II Semester
Course 4: Digital Logic Design
UNIT- 1

Number Systems

Number systems are a way to express numbers in different notations using different bases. The most common number systems are **Binary**, **Octal**, **Decimal**, and **Hexadecimal**, each with its own base and set of digits. Here's an explanation of each:

---

1. Binary Number System (Base 2)

- **Base:** 2
- **Digits:** 0, 1
- **Explanation:** The binary number system uses only two digits: 0 and 1. Each digit in a binary number is called a **bit** (binary digit). Computers use binary numbers to represent data because they operate using two states (on/off, true/false, etc.).

Example:
Binary number 1011 represents the decimal number 11. To convert from binary to decimal:

1011_2 **$=(1\times2^3)+(0\times2^2)+(1\times2^1)+(1\times2^0)$**

$=8+0+2+1=1110112$

$=(1\times23)+(0\times22)+(1\times21)+(1\times20)$

$=8+0+2+1$

$=11$

- ---

2. Octal Number System (Base 8)

- **Base:** 8
- **Digits:** 0, 1, 2, 3, 4, 5, 6, 7
- **Explanation:** The octal number system uses eight digits. It is often used in computing as a more compact representation of binary numbers. Each octal digit represents exactly three binary digits (bits).
- **Example:**
  Octal number 17 represents the decimal number 15. To convert from octal to decimal:
- $178=(1\times8^1)+(7\times8^0)=8+7=15$

- **Relation to Binary:**
  Each octal digit corresponds to three binary digits:
  - 0 = 000
  - 1 = 001
  - 2 = 010
  - 3 = 011
  - 4 = 100
  - 5 = 101
  - 6 = 110
  - 7 = 111

---

3. Decimal Number System (Base 10)

- **Base:** 10
- **Digits:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Explanation:** The decimal number system is the most commonly used number system in everyday life. It uses ten digits: 0 through 9. Each place value is a power of 10, starting from the rightmost digit (which is the least significant digit).
- **Example:**
  Decimal number 573 represents:
- $573_{10}=(5\times10^{2})+(7\times10^{1})+(3\times10^{0})$

  $=500+70+3=573$
- **Relation to Binary and Octal:**
  Decimal numbers are often converted into binary or octal for use in computing, as these systems are more efficient for binary computation.

---

4. Hexadecimal Number System (Base 16)

- **Base:** 16
- **Digits:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (10), B (11), C (12), D (13), E (14), F (15)
- **Explanation:** The hexadecimal number system uses sixteen digits: 0 to 9 for values 0 to 9, and A to F for values 10 to 15. Hexadecimal is commonly used in computing because it is more compact than binary and easier to read. Each hexadecimal digit represents exactly four binary digits (bits).
- **Example:**
  Hexadecimal number 2F represents the decimal number 47. To convert from hexadecimal to decimal:

- $2F_{16}=(2\times16^1)+(15\times16^0)$
- $=32+15=47$
- **Relation to Binary:**
  Each hexadecimal digit corresponds to four binary digits:
    - 0 = 0000
    - 1 = 0001
    - 2 = 0010
    - 3 = 0011
    - 4 = 0100
    - 5 = 0101
    - 6 = 0110
    - 7 = 0111
    - 8 = 1000
    - 9 = 1001
    - A = 1010
    - B = 1011
    - C = 1100
    - D = 1101
    - E = 1110
    - F = 1111

Conversion between Number Systems

- **Binary to Decimal:** Multiply each bit by the corresponding power of 2 and sum them up.
- **Decimal to Binary:** Divide the decimal number by 2 and record the remainder. Continue dividing the quotient by 2 until it becomes 0. The binary representation is the sequence of remainders.
- **Octal to Decimal:** Multiply each octal digit by the corresponding power of 8 and sum them up.
- **Decimal to Octal:** Divide the decimal number by 8 and record the remainder. Continue dividing the quotient by 8 until it becomes 0. The octal representation is the sequence of remainders.
- **Hexadecimal to Decimal:** Multiply each hexadecimal digit by the corresponding power of 16 and sum them up.

- **Decimal to Hexadecimal:** Divide the decimal number by 16 and record the remainder. Continue dividing the quotient by 16 until it becomes 0. The hexadecimal representation is the sequence of remainders.

Applications of Number Systems in Computing

- **Binary:** Used for internal data representation in computers, since computer processors use binary logic.
- **Octal:** Sometimes used as a more compact representation of binary data in older systems and hardware.
- **Decimal:** Commonly used by humans in everyday life and by applications requiring human-readable numbers.
- **Hexadecimal:** Often used in debugging, memory addressing, and low-level programming due to its compactness and ease of conversion to/from binary.

Conversion of Numbers from One Radix to Another

The conversion between different number systems or **radices** (the base of the number system) is a common task in computing. Here are the methods to convert numbers between various bases like **binary**, **octal**, **decimal**, and **hexadecimal**.

1. Binary to Decimal Conversion

To convert a binary number to decimal, we multiply each bit (starting from the right) by the corresponding power of 2 and then sum the results.

Steps:

1. Start from the rightmost digit (least significant bit).
2. Multiply each digit by the power of 2 corresponding to its position.
3. Sum all the results to get the decimal value.

Example:

Convert 1011 (binary) to decimal.

$1011_2=(1\times2^3)+(0\times2^2)+(1\times2^1)+(1\times2^0)$

$1011_2=(1\times2^3)+(0\times2^2)+(1\times2^1)+(1\times2^0)$

$=8+0+2+1=11$

$=8+0+2+1=11$

So, 1011 in binary equals 11 in decimal.

2. Decimal to Binary Conversion

To convert a decimal number to binary, divide the decimal number by 2 repeatedly, keeping track of the remainders. The binary representation is the sequence of remainders, read from bottom to top.

