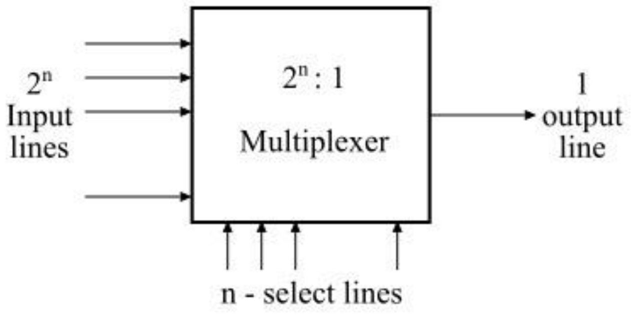


Figure 1 - Multiplexer

The multiplexer functions as a multi-position switch which is digitally controlled by a control signal. In case of the multiplexer, the select lines determine that which input signal will get switched to the output line among the many input signals.

What is Demultiplexer?

A digital combinational circuit which takes one input signal and generates multiple output signals is known as **demultiplexer** or **DEMUX**. As it distributes a single input signal over many output lines, hence it is also referred to as a type of **data distributor**.

In a demultiplexer, there is only 1 input line and 2^n output lines. Where, n denotes the number of select lines. Therefore, it can be noted that a demultiplexer reverses the operation of a multiplexer. The block diagram of a demultiplexer is shown in Figure-2.

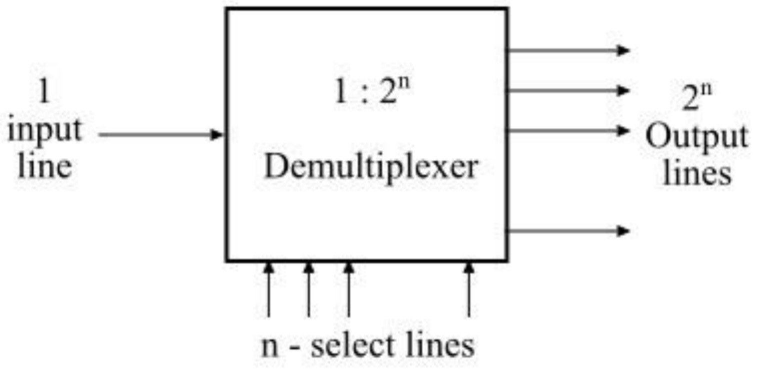


Figure 2 - Demultiplexer

Difference between Multiplexer and Demultiplexer

Both multiplexer and demultiplexer are types of combinational digital circuit that are used in several large-scale digital systems. However, there are many differences between a multiplexer and a demultiplexer, which are highlighted in the following table –

Difference	Multiplexer	Demultiplexer
Definition	A multiplexer is a combinational digital circuit that takes multiple data inputs and provides only single output.	A demultiplexer is a combinational digital circuit that takes single input and provides multiple outputs.
Abbreviated name	The abbreviation used to represent the multiplexer is MUX.	The abbreviation used to represent the demultiplexer is DEMUX.
Input and output lines	Multiplexer has $\mathbf{2^n}$ input lines and 1 output line. Where, n is the number of select lines.	Demultiplexer has 1 input line and $\mathbf{2^n}$ output lines. Where, n is the number of select lines.
Also known as	Multiplexer is also known as a "data selector".	Demultiplexer is also known as "data distributor".
Operating principle	The operating principle of the multiplexer is "many to one".	The operating principle of a demultiplexer is "ne to many".
Acts as	Multiplexer acts as a digital multi-position switch.	Demultiplexer acts as a digital circuit.
Conversion technique	A multiplexer performs parallel to serial conversion.	A demultiplexer performs serial to parallel conversion.

Function of control signal	In case of multiplexer, the function of control signal is to select a specific input that has to be transmitted at the output.	In demultiplexer, the function of control signal is to deliver the single input signal over the multiple output lines.
Examples	Examples of some common multiplexers are − • 8:1 Multiplexer • 16:1 Multiplexer • 32:1 Multiplexer	Some common demultiplexers are − • 1:2 Demultiplexer • 1:4 Demultiplexer • 1:8 Demultiplexer • 1:16 Demultiplexer
Practical importance	In practice, the multiplexer increases the efficiency of the communication system by enabling the data transmission using a single line.	In practice, the demultiplexer takes the output of a multiplexer and convert in its original form at the receiver end.
Usage in time-division multiplexing	A multiplexer is used at the transmitter end in the time-division multiplexing (TDM).	The demultiplexer is used at the receiver end in the time-division multiplexing.
Applications	The multiplexers are commonly used in communication systems, telephone networks, computer memories, etc.	The demultiplexers are used in communication systems, reconstruction of parallel data, ALU, etc.

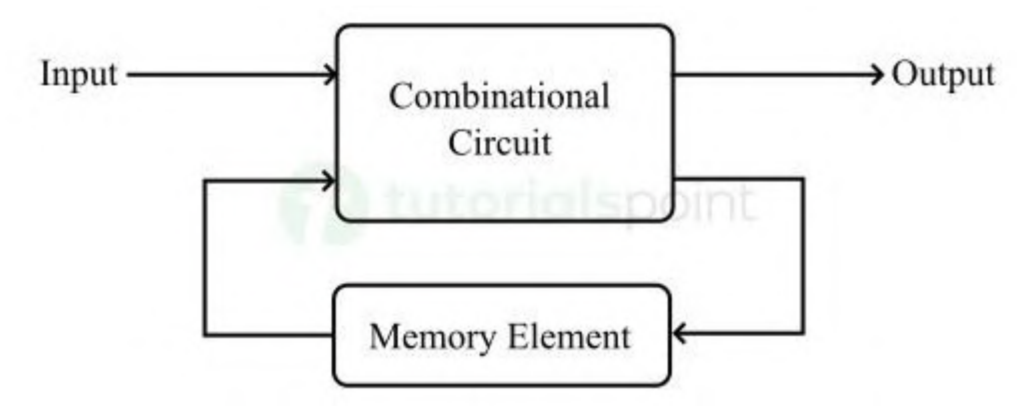
Both **multiplexers** and **demultiplexers** are required in the communication system because of its bidirectional nature. These two devices perform the exact opposite operations of each other. A major difference between multiplexer and demultiplexer is based on their input and output lines, i.e., a multiplexer has many input lines and one output line, whereas a demultiplexer has one input line and many output lines.

What is a Sequential Circuit?

Digital circuits are classified into two major categories namely, combinational circuits **and** sequential circuits**.**

A **sequential circuit** is a logic circuit that consists of a memory element to store history of past operation of the circuit. Therefore, the output of a sequential circuit depends on present inputs as well as past outputs of the circuit.

The **block diagram of a typical sequential circuit** is shown in the following figure −



Here, it can be seen that a sequential circuit is basically a combination of a combinational circuit and a memory element. The combinational circuit performs the logical operations specified, while the memory element records the history of operation of the circuit. This history is then used to perform various logical operations in future.

The sequential circuits are named so because they use a series of latest and previous inputs to determine the new output

Main Components of Sequential Circuit

A sequential circuit consists of several different digital components to process and hold information in the system. Here are some key components of a sequential circuit explained −

Logic Gates

The logic gates like AND, OR, NOT, etc. are used to implement the data processing mechanism of the sequential circuits. These logic gates are basically interconnected in a specific manner to implement combinational circuits to perform logical operations on input data.

Memory Element

In sequential circuits, the memory element is another crucial component that holds history of circuit operation. Generally, flip-flops are used as the memory element in sequential circuits.

In sequential circuits, a feedback path is provided between the output and the input that transfers information from output end to the memory element and from memory element to the input end.

All these components are interconnected together to design a sequential circuit that can perform complex operations and store state information in the memory element.

What is Flip-Flops?

A flip-flop in digital electronics is a circuit with two stable states that can be used to store binary data. The stored data can be changed by applying varying inputs. Flip-flops and latches are fundamental building blocks of digital electronics systems used in computers, communications, and many other types of systems. Both are used as data storage elements.

Why is it called Flip Flop?

Its name comes from its ability to “flip” or “flop” between two stable states. By latching a value and changing it when triggered by a clock signal, flip-flops can store data over time. They are called flip-flops because they have two stable states and switch between them based on a triggering event.