Steps:

1. Divide the decimal number by 2.
2. Record the remainder.
3. Continue dividing the quotient by 2 until the quotient is 0.
4. The binary representation is the sequence of remainders in reverse order.

Example:

Convert 13 (decimal) to binary.

13÷2=6 remainder 1

6÷2=3 remainder 0

3÷2=1 remainder 1

1÷2=0 remainder 1

Reading the remainders from bottom to top, we get 1101 in binary.

So, 13 in decimal equals 1101 in binary.

3. Decimal to Octal Conversion

To convert a decimal number to octal, divide the decimal number by 8 repeatedly, keeping track of the remainders. The octal representation is the sequence of remainders, read from bottom to top.

Steps:

1. Divide the decimal number by 8.
2. Record the remainder.
3. Continue dividing the quotient by 8 until the quotient is 0.
4. The octal representation is the sequence of remainders in reverse order.

Example:

Convert 56 (decimal) to octal.

56÷8=7 remainder 0

7÷8=0 remainder 7

Reading the remainders from bottom to top, we get 70 in octal.

So, 56 in decimal equals 70 in octal.

4. Octal to Decimal Conversion

To convert an octal number to decimal, multiply each digit (starting from the right) by the corresponding power of 8 and then sum the results.

Steps:

1. Start from the rightmost digit of the octal number.
2. Multiply each digit by the power of 8 corresponding to its position.
3. Sum all the results to get the decimal value.

Example:

Convert 52 (octal) to decimal.

$52_8=(5\times8^1)+(2\times8^0)$

$=(5\times8)+(2\times1)$

$=40+2=42$

So, 52 in octal equals 42 in decimal.

5. Binary to Hexadecimal Conversion

To convert a binary number to hexadecimal, group the binary number into sets of four digits, starting from the right. If the number of digits is not a multiple of 4, pad with leading zeros. Then, convert each group of four binary digits to the corresponding hexadecimal digit.

Steps:

1. Group the binary digits into sets of four, starting from the right.
2. Convert each group of four binary digits into a hexadecimal digit.
3. Combine all the hexadecimal digits to form the final result.

Example:

Convert 11010111 (binary) to hexadecimal.

Group the binary digits into sets of four: 1101 0111

- 1101 in binary = D in hexadecimal
- 0111 in binary = 7 in hexadecimal

So, 11010111 in binary equals D7 in hexadecimal.

6. Hexadecimal to Binary Conversion

To convert a hexadecimal number to binary, replace each hexadecimal digit with its equivalent 4-bit binary representation.

Steps:

1. For each hexadecimal digit, find its 4-bit binary equivalent.
2. Concatenate all the 4-bit binary groups to form the final binary number.

Example:

Convert 2F (hexadecimal) to binary.

- 2 in hexadecimal = 0010 in binary
- F in hexadecimal = 1111 in binary

So, 2F in hexadecimal equals 00101111 in binary.

7. Decimal to Hexadecimal Conversion

To convert a decimal number to hexadecimal, divide the decimal number by 16 repeatedly, keeping track of the remainders. The hexadecimal representation is the sequence of remainders, read from bottom to top.

Steps:

1. Divide the decimal number by 16.
2. Record the remainder. If the remainder is greater than 9, use the corresponding hexadecimal digit (A = 10, B = 11, ..., F = 15).
3. Continue dividing the quotient by 16 until the quotient is 0.
4. The hexadecimal representation is the sequence of remainders in reverse order.

Example:

Convert 254 (decimal) to hexadecimal.

254÷16=15 remainder 14 (which is E)

15÷16=0 remainder 15 (which is *F*)

Reading the remainders from bottom to top, we get FE in hexadecimal.

So, 254 in decimal equals FE in hexadecimal.

8. Hexadecimal to Decimal Conversion

To convert a hexadecimal number to decimal, multiply each digit (starting from the right) by the corresponding power of 16 and then sum the results.

Steps:

1. Start from the rightmost digit of the hexadecimal number.
2. Multiply each digit by the power of 16 corresponding to its position.

3. Sum all the results to get the decimal value.

Example:

Convert 2F (hexadecimal) to decimal.

$2F_{16}=(2\times16^{1})+(15\times16^{0})$

$=32+15=47$

So, 2F in hexadecimal equals 47 in decimal.

Summary of Conversion Methods:

- **Binary ↔ Decimal**: Convert by powers of 2.
- **Decimal ↔ Octal**: Convert by dividing by 8.
- **Octal ↔ Decimal**: Convert by powers of 8.
- **Binary ↔ Hexadecimal**: Group binary into sets of four and convert each set.
- **Hexadecimal ↔ Binary**: Convert each hexadecimal digit to its 4-bit binary equivalent.

These conversions are essential for programming, debugging, and working with low-level systems in computing.

Conversion of Numbers from One Radix to R's Complement

When working with **number systems**, especially **binary, octal, and hexadecimal**, complements are used to represent negative numbers and perform subtraction using addition. The **R's complement** is a specific form of complement representation in positional number systems.

What is R's Complement?

The **R's complement** (or **Radix complement**) of a number is the complement with respect to its base RR. For a number system with radix RR, the R's complement of a number NN is defined as:

$$R^n - N$$

Where:

- R = Base of the number system (e.g., 2 for binary, 8 for octal, 10 for decimal, 16 for hexadecimal)
- n = Total number of digits in the number
- N = Given number

Steps to Find R's Complement

1. **Find the Total Digits**: Determine the total number of digits n based on the number format.
2. **Calculate R^n**: Raise the radix RR to the power n.
3. **Subtract the Number**: Subtract the given number N from R^n
4. **Verify**: Ensure proper padding with leading zeros if required.

Examples of R's Complement

1. R's Complement in Binary (Base 2):

Example: Find the **2's complement** of 1101_2

Steps:

1. Total digits n=4
2. Calculate $2^4=16^2$
3. Subtract 1101_2=13 (in decimal) from 16

   16−13=3
4. Convert 3 back to binary: $3=0011_2$

Answer: The **2's complement** of $1101_2$is 0011_2

---

2. R's Complement in Decimal (Base 10):

Example: Find the **10's complement** of 234

Steps:

1. Total digits n=3
2. Calculate $10^3=1000$
3. Subtract 234 from 1000

   1000−234=766

Answer: The **10's complement** of 234 is 766

---

3. R's Complement in Octal (Base 8):

Example: Find the **8's complement** of 345_8

Steps:

1. Total digits n=3
2. Calculate $8^3=512$.
3. Subtract 345_8 (in decimal, $3\cdot64+4\cdot8+5=229$

   from 512 - 229 = 283
4. Convert 283283 back to octal: $283=433_8$

Answer: The **8's complement** of 345_8 is 433_8

---

4. R's Complement in Hexadecimal (Base 16):

Example: Find the **16's complement** of $1A3_16$

Steps:

1. Total digits n = 3.
2. Calculate $16^3=40961$
3. Subtract $1A3_{16}$ (in decimal, $1\cdot256+10\cdot16+3=419$) from 4096

   $4096-419=3677$
4. Convert 3677 back to hexadecimal: $3677=E5D_{16}$

Answer: The **16's complement** of $1A3_{16}$ is $E5D_{16}$

Key Notes:

1. **R's Complement** is widely used for representing **negative numbers** in computer systems.
2. The **R's complement** is different from the **(R-1)'s complement**, which is another method for complements (e.g., 1's complement for binary).
3. For binary systems, the **2's complement** is most common because it simplifies arithmetic operations.

Conversion of Numbers from One Radix to (R-1)'s Complement

In positional number systems, **(R-1)'s complement** (also called **Diminished Radix Complement**) is used as an alternative to **R's complement** to represent negative numbers and simplify subtraction.

What is (R-1)'s Complement?

For a number system with base RR, the **(R-1)'s complement** of a number NN is obtained by subtracting each digit of NN from the maximum possible digit in that base.

The general formula is:

(R−1)'s complement of N=(Rn−1)−N(R−1)'s complement of$N$=(Rn−1)−N

Where:

- RR: Base of the number system (e.g., 2 for binary, 8 for octal, 10 for decimal, 16 for hexadecimal).
- nn: Total number of digits in the number.
- NN: Given number.

Steps to Find (R-1)'s Complement

1. **Find the Maximum Digit**: The maximum digit in base RR is R−1R−1.
2. **Subtract Each Digit**: Subtract each digit of the number from R−1R−1.
3. **Verify**: Ensure proper padding with leading zeros if required.

Examples of (R-1)'s Complement

1. (R-1)'s Complement in Binary (Base 2):

The maximum digit in binary is 11 (i.e., 2−1=1).

Example: Find the **1's complement** of 1101_2

Steps:

1. Subtract each digit from 11:
    - 1→0
    - 1→0
    - 0→1
    - 1→0
2. Result: 0010_2

Answer: The **1's complement** of 1101_2 is 0010_2.

2. (R-1)'s Complement in Decimal (Base 10):

The maximum digit in base 10 is 9 (i.e., 10−1=9)

Example: Find the **9's complement** of

Steps:

1. Subtract each digit from 99:
    - 9−2=7
    - 9−3=6
    - 9−4=5
2. Result: 765

Answer: The **9's complement** of 234 is 765.

3. (R-1)'s Complement in Octal (Base 8):

The maximum digit in base 8 is 7 (i.e., 8−1=7).

Example: Find the **7's complement** of 345_8.

Steps:

1. Subtract each digit from 7:
    - 7−3=4
    - 7−4=3
    - 7−5=2
2. Result: 432_8

Answer: The **7's complement** of 345_8 is 432_8.

4. (R-1)'s Complement in Hexadecimal (Base 16):

The maximum digit in base 16 is F (i.e., 16−1=15).

Example: Find the **F's complement** of $1A3_{16}$.

Steps:

1. Subtract each digit from F*F*:
    - F−1=E
    - F−A=5 (A = 10 in decimal)

- F−3=C

2. Result: $E5C_{16}$

Answer: The **F's complement** of $1A3_{16}$ is $E5C_{16}$

Key Differences Between R's Complement and (R-1)'s Complement

Feature	**R's Complement**	**(R-1)'s Complement**
Definition	Subtract the number from R^n	Subtract the number from R^n-1
Final Adjustment	Requires adding 1 to perform subtraction	Direct subtraction without adding 1
Common Examples	2's complement, 10's complement	1's complement, 9's complement
Arithmetic Operations	Used for signed numbers & subtraction	Less efficient for arithmetic operations

Signed Binary Numbers

In binary number systems, **signed binary numbers** are used to represent both positive and negative integers. Since binary systems only use 0 and 1, a method is needed to distinguish between positive and negative numbers. This is typically done using **sign bits** or **complement systems**.

Concept of Signed Binary Numbers

In a signed binary number:

- The **most significant bit (MSB)** acts as the **sign bit**:
    - 0 represents a **positive** number.
    - 1 represents a **negative** number.
- The remaining bits are used to represent the magnitude (absolute value) of the number.

For example, in an **8-bit system**:

- 0110 0101 → Positive number (MSB = 0).
- 1010 0101 → Negative number (MSB = 1).

Methods to Represent Signed Binary Numbers

There are three common methods to represent signed binary numbers:

1. **Sign and Magnitude Representation**

2. **1's Complement Representation**
3. **2's Complement Representation**

---

1. Sign and Magnitude Representation

In this method:

- The **MSB** is the sign bit:
    - 0 → Positive
    - 1 → Negative
- The remaining n−1n-1 bits represent the magnitude.