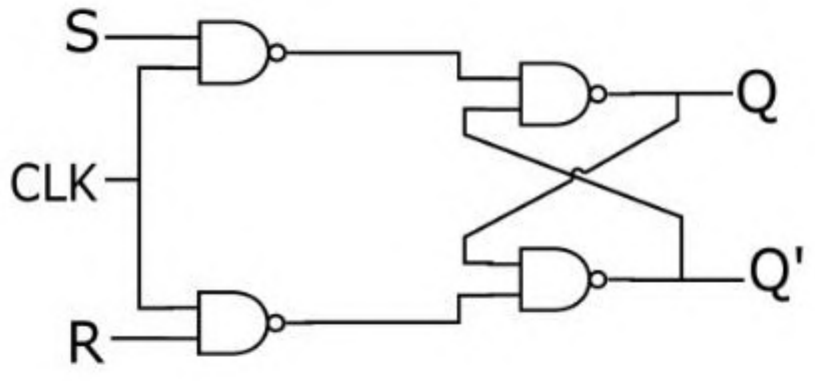
A flip-flop is a sequential digital electronic circuit having two stable states that can be used to store one bit of binary data. Flip-flops are the fundamental building blocks of all memory devices.

Types of Flip-Flops

- S-R Flip-Flop
- J-K Flip-Flop
- D Flip-Flop
- T Flip-Flop

S-R Flip-Flop

This is the simplest flip-flop circuit. It has a set input (S) and a reset input (R). When in this circuit when S is set as active, the output Q would be high and the Q' will be low. If R is set to active then the output Q is low and the Q' is high. Once the outputs are established, the results of the circuit are maintained until S or R gets changed, or the power is turned off.



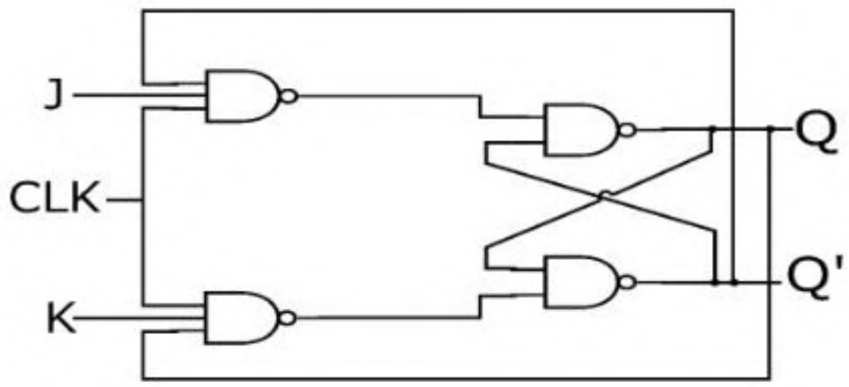
Truth Table of S-R Flip-Flop

S	R	Q	State
0	0	0	No Change
0	1	0	Reset
1	0	1	Set
1	1	X	

J-K Flip-Flop

Because of the invalid state corresponding to S=R=1 in the SR flip-flop, there is a need of another flip-flop. The JK flip-flop operates with only positive or negative clock transitions. The operation of the JK flip-flop is similar to the SR flip-flop. When the input J and K are different then the output Q takes the value of J at the next clock edge.

When J and K both are low then NO change occurs at the output. If both J and K are high, then at the clock edge, the output will toggle from one state to the other.

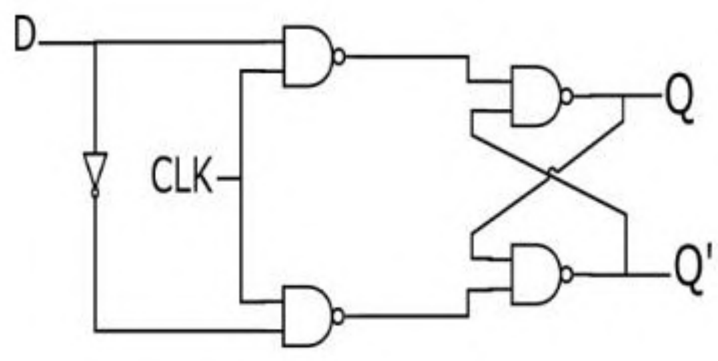


Truth Table of JK Flip-Flop

J	**K**	**Q**	**State**
0	0	0	No Change
0	1	0	Reset
1	0	1	Set
1	1	Toggles	Toggle

D Flip-Flop

In a D flip-flop, the output can only be changed at positive or negative clock transitions, and when the inputs changed at other times, the output will remain unaffected. The D flip-flops are generally used for shift-registers and counters. The change in output state of D flip-flop depends upon the active transition of clock. The output (Q) is same as input and changes only at active transition of clock

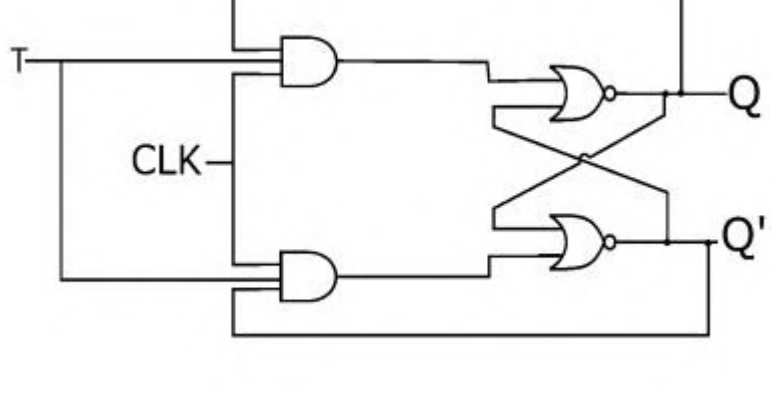


Truth Table of D Flip-Flop

D	**Q**
0	0
1	1

T Flip-Flop

A T flip-flop (Toggle Flip-flop) is a simplified version of JK flip-flop. The T flop is obtained by connecting the J and K inputs together. The flip-flop has one input terminal and clock input. These flip-flops are said to be T flip-flops because of their ability to toggle the input state. Toggle flip-flops are mostly used in counters.



Truth Table of T Flip-Flop

T	**Q(t)**	**Q(t+1)**
0	0	0
0	1	1
1	0	1
1	1	0

Applications of Flip-Flops

- Counters
- Shift Registers
- Storage Registers, etc.

Functional Components and their Interconnections

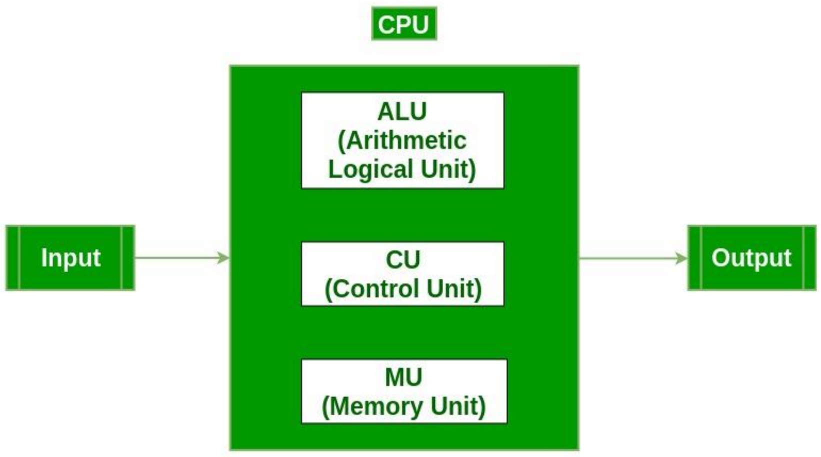
Functional Components of a Computer

Computer: A computer is a combination of hardware and software resources which integrate together and provides various functionalities to the user. Hardware are the physical components of a computer like the processor, memory devices, monitor, keyboard etc. while software is the set of programs or instructions that are required by the hardware resources to function properly.

There are a few basic components that aids the working-cycle of a computer i.e. the Input-Process- Output Cycle and these are called as the functional components of a computer. It needs certain input, processes that input and produces the desired output. The input unit takes the input, the central processing unit does the processing of data and the output unit produces the output. The memory unit holds the data and instructions during the processing.

Digital Computer: A digital computer can be defined as a programmable machine which reads the binary data passed as instructions, processes this binary data, and displays a calculated digital output. Therefore, Digital computers are those that work on the digital data.

Details of Functional Components of a Digital Computer



The basic functional components or elements of a digital computer system basically have the hardware and software. The hardware is the physical component/part such as a keyboard, mouse, monitor, etc. The software is the set of programs and instructions which perform several specific operations.

Both hardware and software together act as functional components. They help to complete the functional cycle which consists of input, processing, and output. Let us learn about the different functional components of a digital computer and their working and interconnections. Let us study the basic components of a computer.

1. Input Unit

The input unit basically includes the input devices and its operation is to take the input from the user. It converts the input data into binary code. As the computer understands only machine language (binary code).

Some important input devices are:

- Keyboard
- Mouse
- Microphone
- Scanner
- Barcode Reader
- Light Pen
- Joystick etc.

2. Central Processing Unit (CPU)

Once the information is entered into the computer by the input device, the processor processes it. The CPU is called the brain of the computer because it is the control center of the computer. It first fetches instructions from memory and then interprets them so as to know what is to be done. If required, data is fetched from memory or input device. Thereafter CPU executes or performs the required computation and then either stores the output or displays on the output device. The CPU has three main components which are responsible for different functions – Arithmetic Logic Unit (ALU), Control Unit (CU) and Memory registers

This is a really important part of a computer as it performs all the processing parts of the computer. It processes the data and instructions which the user gives. Moreover, it carries out the calculations and other such tasks. As it is present on a single small chip, it is also called a microprocessor. Other names of CPU are Central Processor or Main Processor. It has two subparts:

1. Arithmetic and Logical Unit

As the name suggests, this unit is responsible for performing arithmetic tasks like addition, subtraction, multiplication, division moreover, it also makes logical decisions like greater than less than, etc. And hence the name, the 'brain' of the computer.

2. Control Unit

This unit is responsible for looking after all the processing. It organizes and manages the execution of tasks of the CPU.

The Control unit coordinates and controls the data flow in and out of CPU and also controls all the operations of ALU, memory registers and also input/output units. It is also responsible for carrying out all the instructions stored in the program. It decodes the fetched instruction, interprets it and sends control signals to input/output devices until the required operation is done properly by ALU and memory.

3. Registers

These are memory areas which the CPU directly uses for processing. So, it's function is to store data from input or store data between calculations. In addition, it also stores the output results.

A register is a temporary unit of memory in the CPU. These are used to store the data which is directly used by the processor. Registers can be of different sizes(16 bit, 32 bit, 64 bit and so on) and each register inside the CPU has a specific function like storing data, storing an instruction, storing address of a location in memory etc. The user registers can be used by an assembly language programmer for storing operands, intermediate results etc. Accumulator (ACC) is the main register in the ALU and contains one of the operands of an operation to be performed in the ALU.

3. Memory-

Memory attached to the CPU is used for storage of data and instructions and is called internal memory.The internal memory is divided into many storage locations, each of which can store data or instructions. Each memory location is of the same size and has an address. With the help of the address, the computer can read any memory location easily without having to search the entire memory. When a program is executed, it's data is copied to the internal memory and is stored in the memory till the end of the execution. The internal memory is also called the Primary memory or Main memory. This memory is also called as RAM, i.e. Random Access Memory. The time of access of data is independent of its location in memory; therefore this memory is also called Random Access memory (RAM).

The parts of memory are:

Primary Memory

This is the internal memory that stores the data and instructions of the CPU. It is volatile in nature (data is lost when the power is disconnected).

The primary memory has two types:

1. **RAM (Random Access Memory)**

   As per the name, data can be accessed randomly and quickly.

2. **ROM (Read Only Memory)As per the name, we can only read data and cannot write (store) to it.**

As per the name, data can be accessed randomly and quickly.

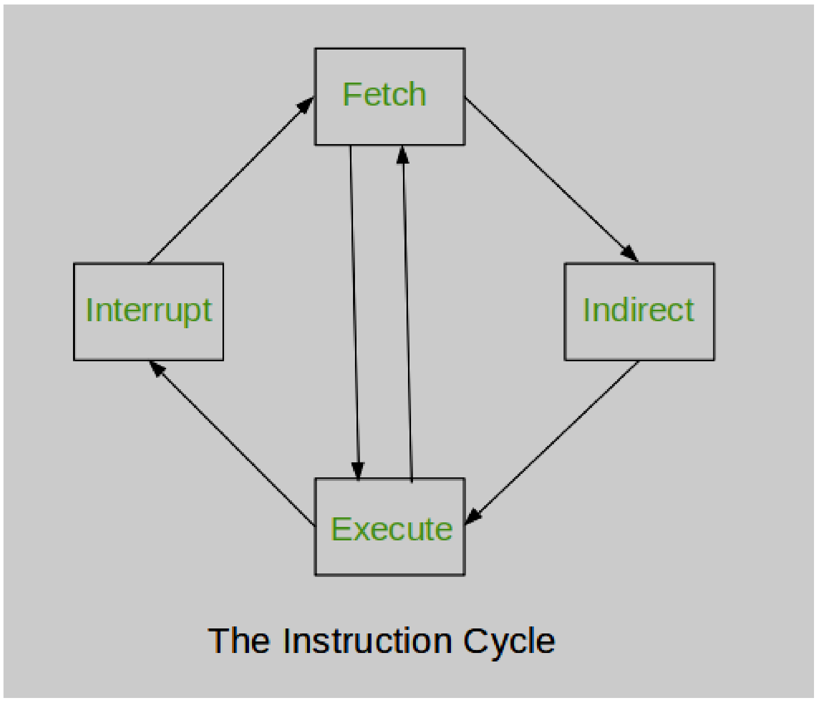
Secondary Memory

As we know that the primary memory is volatile therefore, we need some devices to store the data permanently so we use some external storage devices for this purpose which we name as the secondary memory. Some examples: CD, DVD, etc.

Output Unit : The output unit consists of output devices that are attached with the computer. It converts the binary data coming from CPU to human understandable form. The common output devices are monitor, printer, plotter etc.

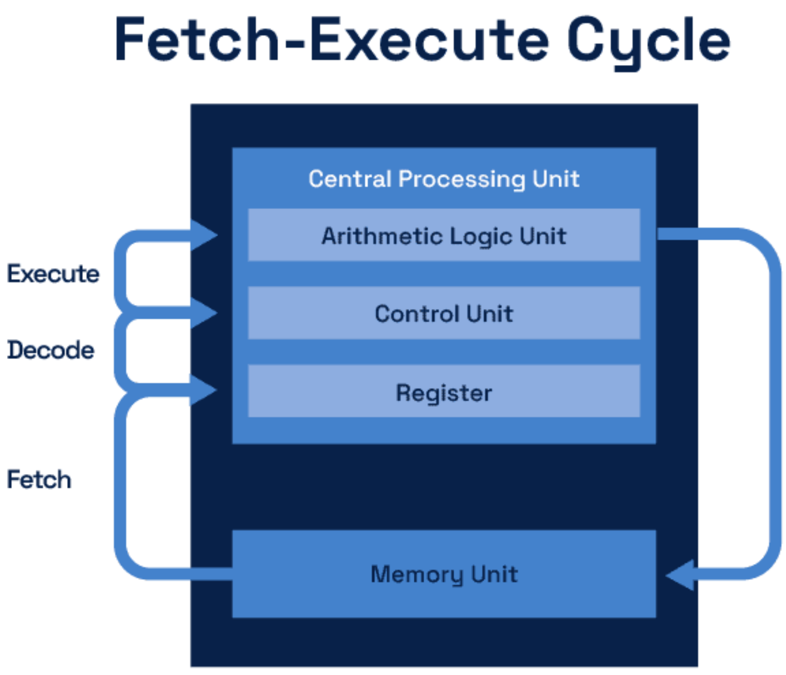
Instruction Cycle: Fetch, Decode and Execute Cycle

The CPU executes the instructions regarding a program stored in the memory. There's one general rule applied to all these instructions being carried out in the processors. The execution definition is outlined by a cycle of instructions conducted in the particular execution. This cycle, better known as the instruction cycle, has three stages – fetch, decode and execute.