For example, in a 4-bit system:

- +5 → 0101
- -5 → 1101

Advantages:

- Simple representation.

Disadvantages:

- Two representations for zero: 0000 (+0) and 1000 (-0).
- Arithmetic operations (addition, subtraction) are complex.

---

2. 1's Complement Representation

In this method:

- Positive numbers are represented as usual in binary.
- Negative numbers are represented by **inverting all bits** (finding the **1's complement**) of the corresponding positive number.

Steps to find 1's complement:

1. Represent the positive number in binary.
2. Invert all bits (replace 0 with 1 and 1 with 0).

For example, in a 4-bit system:

- +5 → 0101
- -5 → 1's complement of 0101 → 1010

Advantages:

- Easier to compute than sign and magnitude.

Disadvantages:

- Two representations for zero: 0000 (+0) and 1111 (-0).
- Arithmetic operations require handling a carry bit.

---

3. 2's Complement Representation

The **2's complement** is the most widely used method for representing signed binary numbers.

Key Features:

- Positive numbers are represented as usual in binary.
- Negative numbers are represented by:
    1. Taking the **1's complement** of the positive number.
    2. Adding 1 to the result.

Steps to find 2's complement:

1. Write the binary form of the positive number.
2. Find the 1's complement (invert all bits).
3. Add 1 to the result.

For example, in a 4-bit system:

- +5 → 0101
- −5 →
    1. 1's complement of 0101 → 1010
    2. Add 1 → 1011

Advantages:

- Only one representation for zero (0000).
- Arithmetic operations (addition, subtraction) are simpler.

Disadvantages:

- None; widely adopted due to efficiency.

Range of Signed Binary Numbers

For an nn-bit system, the range of signed binary numbers is:

-2^{n-1} to $(2^{n-1} - 1)$

- **Example for 4-bit signed binary numbers**:
    - Range = -2^{4-1} to $(2^{4-1} - 1)$ → −8 to +7

- Representations:
  - 0111 → +7
  - 1000 → −8

Comparison of Representations

Feature	Sign & Magnitude	1's Complement	2's Complement
Zero Representation	Two (+0 and -0)	Two (+0 and -0)	One (+0 only)
Ease of Arithmetic	Complex	Moderate (carry bit)	Simple and efficient
Used In Practice	Rarely used	Rarely used	Widely used

Summary

- **Signed binary numbers** use the MSB to indicate the sign (0 for positive and 1 for negative).
- The **2's complement** method is the most common due to its simplicity and efficiency in arithmetic operations.
- Signed binary numbers are crucial in computer systems for representing positive and negative integers.

Addition and Subtraction of Unsigned and Signed Numbers

Addition and subtraction operations in binary differ slightly when dealing with **unsigned numbers** and **signed numbers** (represented using methods like **2's complement**). Let's discuss both concepts in detail.

---

1. Unsigned Numbers

Unsigned numbers can only represent non-negative values. All bits in an unsigned binary number are used to represent the magnitude.

Addition of Unsigned Numbers

1. Binary addition follows the same rules as decimal addition:
    - 0+0=0
    - 0+1=1
    - 1+0=1
    - 1+1=1 (carry 1 to the next higher bit).
2. If a carry exceeds the fixed number of bits, **overflow** occurs.

Example (4-bit unsigned numbers):

- Add 6 (0110) and 5 (0101):

```
  0110
+ 0101
--------
  1011   → 11 (Correct result)
```

Overflow Example:

- Add 9 (1001) and 7 (0111):

```
  1001
+ 0111
--------
 10000  → Overflow occurs (5-bit result, but only 4 bits available).
```

Subtraction of Unsigned Numbers

Subtraction is performed using the **borrow method**, similar to decimal subtraction:

- 0−0=0
- 1−0=1

- 1−1=0
- 0−1=1 with a borrow from the next higher bit.

Alternatively, subtraction can be performed using the **2's complement method**:

1. Take the 2's complement of the number to be subtracted.
2. Add it to the first number.
3. If there is a carry-out, discard it.

Example (4-bit unsigned numbers):

- Subtract 5 (0101) from 6 (0110):
    - 2's complement of 55 → Invert bits (1010), add 1 → 1011.
    - Add 66 (0110) and 55's complement (1011):

```
  0110
+ 1011
--------
 10001   → Discard carry → Result = `0001` (1).
```

---

2. Signed Numbers

Signed numbers can represent both positive and negative integers. The **2's complement** method is most commonly used for signed binary arithmetic.

Addition of Signed Numbers

When adding signed numbers:

1. Represent the numbers in 2's complement form.
2. Add the numbers as if they were unsigned.
3. If the result exceeds the bit limit, **overflow** occurs.

Rules for Detection of Overflow:

- **For addition**:
    - If two positive numbers are added and the result is negative → Overflow.
    - If two negative numbers are added and the result is positive → Overflow.

Example (4-bit signed numbers):

- Add +5 (0101) and −3 (1101):
    - Step 1: Represent −3-in 2's complement → 1101.

- Step 2: Add:

```
 0101   (5)
+ 1101   (-3)
--------
 0010   → Result = `0010` → +2 (Correct).
```

- Add −5 (1011) and −3 (1101):
  - Step 1: Add directly:

```
 1011   (-5)
+ 1101   (-3)
--------
 10000   → Overflow detected → Ignore carry → `1000` → -8.
```

Subtraction of Signed Numbers

Subtraction in signed numbers is performed using the **2's complement method**:

1. Take the 2's complement of the number to be subtracted.
2. Add it to the first number.
3. Check for overflow.

Example (4-bit signed numbers):

- Subtract −3 (1101) from +5 (0101):
  - Step 1: 2's complement of −3 → 0011.
  - Step 2: Add:

```
 0101   (5)
+ 0011   (2's complement of -3)
--------
 1000   → Result = `1000` → -8 (Overflow occurs in 4 bits).
```

Summary Table

Operation	**Unsigned Numbers**	**Signed Numbers**
Addition	Add directly; check for carry/overflow.	Add as unsigned; check overflow conditions.