What is the Instruction Cycle?
The execution instructions define the instruction cycle. This is the thorough methodology computer processors use for executing a given instruction. Many times processors can be compared to combustion engines. Both follow a process continuously being carried out to fetch

the desired outcome. Every processor shows a three-step instruction cycle. These three steps of the instruction execution cycle are,



1. Fetch:
The processor copies the instruction data captured from the RAM.

2. Decode:
Decoded captured data is transferred to the unit for execution.

3. Execute:
Instruction is finally executed. The result is then registered in the processor or RAMS (memory address).

First step: Fetch (instruction cycle)
According to the execution instruction definition, the instruction cycle's first step is to capture or fetch the instruction. This instruction in the fetch stage is captured from RAM. This memory is assigned to the processor through various units and registers; they are:

- **Program counter:**

  It works by pointing towards the next memory line, where the next instruction for the processor is stored.

- **Register (memory address):** Responsible for copying PC content and sending it to the RAM. This is done through the CPU's address pins.

- **Register (memory data):**

  It takes the responsibility of copying the memory address to the internal register.

- **Register (instruction):**

  It has involvement in the last step of the fetch phase. It is where the instruction is written. The control unit from here copies the content for carrying out the instruction cycle.

How does the control unit work?
As discussed earlier, the control unit plays an important part and is present in a processor. The tasks that it performs are,

- The control unit is responsible for controlling both internal and external movements of the data in the processor. Also, it is responsible for controlling the movement of the data in various subunits involved
- Various units of the capture stage of an instruction cycle are considered part of the hardware. This hardware is called a control unit or a processor's front-end
- It is responsible for interpreting various instructions and sending them to the execution units
- Communicate that data for instruction to the various ALUs and execution units at work
- It is the part of the processor that captures and decodes the instruction for execution. Also, it is responsible for writing results on the registers and even in the respective addresses of the RAM

Second step: Decode (instruction cycle)
There are various instructions, and we can never be sure which instruction belongs to which execution unit. Decoding sorts this out. A decoder is responsible for taking in the instruction and decoding it to assign the respective execution unit to complete the execution instruction cycle.

The easiest example of how an instruction works is visualising them as trains that keep circulating through a complex railway network. The control unit here acts as the station at the terminal and, therefore, it is held in charge of being the execution unit to solve the given instruction.

Third step: Execute (instruction cycle)
The last stage of the execute instruction definition is to execute. It involves executing the given instruction that was fetched at the first stage. No two instructions ever get resolved in the same manner because their ways of utilising the hardware depend on their functions. There are four types of instructions that are generally present,

- **Bit movement instructions:**
  This instruction involves the manipulation of the bits' order. These bits contain the data.

- **Arithmetic instructions:**

These are the instructions that involve logical as well as mathematical operations. They are most often solved in arithmetic logical units (ALUs)

- **Jump instructions:**
  The code in this instruction is used recursively because the value for the next program counter is changed.

- **Instructions to memory:**
  These instructions involve the processor writing as well as reading the information from the memory of the system.

On completing the cycle, that is, the instruction being executed, a new instruction gets fetched, and the cycle continues.

Conclusion:

The processor in a computer is responsible for the beginning of the instruction cycle and the forthcoming steps of the instruction execution cycle. The first step for the processor is to check the next instruction to run in the program counter.

The program counter now gives the address value for the next instruction in the memory. The value for instruction is fetched out of this given location within the memory; after being fetched just according to the execution instruction definition, decoding and executing follow suit.

After this instruction is executed, the processor again turns to the program counter. It gets a new instruction. This way, the instruction cycle keeps repeating itself unless the instruction for STOP.

Instruction Interrupts in Computer Architecture (Simple Explanation)

An **instruction interrupt** is when a computer's processor stops what it is doing to handle something more urgent. Think of it like a teacher pausing a lesson to answer an emergency call.

In computer architecture, an **instruction interrupt** is an event that temporarily halts the processor's execution of a program so that it can handle a more urgent task. After addressing the interrupt, the processor resumes its previous task. Interrupts play a crucial role in multitasking, error handling, and efficient system performance.

1. What is an Interrupt?

An **interrupt** is a signal sent to the processor to indicate that an event needs immediate attention. Instead of continuously checking for events (which wastes processing power), the CPU **waits for an interrupt signal** before reacting.

□ **Example:** Imagine a teacher is writing on the board (executing instructions), and a student raises their hand (interrupt). The teacher stops, listens to the student (handles the interrupt), and then resumes writing.

Why Do Interrupts Happen?

Interrupts occur when the processor needs to pay attention to something important, like:

1. **Hardware Issues** (e.g., keyboard input, printer ready, mouse click).
2. **Software Requests** (e.g., a program asking for more memory).
3. **Errors** (e.g., divide by zero, illegal operation).

Types of Interrupts

1. **Hardware Interrupts** – Comes from external devices (e.g., pressing a key on the keyboard).
   Example-
   These are triggered by external devices like the keyboard, mouse, or printer.
   Example 1: You press a key on the keyboard → CPU stops to process the keystroke.
   Example 2: The printer finishes printing → CPU gets an interrupt to send more data.

2. **Software Interrupts** – Comes from programs (e.g., requesting an OS service).
   Example 1: A program requests a system service (e.g., saving a file).
   Example 2: The operating system schedules a new task for execution.

3. **Exceptions** – Comes from errors in execution (e.g., division by zero).
   Example 1: A program tries to divide a number by zero (division by zero error).
   Example 2: A program accesses memory that it shouldn't (invalid memory access).

How Does an Interrupt Work? (Step-by-Step Process)

The interrupt handling process involves several key steps:

1. **The CPU is executing a program.**
2. **An interrupt occurs** (from hardware, software, or an error).
3. **The CPU pauses execution** of the current program.
4. **The CPU saves its current state** (register values, program counter, etc.).
5. **The CPU identifies the source of the interrupt.**
6. **The CPU jumps to the Interrupt Service Routine (ISR),** a special program designed to handle the interrupt.
7. **The ISR executes** to handle the interrupt (e.g., process a keypress, send data to a printer, or fix an error).
8. **The CPU restores its previous state** (loads saved data).
9. **The CPU resumes execution** of the interrupted program.

Example:

- You type "Hello" on the keyboard.
- Each key press sends an **interrupt** to the CPU.
- The CPU **pauses the current task** and calls an interrupt handler to store and display the letter.
- After displaying the letter, the CPU **resumes the previous program**.

Benefits of Interrupts

Efficient CPU Usage: The processor doesn't waste time checking for events but reacts when needed.
Enables Multitasking: The CPU can switch between tasks efficiently.
Handles Errors Quickly: Prevents system crashes by detecting faults.
Improves System Responsiveness: Devices like keyboards and printers get quick responses.

Example Scenario of Interrupt Handling

Scenario: You Press a Key on Your Keyboard

1. The keyboard sends an **interrupt signal** to the CPU.
2. The CPU **pauses the current program** (e.g., watching a video).
3. The CPU saves its current state.
4. The CPU jumps to the **keyboard interrupt handler**.
5. The handler reads the **key pressed** and processes it.
6. The CPU **restores its previous state** and resumes video playback.

Conclusion-Interrupts are a key part of modern computer systems, allowing them to efficiently handle multiple tasks, respond to user input, and prevent crashes. Whether they come from hardware, software, or errors, interrupts ensure that the system remains fast and responsive.

Interconnection Structures

Interconnection structures are the pathways that allow different components of a computer (like CPU, memory, and input/output devices) to communicate with each other. These structures are essential because, without them, the components wouldn't be able to share data or coordinate tasks.

In a computer system, different components like the **CPU (Processor), Memory (RAM, Cache), and Input/output (I/O) devices** need to communicate with each other to perform tasks. **Interconnection structures** are the systems that allow this communication to happen efficiently.

There are different ways to connect these components, and the choice of structure affects the speed, cost, and performance of the system.

There are five main types of interconnection structures:

1. **Bus Interconnection**
2. **Point-to-Point Interconnection**

1. Bus Interconnection

A **bus** is a single communication line (like a shared road) used by multiple components to send and receive data. It consists of three types of signals:

- **Data Bus** → Transfers data between CPU, memory, and I/O devices.
- **Address Bus** → Tells where the data should go (memory address or I/O port).
- **Control Bus** → Manages signals to control data transfer (e.g., Read/Write).

How It Works

Think of it like a **school bus** that picks up and drops off students (data) at different stops (CPU, memory, etc.). Only one student (data) can get on or off at a time.

Advantages:

Simple and low-cost → Fewer wires and circuits needed.
Easy to expand → Can add more devices without major changes.

Disadvantages:

Slow when many devices are connected → Just like a crowded road, data transfer becomes slower.
Only one device can communicate at a time → Others have to wait their turn.

Where it is used:

- Used in **small computers, embedded systems, and general-purpose processors**.

---

2. Point-to-Point Interconnection

In this structure, each component has a dedicated link to another component, similar to a **direct phone call** between two people.

How It Works

Instead of using a shared bus, the CPU and memory have a **direct** connection, ensuring fast and interference-free communication.

Advantages:

Fast communication → No waiting like in a bus system.
No data collision → Since each connection is dedicated.

Disadvantages:

Expensive → More wires and circuits are needed.
Difficult to expand → Adding new components means adding new connections.

Where it is used:

- Found in **high-speed processors, high-end graphics cards, and direct memory connections (like in high-performance gaming PCs and servers).**

What are Registers in a Computer?

A **register** is a small, high-speed storage location inside the **CPU (Central Processing Unit)**. It temporarily holds data, instructions, or addresses that the CPU needs **quick access to** while executing tasks.

Why Do We Need Registers?

- CPU processes data much faster than RAM, so it needs a **faster** place to store data temporarily.
- Registers help reduce the time taken to **fetch** and **execute** instructions.
- They improve the overall speed and efficiency of the computer.

Types of Registers in a Computer

Registers can be classified into different types based on their function. The main types of registers are:

1. **Accumulator (AC)**
2. **Program Counter (PC)**
3. **Instruction Register (IR)**

1. Program Counter (PC)

The **Program Counter (PC)** is a special register in the CPU that keeps track of the **next instruction** that needs to be executed.

Key Features of the Program Counter:

Stores the memory address of the next instruction to be fetched and executed.
Automatically increments after each instruction are executed.
Controls the flow of execution in a program.

How the Program Counter Works (Step by Step):

1. **Fetch:** The CPU reads the **memory address** stored in the PC to fetch the next instruction.
2. **Increment:** The PC increases by 1 (or by the instruction size) to move to the next instruction.
3. **Execute:** The instruction is decoded and executed.
4. **Repeat:** The process continues, ensuring that the CPU follows the sequence of instructions in the program.

Example of the Program Counter in Action:

Imagine a **to-do list** where each task has a number:

Task Number	Task Description
1000	Add 5 and 3
1001	Multiply the result by 2
1002	Store the result in memory

- Initially, the **PC = 1000**, pointing to the first instruction.
- After fetching instruction **1000**, the **PC increments to 1001**.
- The next instruction (1001) is executed, and the **PC moves to 1002**, and so on.

Advantage: Ensures that instructions are executed **in the correct sequence**.
Disadvantage: If the PC is modified incorrectly (e.g., due to a programming error), the program may crash.

---

2. Accumulator (AC)

The **Accumulator** is a special-purpose register inside the **Arithmetic Logic Unit (ALU)** that stores the results of **arithmetic and logical operations**.

Key Features of the Accumulator:

Stores temporary results of **add, subtract, multiply, divide, and logic** operations.
Helps in **reducing memory access**, making the CPU faster.
Works directly with the **ALU** to process data.

How the Accumulator Works (Step by Step):

1. **Load Data:** The CPU loads the first number into the Accumulator.
2. **Perform Operation:** The ALU performs the operation (e.g., addition, subtraction) using the value in the Accumulator.
3. **Store Result:** The final result remains in the Accumulator until the next instruction moves it elsewhere.

Example of the Accumulator in Action:

Let's say we want to compute:
(5 + 3) × 2

Step	Operation	Accumulator (AC)
1	Load **5** into AC	5
2	Add **3** to AC	8
3	Multiply AC by **2**	16
4	Store the result in memory	AC is cleared

Advantage: Speeds up calculations because the CPU does not have to store intermediate results in memory.
Disadvantage: Can only hold **one** value at a time, so it must be emptied or overwritten for new calculations.

---

3. Instruction Register (IR)

The **Instruction Register (IR)** is used to **store the instruction that is currently being executed** by the CPU.

Key Features of the Instruction Register:

Holds the **binary machine code** of the current instruction.
The instruction inside the IR is **decoded** by the CPU to determine what action to take.
Works with the **Control Unit** to execute instructions correctly.

How the Instruction Register Works (Step by Step):

1. **Fetch:** The CPU fetches an instruction from memory and places it in the **Instruction Register (IR)**.
2. **Decode:** The Control Unit reads the instruction from IR and **decodes** it (e.g., whether it’s an addition, subtraction, or data transfer instruction).

3. **Execute:** The CPU performs the operation as per the decoded instruction.
4. **Clear:** The instruction is removed from IR after execution, and the next instruction is fetched.

Example of the Instruction Register in Action:

Suppose a program contains the following **assembly code**:

```
LOAD A  → Load value from memory into AC
ADD B   → Add value of B to AC
STORE C → Store result in memory location C
```

Step	Program Counter (PC)	Instruction Register (IR)	Action
1	**1000**	LOAD A	Load A into AC
2	**1001**	ADD B	Add B to AC
3	**1002**	STORE C	Store AC into C

Advantage: Ensures the CPU knows **exactly which instruction** is being processed at a given moment.
Disadvantage: Can hold **only one instruction** at a time, so it must be refreshed frequently.

Final Comparison Table

Register	Function	Example
Program Counter (PC)	Holds the address of the next instruction to execute.	If PC = 1000, the CPU will fetch the instruction stored at address 1000.
Accumulator (AC)	Stores the result of arithmetic and logic operations.	If CPU adds 5 + 3, the result (8) is stored in AC.
Instruction Register (IR)	Holds the currently executing instruction.	If IR contains "ADD A, B", the CPU adds values of A and B.

Conclusion

- The **Program Counter (PC)** ensures that instructions are executed in the correct sequence.
- The **Accumulator (AC)** is a temporary storage for mathematical and logical operations.

- The **Instruction Register (IR)** holds and decodes the current instruction.

Each of these **works together inside the CPU** to ensure fast and efficient execution of programs.

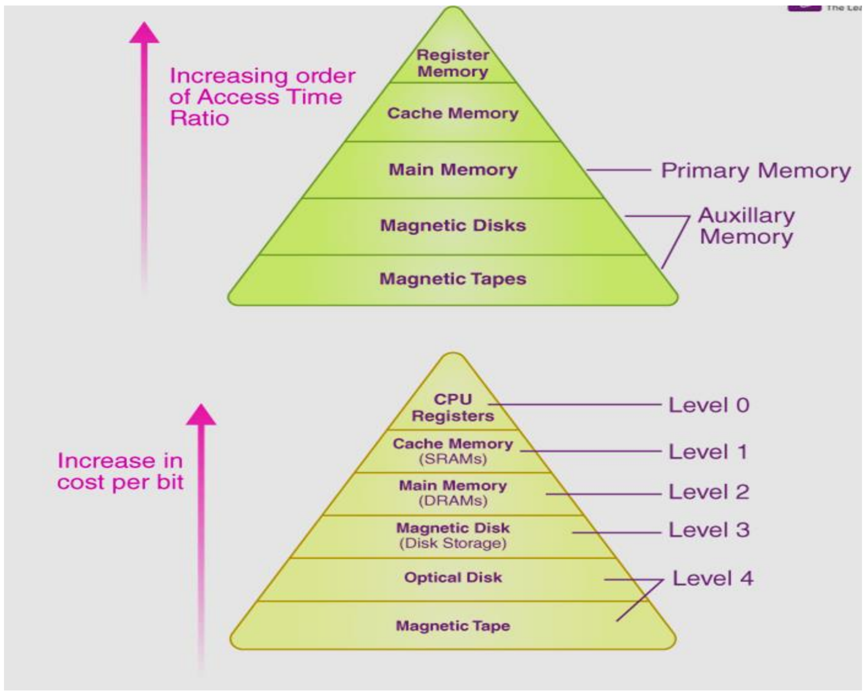
Memory Organization in Computers

Memory organization in a computer defines how data is **stored, accessed, and managed** efficiently to help the CPU perform operations quickly.