Operation	Unsigned Numbers	Signed Numbers
Subtraction	Use borrow method or 2's complement.	Use 2's complement method.
Overflow Check	Carry out of MSB indicates overflow.	Overflow occurs when sign bit changes unexpectedly.

Key Notes

1. **Unsigned numbers** only represent non-negative values, and overflow occurs when the result exceeds the bit limit.
2. **Signed numbers** use the MSB as the sign bit (0 for positive, 1 for negative), and the 2's complement method simplifies arithmetic operations.
3. Overflow in signed arithmetic is detected based on the **sign bit behavior**.

This distinction is critical when performing arithmetic operations in computer systems to avoid errors.

Weighted and Unweighted Codes

Number codes are used to represent numerical values in digital systems. They are classified into **weighted** and **unweighted codes** based on whether the positional values contribute to the total number.

---

1. Weighted Codes

- **Definition**: Weighted codes are positional number systems where each digit position has a specific weight assigned to it. The value of the number is determined by multiplying each digit by its positional weight and summing up the results.
- **Example**: Binary, Decimal, Octal, Hexadecimal, and BCD codes.

Key Features of Weighted Codes:

1. Each position in the number has a fixed weight.
2. The value of the number is calculated as:

$$\text{Value} = \sum(\text{digit} \times \text{weight of the position})$$

   It is widely used in arithmetic operations and logical circuits.

Examples of Weighted Codes:

1. **Binary Number System**:
    - A pure positional system.

- Each position has a weight of 2n2^n, where nn is the position from the right, starting at 0.
  Example: Binary number 1011

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 8 + 0 + 2 + 1 = 11$$

2. **BCD (Binary-Coded Decimal)**:
   - A 4-bit code representing each decimal digit (0–9).
   - Weights for each position are 8,4,2,1.
     Example: Decimal 5 → BCD 0101.
3. **2421 Code**:
   - A weighted code with position weights 2,4,2,12, 4, 2, 1.
     Example: Decimal 5 → 1100 in 2421 code.
4. **Excess-3 Code**:
   - Derived from BCD by adding 3 to each decimal digit.
   - Weights are still positional, though shifted.
     Example: Decimal 2 → Add 3 → 5 → Binary 0101.

---

2. Unweighted Codes

- **Definition**: Unweighted codes do not have positional weights assigned to each digit. The position of the digit does not affect its overall value.
- **Example**: Gray Code, ASCII Code, Excess-3 Code (although derived from weighted systems), and Error-Detecting Codes.

Key Features of Unweighted Codes:

1. Digits do not carry positional weight.
2. These codes are often used for special purposes like error detection, data transmission, and communication.

Examples of Unweighted Codes:

1. **Gray Code**:
   - A **non-weighted** code where two consecutive numbers differ by only one bit.
   - Used in analog-to-digital converters to reduce errors.

- **Example**:

Decimal	Binary	Gray Code
0	000	000
1	001	001
2	010	011
3	011	010

2. **ASCII Code**:
    - American Standard Code for Information Interchange.
    - Represents characters (letters, numbers, symbols) using 7 or 8 bits.
      Example:
    - 'A' → 01000001
    - 'B' → 01000010
3. **Error-Detecting Codes**:
    - Used to detect errors in digital data transmission.
    - Examples include **parity codes** and **Hamming codes**.

Comparison Between Weighted and Unweighted Codes

Aspect	**Weighted Codes**	**Unweighted Codes**
Weighting	Each digit position has a weight.	No positional weights assigned.
Examples	Binary, BCD, 2421, Excess-3	Gray Code, ASCII, Error-Detecting Codes
Usage	Arithmetic and logical operations.	Special purposes like error detection.
Value Calculation	Multiply digit by its weight.	Not based on position weight.
Error Susceptibility	Lower in arithmetic operations.	Reduced errors in transmissions (e.g., Gray).

Summary

- **Weighted Codes**: Each position has a specific weight, and the value is calculated based on this weight (e.g., Binary, BCD, 2421 Code).
- **Unweighted Codes**: Positional weights are absent; primarily used for special applications like Gray Code, ASCII, and error detection.

These codes are fundamental to digital systems, enabling numerical representation, efficient data transmission, and error control.

UNIT – II

Logic Gates and Boolean Algebra

Logic Gates and Boolean Algebra

Logic gates are the building blocks of digital circuits. They perform logical operations on one or more inputs to produce a single output. Boolean algebra is a branch of mathematics that simplifies logic circuits.

Basic Logic Gates

1. NOT Gate

- **Symbol**: ! or ¬
- **Operation**: Inverts the input (logical complement).
- **Truth Table**:

Input (A)	Output (Y = A')
0	1
1	0

- **Logic Symbol**:
- A -->|¬|--> Y

---

2. AND Gate

- **Symbol**: .
- **Operation**: Outputs 1 only if **all inputs** are 1.
- **Boolean Expression**: Y=A·BY = A \cdot B
- **Truth Table**:

A	B	Y (A.B)
0	0	0
0	1	0
1	0	0
1	1	1

- **Logic Symbol**:
- A --|&|--

- |--> Y
- B --|&|--

---

3. OR Gate

- **Symbol**: +
- **Operation**: Outputs 1 if **any input** is 1.
- **Boolean Expression**: Y=A+BY = A + B
- **Truth Table**:

A	B	Y (A+B)
0	0	0
0	1	1
1	0	1
1	1	1

- **Logic Symbol**:
- A --|≥|--
- |--> Y
- B --|≥|--

Universal Gates

Universal gates can perform the function of any basic gate (AND, OR, NOT).