Computers use **different types of memory**, each with **different speeds, sizes, and access methods**. These memory types are organized in a **hierarchy**, with the **fastest memory closest to the CPU** and the **slowest memory farthest from it**.

Memory Hierarchy (Diagram)

The following diagram shows how different memory types are organized in a **hierarchical structure**:



The memory in a computer can be divided into five hierarchies based on the speed as well as use. The processor can move from one level to another based on its requirements. The five hierarchies in the memory are registers, cache, main memory, magnetic discs, and magnetic tapes. The first three hierarchies are volatile memories which mean when there is no power, and then automatically they lose their stored data. Whereas the last two hierarchies are not volatile which means they store the data permanently.

A memory element is the set of storage devices which stores the binary data in the type of bits. In general, the storage of memory can be classified into two categories such as volatile as well as non- volatile.
The **memory hierarchy design** in a computer system mainly includes different storage devices. Most of the computers were inbuilt with extra storage to run more powerfully beyond the main memory capacity. The following **memory hierarchy diagram** is a hierarchical pyramid for computer memory. The designing of the memory hierarchy is divided into two types such as primary (Internal) memory and secondary (External) memory.

Primary Memory
The primary memory is also known as internal memory, and this is accessible by the processor straightly. This memory includes main, cache, as well as CPU registers.

Secondary Memory
The secondary memory is also known as external memory, and this is accessible by the processor through an input/output module. This memory includes an optical disk, magnetic disk, and magnetic tape.

.Registers (Fastest Memory, Inside CPU)

Definition:
Registers are **super-fast, small storage units inside the CPU** used to hold temporary data and instructions during processing.

Characteristics of Registers

Fastest memory type (nanosecond-level speed).
Directly accessed by the CPU.
Stores data for immediate processing.
Very small in size (a few bytes).

Example of Registers in Action:

- Suppose the CPU is calculating **5 + 3**:
  1. The number **5** is stored in a **Register**.
  2. The number **3** is stored in another **Register**.
  3. The ALU (Arithmetic Logic Unit) performs **5 + 3**.
  4. The result (**8**) is stored in the **Accumulator Register**.

Types of Registers:

Register	**Function**

Register	Function
Program Counter (PC)	Holds the address of the next instruction to execute.
Instruction Register (IR)	Holds the instruction currently being executed.
Accumulator (AC)	Stores intermediate results of calculations.

Advantage: Ultra-fast memory, directly inside the CPU.
Disadvantage: Very limited storage (only a few bytes).

What is Main Memory?

Main Memory (also called **Primary Memory** or **RAM**) is the **working memory of the computer** where programs and data are stored temporarily **while they are being used**.

Key Points:
Main memory is **directly accessible** by the CPU.
It is **faster than secondary storage** (HDD/SSD) but **slower than Cache**.
Data is lost when power is off (Volatile Memory).
It helps in the **execution of programs** by storing instructions and data temporarily.

Main Memory Diagram

```
   CPU  <---->  Cache Memory  <---->  Main Memory (RAM)  <---->  Secondary
Storage (HDD/SSD)
```

How Data Moves in a Computer?

1. The CPU first looks for data in **Cache Memory**.
2. If the data is not in Cache, it is fetched from **Main Memory (RAM)**.
3. If it’s not in RAM, it is **loaded from Secondary Storage (HDD/SSD) into RAM** for faster access.

Types of Main Memory

Main Memory is divided into **two types:**
1**RAM (Random Access Memory) - Read & Write Memory**
2**ROM (Read-Only Memory) - Permanent Memory**

---

1. RAM (Random Access Memory)

Definition:
RAM is a type of memory that allows the **CPU to read and write data quickly**. It stores data **temporarily** and loses all data when power is turned off.

Characteristics of RAM

Fast access speed (compared to HDD/SSD).
Directly accessible by CPU.
Stores active programs and data for quick execution.
Volatile Memory (data is lost when power is off).

Example of RAM in Action:

- When you **open a web browser**, the browser program is loaded into RAM.
- When you **close the browser**, the memory is freed for other tasks.

Types of RAM

Type	Description
SRAM (Static RAM)	Faster and expensive, used in **Cache Memory**.
DRAM (Dynamic RAM)	Slower and cheaper, used in **Main Memory (RAM)**.

Advantage of RAM: Much faster than SSD/HDD.
Disadvantage of RAM: Data is lost when power is off.

---

2. ROM (Read-Only Memory)

Definition:
ROM is **permanent memory** that stores **important system instructions** like the **BIOS (Basic Input/Output System)**.

Characteristics of ROM

Non-volatile Memory (data is **not lost** when power is off).
Stores firmware & boot instructions for the computer.
Read-only (Cannot be modified easily).

Example of ROM in Action:

- When you **turn on a computer**, the **BIOS (stored in ROM)** loads the **Operating System (OS)**.

Types of ROM

Type	Description
PROM (Programmable ROM)	Can be programmed once.
EPROM (Erasable Programmable ROM)	Can be erased using UV light and reprogrammed.
EEPROM (Electrically Erasable Programmable ROM)	Can be erased electronically and rewritten multiple times.

Advantage of ROM: Stores **important system data permanently**.
Disadvantage of ROM: Cannot be modified easily.

Differences between RAM & ROM

Feature	RAM (Random Access Memory)	ROM (Read-Only Memory)
Type	Volatile Memory (Data is lost when power is off)	Non-Volatile Memory (Data is permanent)
Usage	Stores programs & data for CPU processing	Stores system firmware & boot instructions
Read/Write	Read & Write	Read-Only
Speed	Fast	Slower than RAM
Example	Running programs in Windows/Linux	BIOS, Firmware in devices

- **Main Memory (RAM & ROM) is essential** for program execution.
- **RAM is fast but volatile**, used for **temporary data storage**.
- **ROM is non-volatile**, used for **permanent system instructions**.
- **Virtual Memory acts as backup RAM** but is much slower.

What is Auxiliary Memory?

Auxiliary Memory (also called **Secondary Storage**) is a type of **permanent memory** used to store data and programs **for long-term use**. It is **not directly accessed by the CPU** and is used when data needs to be retrieved for future use.

Key Points:
Stores data permanently (Non-volatile memory).
Larger storage capacity than RAM or Cache.
Slower than Main Memory (RAM) but much **cheaper**.
Used for backup and long-term storage.

How it Works?
1 The **CPU first checks Cache & RAM** for required data.
2 If not found, it **fetches data from Auxiliary Memory** (HDD/SSD).
3 Once data is in RAM, it can be **processed faster** by the CPU.

Types of Auxiliary Memory

Auxiliary Memory is divided into the following types:
1 **Magnetic Storage** (Hard Disk, Magnetic Tape)
2 **Optical Storage** (CD, DVD, Blu-ray)
3 **Flash Storage** (SSD, USB Drives, Memory Cards)
4 **Cloud Storage** (Google Drive, OneDrive, Dropbox)

Magnetic Storage (HDD, Floppy Disks, Magnetic Tapes)

Definition:
Magnetic storage **uses magnetic fields to store data** on rotating disks or tapes.

Characteristics of Magnetic Storage

Non-volatile (Data remains even when power is off).
Large storage capacity (up to several TBs).
Cheaper than SSDs and Flash storage.
Slower than RAM but faster than Tertiary storage.

Example:

- **Hard Disk Drives (HDDs)** are used in desktop computers and laptops.
- **Magnetic Tapes** are used for **data backup in large organizations**.

Advantages & Disadvantages:

Advantage: Cost-effective, large storage.
Disadvantage: Slower than SSDs, mechanical parts can fail.