4. NAND Gate

- **Symbol**: AND gate + NOT
- **Operation**: Outputs the complement of AND operation.
- **Boolean Expression**: Y=(A·B)’Y = (A \cdot B)'
- **Truth Table**:

A	B	Y = (A.B)'
0	0	1
0	1	1

A	B	Y = (A.B)'
1	0	1
1	1	0

- **Logic Symbol**:
- A --|&|o--
- |--> Y
- B --|&|o--

5. NOR Gate

- **Symbol**: OR gate + NOT
- **Operation**: Outputs the complement of OR operation.
- **Boolean Expression**: Y=(A+B)’Y = (A + B)'
- **Truth Table**:

A	B	Y = (A+B)'
0	0	1
0	1	0
1	0	0
1	1	0

- **Logic Symbol**:
- A --|≥|o--
- |--> Y
- B --|≥|o--

Exclusive Gates

6. XOR (Exclusive OR) Gate

- **Symbol**: ⊕\oplus
- **Operation**: Outputs 1 only when the inputs are different.
- **Boolean Expression**: Y=A⊕B=A’B+AB’Y = A \oplus B = A'B + AB'
-

Truth Table:

A	B	Y = A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

- **Logic Symbol**:
- A --|=1|--
- |--> Y
- B --|=1|--

7. XNOR (Exclusive NOR) Gate

- **Symbol**: ⊙\odot
- **Operation**: Outputs 1 only when the inputs are the same.
- **Boolean Expression**: Y=(A⊕B)’=A’B’+ABY = (A \oplus B)' = A'B' + AB
- **Truth Table**:

A	B	Y = A⊙B
0	0	1
0	1	0
1	0	0
1	1	1

- **Logic Symbol**:
- A --|=1o|--
- |--> Y
- B --|=1o|--

Boolean Algebra Basics

Boolean algebra is used to simplify logic expressions and reduce circuit complexity.

- Basic Laws:
  1. Identity Law: $A + 0 = A, A \cdot 1 = A$
  2. Null Law: $A + 1 = 1, A \cdot 0 = 0$
  3. Complement Law: $A + A' = 1, A \cdot A' = 0$
  4. Idempotent Law: $A + A = A, A \cdot A = A$
  5. Commutative Law: $A + B = B + A, A \cdot B = B \cdot A$
  6. Associative Law: $A + (B + C) = (A + B) + C, A \cdot (B \cdot C) = (A \cdot B) \cdot C$
  7. Distributive Law: $A \cdot (B + C) = A \cdot B + A \cdot C$

Applications of Logic Gates

1. **Arithmetic Circuits**: Adders, Subtractors.
2. **Data Transmission**: Error detection using XOR gates.
3. **Memory Units**: Flip-flops and registers.
4. **Control Systems**: Decoders, Multiplexers.

Summary

Logic gates perform fundamental logical operations using Boolean algebra:

- **Basic Gates**: NOT, AND, OR.
- **Universal Gates**: NAND, NOR.
- **Special Gates**: XOR, XNOR.

Boolean algebra helps simplify expressions and optimize digital circuits.

Boolean Laws and Theorems

Boolean algebra is a branch of algebra that deals with binary values (0 and 1) and logical operations. It is fundamental in the design and simplification of digital circuits. Below are the basic laws, theorems, and properties of Boolean algebra:

---

1. Basic Boolean Laws

a. Identity Law

- AND Operation: $A \cdot 1 = A$
- OR Operation: $A + 0 = A$

Explanation:

- If you AND a variable with 1, the result is the variable itself.
- If you OR a variable with 0, the result is the variable itself.

---

b. Null (Dominance) Law

- AND Operation: $A \cdot 0 = 0$
- OR Operation: $A + 1 = 1$

Explanation:

- If you AND a variable with 0, the result is always 0.
- If you OR a variable with 1, the result is always 1.

---

c. Idempotent Law

- AND Operation: $A \cdot A = A$
- OR Operation: $A + A = A$

Explanation:

- ANDing or ORing a variable with itself does not change its value.

---

d. Complement Law

- AND Operation: $A \cdot A' = 0$
- OR Operation: $A + A' = 1$

Explanation:

- A variable ANDed with its complement (NOT) results in 0.
- A variable ORed with its complement results in 1.

---

e. Involution Law

- $(A')' = A$

Explanation:

- The complement of the complement of a variable is the variable itself.

---

2. Commutative Law

- AND Operation: $A \cdot B = B \cdot A$
- OR Operation: $A + B = B + A$

Explanation:

- The order of variables does not matter in AND or OR operations.

---

3. Associative Law

- AND Operation: $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- OR Operation: $A + (B + C) = (A + B) + C$

Explanation:

- Grouping of variables does not affect the result of AND or OR operations.

---

4. Distributive Law

- AND Distributive Over OR: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$
- OR Distributive Over AND: $A + (B \cdot C) = (A + B) \cdot (A + C)$

Explanation:

- AND can be distributed over OR, and OR can be distributed over AND.

---

5. Absorption Law

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

Explanation:

- A variable absorbs its operation with another variable if the result can be simplified.

---

6. De Morgan's Theorems

These theorems are used to simplify expressions involving complements.

1. **First Theorem**: $(A \cdot B)' = A' + B'$
   - The complement of an AND operation equals the OR of the complements.
2. **Second Theorem**: $(A + B)' = A' \cdot B'$
   - The complement of an OR operation equals the AND of the complements.

Example:

- If $A = 0$ and $B = 1$, then:
  $(A \cdot B)' = (0 \cdot 1)' = 0' + 1' = 1 + 0 = 1.$

---

7. Consensus Theorem

- $A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C$

Explanation:

- The redundant term $B \cdot C$ can be eliminated.

8. Duality Principle

- Every Boolean expression has a **dual** where:
  - Replace AND (·) with OR (+) and vice versa.
  - Replace 0 with 1 and 1 with 0.

Example:

- Original: $A + (B \cdot C) = (A + B) \cdot (A + C)$
- Dual: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$.

---

9. Redundancy Theorem

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

Explanation:

- Redundant terms can be removed without changing the output.

Summary of Important Boolean Laws

Law	AND Operation	OR Operation
Identity Law	$A \cdot 1 = A$	$A + 0 = A$
Null Law	$A \cdot 0 = 0$	$A + 1 = 1$
Idempotent Law	$A \cdot A = A$	$A + A = A$
Complement Law	$A \cdot A' = 0$	$A + A' = 1$
Commutative Law	$A \cdot B = B \cdot A$	$A + B = B + A$
Associative Law	$A \cdot (B \cdot C)$	$A + (B + C)$ = $(A + B) + C$
Distributive Law	$A \cdot (B + C)$	$A + (B \cdot C)$
De Morgan's Theorems	$(A \cdot B)' = A' + B'$	$(A + B)' = A' \cdot B'$

Applications of Boolean Algebra

1. Simplification of logic circuits.
2. Designing combinational circuits like adders, subtractors, and multiplexers.
3. Development of digital systems and processors.
4. Error detection in data transmission.

Complement of a Logic Function

The complement of a logic function F(A,B,C,…) is the function that produces the opposite output for every input combination. It is obtained by negating the function F.

How to Find the Complement:

1. Write the original function in terms of its Boolean expression.
2. Apply the **NOT** operation to the entire function, which can be simplified using Boolean algebra if needed.

Example:

Given $F(A, B) = A + B$ (OR operation):

- Complement: $F'(A, B) = \overline{A + B}$, which simplifies to $\overline{A} \cdot \overline{B}$ (De Morgan's Law).

Given $F(A, B) = A \cdot B$ (AND operation):

- Complement: $F'(A, B) = \overline{A \cdot B}$, which simplifies to $\overline{A} + \overline{B}$ (De Morgan's Law).

Key Differences

Aspect	Complement	Dual
Definition	Negation of the entire logic function	Swap of AND/OR operations
How to Obtain	Negate the function (F')	Swap operators while keeping variables intact
Impact	Changes the truth table output	Does not change the truth table output

Would you like a step-by-step example of calculating both the complement and dual for a specific function?

CANONICAL AND STANDARD FORMS

Canonical Forms

Canonical forms are the most detailed and complete representations of a Boolean function. They are written to include all variables in either complemented or uncomplemented form for every term.

Types of Canonical Forms:

1. **Sum of Minterms (Canonical Sum of Products - SOP):**
   - A Boolean function is expressed as a sum (OR) of **minterms**.
   - Each minterm corresponds to a specific row in the truth table where the function evaluates to 1.
   - A minterm is a product (AND) of all variables in the function, either in complemented or uncomplemented form, depending on whether the variable is 0 or 1 in that truth table row.

   Example: For a function $F(A, B)$, if $F = 1$ when $(A, B) = (1, 0)$ and $(A, B) = (0, 1)$,
   - Canonical SOP: $F(A, B) = \overline{A}B + A\overline{B}$.
2. **Product of Maxterms (Canonical Product of Sums - POS):**
   - A Boolean function is expressed as a product (AND) of **maxterms**.
   - Each maxterm corresponds to a specific row in the truth table where the function evaluates to 0.
   - A maxterm is a sum (OR) of all variables, either in complemented or uncomplemented form, depending on whether the variable is 1 or 0 in that truth table row.

   Example: For the same $F(A, B)$ as above,
   - Canonical POS: $F(A, B) = (A + B)(\overline{A} + \overline{B})$.

Standard Forms

Standard forms are less restrictive and don't require the inclusion of all variables in each term. They represent a Boolean function in either **Sum of Products (SOP)** or **Product of Sums (POS)** format, but without the need to use all variables in every term.

Types of Standard Forms:

1. **Sum of Products (SOP):**
   - A Boolean function is expressed as a sum (OR) of product (AND) terms.
   - Not all variables need to appear in every term.

   Example: $F(A, B) = A + AB$.
2. **Product of Sums (POS):**
   - A Boolean function is expressed as a product (AND) of sum (OR) terms.
   - Not all variables need to appear in every term.

   Example: $F(A, B) = (A + B)(B + C)$.

Key Differences Between Canonical and Standard Forms

Aspect	Canonical Form	Standard Form
Completeness	Includes all variables in every term	May omit some variables in terms
Representation Types	Sum of minterms (SOP) or Product of maxterms (POS)	Sum of Products (SOP) or Product of Sums (POS)
Dependency on Truth Table	Directly derived from truth table	May or may not directly align with the truth table
Complexity	More detailed, more complex	Simpler and more concise

---

When to Use Each Form

- **Canonical Forms**: Useful for exact representation, analysis, and deriving the truth table directly.
- **Standard Forms**: Useful for simplified circuit design and practical implementation.

Two level realization of logic functions using universal gates

Two-Level Realization of Logic Functions Using Universal Gates

Two-level realization involves implementing a Boolean function with two logic levels:

1. The **first level** performs the AND or OR operations.
2. The **second level** combines the results using OR or AND operations.

With **universal gates** like **NAND** or **NOR**, both levels can be constructed entirely from these gates. Here's how it is done:

Two-Level Realization Using NAND Gates

NAND gates are universal, meaning they can perform AND, OR, and NOT operations by appropriate combinations.