---

2. Optical Storage (CD, DVD, Blu-ray)

Definition:
Optical storage uses **laser technology** to read and write data on discs like **CDs, DVDs, and Blu-ray discs**.

Characteristics of Optical Storage

Used for multimedia files, software, and backup.
Portable and cheap compared to hard drives.
Slower than HDDs and SSDs.

Example:

- **CDs (Compact Discs)** store music and small software (700MB).
- **DVDs (Digital Versatile Discs)** store movies and large files (4.7GB – 17GB).
- **Blue-ray Discs** are used for **high-definition (HD) movies** (25GB – 128GB).

Advantages & Disadvantages:

Advantage: Portable and cheap.
Disadvantage: Limited storage, scratches can damage data.

---

3. Flash Storage (SSD, USB, Memory Cards)

Definition:
Flash storage uses **electronic memory (NAND flash)** to store data **without moving parts**.

Characteristics of Flash Storage

Faster than HDDs (Used in SSDs and USB drives).
More durable because it has no moving parts.
Commonly used in laptops, smart phones, and cameras.

Example:

- **Solid State Drives (SSD)** are much faster than HDDs and used in modern laptops.
- **USB Flash Drives** (Pen Drives) are used for portable storage.
- **Memory Cards** store data in mobile phones and cameras.

Advantages & Disadvantages:

Advantage: High-speed storage, durable, low power consumption.
Disadvantage: More expensive than HDDs.

---

4. Cloud Storage (Online Backup)

Definition:
Cloud storage is an **internet-based storage system** where data is stored on **remote servers** instead of local devices.

Characteristics of Cloud Storage

Access data from anywhere using the internet.
Data is backed up on multiple servers to prevent loss.
Used for storing large files, backups, and sharing files globally.

Example:

- **Google Drive, One Drive, Drop box** store personal and business files.
- **iCloud & Amazon AWS** store photos, videos, and documents.

Advantages & Disadvantages:

Advantage: No need for physical storage, easy sharing.
Disadvantage: Requires an internet connection, security concerns.

• Auxiliary **Memory is used for long-term data storage** (HDD, SSD, USB, CD, Cloud).
• It **is slower than Main Memory (RAM)** but much **cheaper and larger in size**.
• Cloud **Storage is growing**, reducing the need for physical storage.
• SSDs **are replacing HDDs** in modern computers due to their speed.

What is Associative Memory?

Associative memory is a **special type of memory** that allows **data to be accessed based on content rather than a specific address**. It is also known as **Content-Addressable Memory (CAM)** because it searches for data by matching content instead of using a memory address like RAM.

Key Points:
Faster than traditional memory because it searches data by content.
Used in applications where quick searches are required (e.g., cache memory, networking).
More expensive than RAM due to complex design.
Can retrieve multiple values at the same time, unlike traditional memory.

How Associative Memory Works (Simple Explanation)

Traditional Memory (RAM) vs. Associative Memory

In **RAM (Random Access Memory):**

- The CPU requests data from a **specific address**.
- Example: "Get data from address **1010**."

In **Associative Memory:**

- The CPU provides **a key (content),** and the memory **searches for matching data automatically**.
- Example: "Find the memory block that contains the word **'Password123'**."

Analogy:
Imagine a **library with books:**

- In **RAM**, you find a book **by its shelf number**.
- In **Associative Memory**, you find a book **by its title or content**.

Associative Memory Diagram

```
          ----------------------
         |  Search Key (Input)  |
          ----------------------
                    |
          ----------------------
         |  Associative Memory  |
          ----------------------
                    |
        ----------------------
       |  Matched Data (Output)  |
        ----------------------
```

The **Search Key** is provided → **Memory searches automatically** → **Returns Matched Data**

Applications of Associative Memory

Where is Associative Memory Used?

Application	Usage
Cache Memory	Stores frequently accessed data for faster CPU access.
Networking (Routers, Switches)	Used in **IP address lookups and routing tables** for fast packet forwarding.
Database Searching	Helps in **fast searching** based on **keywords** or **content**.
Artificial Intelligence (AI)	Used in **pattern recognition and machine learning** for fast data matching.
Security Systems	Used in **password matching** and **biometric verification**.

Types of Associative Memory

There are **two main types** of associative memory:
1**Binary Associative Memory**
2**Multivalued Associative Memory**

1. Binary Associative Memory

Stores **data as binary values (0s and 1s)**.
Uses **bitwise operations** to compare and retrieve data.
Example: **Used in high-speed cache memory**.

2. Multivalued Associative Memory

Stores **data as multiple values** (not just 0s and 1s).
Can store **complex data structures like numbers, words, and images**.
Example: **Used in AI and database systems** for advanced searching.

Advantages & Disadvantages of Associative Memory

Feature	Advantage	Disadvantage
Speed	Very fast search (No need for address-based lookup).	More power consumption.
Efficiency	Finds multiple matching results at once.	Expensive hardware.
Usage	Used in **high-speed applications** (networking, AI).	Complex to design.

Associative Memory is a special type of memory that searches for data based on content rather than memory addresses.
It is used in high-speed applications **like** cache memory, networking, and AI systems.
Faster than RAM **but** more expensive **due to its complex design.**

What is Cache Memory?

Cache memory is a **small, high-speed memory** that **stores frequently used data** so the CPU can access it quickly. It is **faster than RAM** and helps the CPU work **more efficiently** by reducing the time needed to access data from main memory (RAM).

Key Points:
Stores frequently accessed data temporarily.
Much faster than RAM but smaller in size.
Located closer to the CPU for quick access.
Improves system performance and processing speed.

Why is Cache Memory Needed?

Problem Without Cache Memory

- The CPU is very fast, but RAM is slower.
- Every time the CPU needs data, it must fetch it from RAM, which **slows down processing**.

Solution: Using Cache Memory

- Cache memory **stores frequently used data** so the CPU **does not have to fetch it from RAM**.
- This makes processing **much faster** because **cache memory is closer to the CPU** and has a **higher speed** than RAM.

Cache Memory Diagram

```
         ---------------------
        |       CPU          |
         ---------------------
                 ⬆
         ---------------------
        |   Cache Memory     |  (Fastest but Smallest)
         ---------------------
                 ⬆
         ---------------------
        |   Main Memory (RAM) |  (Slower but Larger)
         ---------------------
                 ⬆
         ---------------------
        | Secondary Storage  |  (Slowest, Largest - HDD/SSD)
         ---------------------
```

The **CPU first checks the Cache** for data.
If data is **not found**, it goes to **RAM** and then to **Storage (HDD/SSD)**.

Levels of Cache Memory (L1, L2, L3)

Cache memory is divided into different levels based on speed and proximity to the CPU:

Level	Location	Size	Speed	Function
L1 (Level 1) Cache	Inside CPU	Smallest (2KB - 64KB)	Fastest	Stores the most frequently used data

Level	Location	Size	Speed	Function
L2 (Level 2) Cache	Inside/Outside CPU	Medium (256KB - 8MB)	Fast	Backup for L1 cache
L3 (Level 3) Cache	Shared by CPU cores	Largest (4MB - 64MB)	Slower than L1 & L2 but faster than RAM	Stores additional frequently used data

Example:

- **L1 Cache**: The closest to the CPU, stores **immediate instructions**.
- **L2 Cache**: A backup for L1, holds **recently accessed data**.
- **L3 Cache**: Shared among multiple CPU cores for **improving multi-core performance**.

How Cache Memory Works (Step-by-Step)

1**CPU requests data** → First **checks the cache memory**.
2**If data is found in cache** (Cache Hit) → CPU **uses it instantly** (Fastest Access).
3**If data is NOT found in cache** (Cache Miss) → Data is **fetched from RAM or Storage** (Slower Access).
4The fetched data is **stored in cache** for future use.

Cache Hit = Faster Processing ✅
Cache Miss = Slower Processing ❌

Types of Cache Memory

There are **two main types** of cache memory:

1 Primary Cache (Internal Cache)

- Built **inside the CPU**.
- **Faster but smaller** in size.
- Example: **L1 and L2 cache.**

2 Secondary Cache (External Cache)

- Located **near the CPU but not inside**.
- **Larger but slightly slower** than primary cache.
- Example: **L3 cache.**

- ## Advantages & Disadvantages of Cache Memory

- **Advantages**
- **Faster processing** (Reduces CPU waiting time).
  Reduces RAM access, improving performance.
  Increases system speed for tasks like gaming, video editing.

- **Disadvantages**
- ❌**Small storage size** (Limited MBs compared to RAM).
  ❌**Expensive** (More costly than RAM).
  ❌**Complex design** (Difficult to manage data storage).

- **Cache memory is a high-speed memory** that **stores frequently accessed data** for faster CPU processing.
  It is **faster than RAM** but **smaller in size**.
  There are 3 levels (L1, L2, L3) used in modern processors.
  Used in applications that requires high-speed performance (Gaming, AI, and Networking).

What is Virtual Memory?

- Virtual Memory is a **technique that allows a computer to use part of its hard disk (or SSD) as extra RAM** when the actual RAM is full. It helps run large programs and multitask smoothly, even when the physical memory (RAM) is limited.
- **Key Points:**
  Expands available memory by using storage (HDD/SSD) as RAM.
  Prevents system crashes when RAM is full.
  Slower than actual RAM but still helps run large applications.
  Used when running multiple programs at once.

Why is Virtual Memory Needed?

Problem Without Virtual Memory:

- If RAM is **fully occupied**, the computer **cannot load more programs**.
- Running **multiple applications at once** can **slow down or crash** the system.

Solution: Using Virtual Memory

- The computer moves **less frequently used data from RAM to a special space on the hard disk (called a page file or swap file).**
- This **frees up space in RAM** for active programs, making the system more efficient.

Virtual Memory Diagram

```
 ---------------------
|       CPU          |
 ---------------------
          ⬆
 ---------------------
|   RAM (Fast Memory)  |
 ---------------------
          ⬆
 ---------------------
| Virtual Memory (HDD/SSD) |  (Slower but provides extra space)
 ---------------------
```

When **RAM is full**, the system **temporarily moves some data to Virtual Memory**.
This allows the computer to keep running **without crashing or slowing down significantly**.

How Virtual Memory Works (Step-by-Step)

1 **The CPU stores data in RAM** (Fastest memory).
2 When **RAM gets full**, the operating system (OS) moves some data to Virtual Memory (HDD/SSD).
3 **If that data is needed again,** the OS **moves it back to RAM**.
4 The system keeps **switching data between RAM and Virtual Memory** to **keep multiple programs running smoothly**.

This process is called "Paging."

Advantages & Disadvantages of Virtual Memory

Advantages

Allows larger programs to run even with limited RAM.
Prevents system crashes when RAM is full.
Enables multitasking (running multiple applications at once).

Disadvantages

Slower than RAM (because HDD/SSD is much slower than RAM).
Can cause thrashing (system slowdown due to excessive swapping).
Frequent disk usage may shorten **HDD/SSD lifespan**.

Real-World Example of Virtual Memory

Imagine you are working on a computer with **4GB RAM** while running multiple applications:
Google Chrome (1GB)
Microsoft Word (1GB)
Video Editing Software (3GB needed, but RAM is full)

Since RAM is **only 4GB**, but the system needs **5GB total**, **Virtual Memory is used**.
The **least-used data is moved to the Hard Disk (Virtual Memory)**, so the **CPU can continue working smoothly**.

Virtual Memory extends RAM by using Hard Disk (HDD/SSD) space.
Helps run large programs and multitask efficiently.
Slower than RAM but prevents system crashes.
Excessive use can cause "Thrashing," slowing down the computer.

What is Memory Management Hardware?

Memory Management Hardware is **the part of the computer system that manages memory allocation and access**. It ensures that each program gets the memory it needs while keeping different processes from interfering with each other.

Key Functions:
Allocates memory to different programs and processes.
Protects memory so that one process cannot access another's data.
Handles Virtual Memory by swapping data between RAM and storage.
Improves CPU efficiency by managing memory faster.

Why is Memory Management Hardware Needed?

Problem Without Memory Management Hardware:

- If multiple programs run at the same time, they might **overwrite each other's data**.
- Without proper memory allocation, some programs **may not get enough memory** to run.
- **CPU cannot work efficiently** without organized memory access.

Solution: Using Memory Management Hardware

- The hardware assigns memory to different programs **without conflicts**.
- It ensures **efficient memory usage** and prevents errors.

Components of Memory Management Hardware

Memory management hardware includes several key components:

Component	Function
Memory Management Unit (MMU)	Translates logical addresses (used by programs) into physical addresses (actual memory location).
Base and Limit Registers	Protects memory by setting boundaries for each process.
Translation Lookaside Buffer (TLB)	Speeds up address translation by storing frequently used mappings.
Page Table	Keeps track of how virtual memory is mapped to physical memory.
Segment Table	Used in segmentation memory management to store segment details.

---

1 Memory Management Unit (MMU)

The **Memory Management Unit (MMU)** is **the most important hardware component** for managing memory. It translates **logical addresses (used by programs) into physical addresses (actual memory locations in RAM).**

How MMU Works:

1 A program generates a **logical address**.
2 The MMU converts it into a **physical address**.
3 The CPU then accesses the correct location in RAM.

Example:
If a program requests memory at **logical address 100**, the MMU might translate it to **physical address 5000 in RAM**.

---

2 Base and Limit Registers

- **Base Register**: Stores the **starting address** of a process in RAM.
- **Limit Register**: Defines the **size (boundary)** of the process in memory.

Prevents one program from accessing another program's memory.

Example:
If **Base = 3000** and **Limit = 500**, the process can use memory from **3000 to 3500**.
If it tries to access **memory beyond 3500**, the system blocks it to prevent errors.

---

3 Translation Lookaside Buffer (TLB)

The **TLB is a high-speed cache** that stores recently used address translations.

- Helps the MMU **translate addresses faster**.
- Reduces the number of times the CPU has to check the page table.

Speeds up memory access, making the CPU more efficient.

Example:
If a program frequently accesses memory at **address 2000**, the TLB stores the mapping so the MMU can **quickly retrieve it** next time.

---

4 Page Table

The **Page Table keeps track of how virtual memory is mapped to physical memory.**

- It **divides memory into fixed-sized blocks called pages**.
- Helps in **paging, a method used in Virtual Memory management**.

Prevents processes from interfering with each other.

Example:
A program's **logical address 1000** may be stored in **physical address 5000**.
The **Page Table stores this mapping**, so the CPU can find data correctly.

---

5 Segment Table

- Used in **segmentation memory management** instead of paging.
- **Divides memory into variable-sized segments** based on program needs.
- Each segment has a **Segment Number, Base Address, and Limit Size**.

Example:
Code Segment stores program instructions, **Data Segment** stores variables, etc.

Working of Memory Management Hardware (Step-by-Step)

1A **program requests memory** for execution.
2The **CPU generates a logical address**.
3The **MMU translates** the logical address into a **physical address**.
4The **Base and Limit Registers** check if the access is valid.
5If virtual memory is needed, the **Page Table & TLB** help map virtual pages to physical memory.
6The **CPU retrieves the data** and continues execution.

Real-World Example

Imagine a **library system**:
 The **CPU is the librarian**.
 Memory Management Hardware is the **library catalog**.
 The **Page Table** is the **index** that tells where books (data) are stored.
 The **MMU translates book IDs (logical addresses) into shelf locations (physical addresses).**
 TLB stores frequently accessed book locations to speed up searching.

 This system ensures **efficient book (memory) management** and **prevents confusion** between different users (processes).

Advantages & Disadvantages of Memory Management Hardware

Advantages

Ensures efficient memory allocation to different processes.
Prevents memory corruption by restricting unauthorized access.

Improves CPU performance by using fast memory translation techniques.
Supports Virtual Memory for running large applications.

Disadvantages

Consumes additional system resources (MMU, Page Tables, TLB require space).
Complex address translation may slightly reduce speed if not optimized.
Page Faults & Thrashing can slow down performance if Virtual Memory is overused.

Memory Management Hardware is essential for allocating, protecting, and translating memory efficiently.
The MMU translates logical to physical addresses.
Base and Limit Registers prevent memory corruption.
Page Table and TLB speed up Virtual Memory access.
Prevents system crashes and ensures smooth program execution.