Steps to Implement Using NAND Gates:

1. **Start with the SOP (Sum of Products) form:**
   - Example: $F(A, B, C) = AB + AC$.
2. **Realize the AND operations (products):**
   - Each product term (e.g., AB) is implemented using a NAND gate followed by another NAND gate to invert its output:
     - $AB = \text{NAND}(\text{NAND}(A, B), \text{NAND}(A, B))$.

3. **Realize the OR operation (sum):**
   - Combine the product terms using NAND gates:
     - $AB + AC = \text{NAND}(\text{NAND}(AB, AB), \text{NAND}(AC, AC))$.
4. **Final Implementation:**
   - The entire function is completed in **two levels** of NAND gates: one for the AND-like operation and another for the OR-like operation.

Example:

Given $F(A, B, C) = AB + AC$:

- First Level: Use two NAND gates to generate AB and AC.
- Second Level: Use another NAND gate to combine AB and AC:

$$F = \text{NAND}(\text{NAND}(\text{NAND}(A, B), \text{NAND}(A, B)), \text{NAND}(\text{NAND}(A, C), \text{NAND}(A, C))).$$

Two-Level Realization Using NOR Gates

NOR gates are also universal and can be used to construct any logic function.

Steps to Implement Using NOR Gates:

1. **Start with the POS (Product of Sums) form:**
   - Example: $F(A, B, C) = (A + B)(A + C)$.
2. **Realize the OR operations (sums):**
   - Each sum term (e.g., $A + B$) is implemented using a NOR gate followed by another NOR gate to invert its output:
     - $A + B = \text{NOR}(\text{NOR}(A, A), \text{NOR}(B, B))$.
3. **Realize the AND operation (product):**
   - Combine the sum terms using NOR gates:
     - $(A + B)(A + C) = \text{NOR}(\text{NOR}(A + B, A + B), \text{NOR}(A + C, A + C))$.
4. **Final Implementation:**
   - The entire function is completed in **two levels** of NOR gates: one for the OR-like operation and another for the AND-like operation.

Example:

Given $F(A, B, C) = (A + B)(A + C)$:

- First Level: Use two NOR gates to generate $A + B$ and $A + C$.
- Second Level: Use another NOR gate to combine $A + B$ and $A + C$:

$$F = \text{NOR}(\text{NOR}(\text{NOR}(A, A), \text{NOR}(B, B)), \text{NOR}(\text{NOR}(A, A), \text{NOR}(C, C))).$$

Comparison of NAND and NOR Two-Level Realization

Aspect	NAND Realization	NOR Realization
Input Function Form	SOP (Sum of Products)	POS (Product of Sums)
First-Level Operation	AND-like operation	OR-like operation
Second-Level Operation	OR-like operation	AND-like operation
Universal Gate Used	NAND gates only	NOR gates only

MINIMIZATIONS OF LOGIC FUNCTIONS (POS AND SOP) USING BOOLEAN THEOREMS

Minimizing logic functions in **Sum of Products (SOP)** or **Product of Sums (POS)** form is a critical step in simplifying Boolean expressions and optimizing digital circuits. Boolean theorems and algebraic laws help achieve this simplification systematically.

Key Boolean Theorems and Laws for Minimization

1. **Identity Law:**
   $A + 0 = A, A \cdot 1 = A$
2. **Null Law:**
   $A + 1 = 1, A \cdot 0 = 0$
3. **Idempotent Law:**
   $A + A = A, A \cdot A = A$
4. **Complement Law:**
   $A + \overline{A} = 1, A \cdot \overline{A} = 0$
5. **Distributive Law:**
   $A \cdot (B + C) = A \cdot B + A \cdot C$
   $A + (B \cdot C) = (A + B) \cdot (A + C)$
6. **Absorption Law:**
   $A + A \cdot B = A, A \cdot (A + B) = A$
7. **De Morgan's Laws:**
   $\overline{A + B} = \overline{A} \cdot \overline{B}, \overline{A \cdot B} = \overline{A} + \overline{B}$
8. **Consensus Theorem:**
   $A \cdot B + \overline{A} \cdot C + B \cdot C = A \cdot B + \overline{A} \cdot C$

Steps to Minimize a Logic Function

1. **Start with the given SOP or POS form.**
2. **Apply Boolean theorems** to simplify the expression:
    - Combine like terms.
    - Eliminate redundant terms using theorems like the Idempotent, Absorption, or Consensus laws.
3. **Look for common factors** (factorization) to further simplify.
4. **Verify the minimized function** using a truth table if needed.

Minimization Example (SOP Form)

Given Function:

$$F(A, B, C) = A \cdot B + A \cdot \overline{B} + \overline{A} \cdot B \cdot C$$

Step-by-Step Simplification:

1. Combine like terms:
    - $A \cdot B + A \cdot \overline{B}$: Apply the Distributive Law.

$$A \cdot (B + \overline{B}) = A \cdot 1 = A$$

2. Substitute into the function:

$$F(A, B, C) = A + \overline{A} \cdot B \cdot C$$

3. Apply the Absorption Law:
    - $A + \overline{A} \cdot B \cdot C = A + B \cdot C$
      (since $A + \overline{A} \cdot X = A + X$).

Final Minimized SOP Form:

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

Minimization Example (POS Form)

Given Function:

$$F(A, B, C) = (A + B)(A + \overline{B})(\overline{A} + C)$$

Step-by-Step Simplification:

1. Expand and simplify:
   - Combine the first two terms using the Distributive Law:
     $$(A + B)(A + \overline{B}) = A + (B \cdot \overline{B}) = A + 0 = A$$
2. Substitute into the function:
   $$F(A, B, C) = (A)(\overline{A} + C)$$
3. Apply the Distributive Law:
   $$A \cdot (\overline{A} + C) = A \cdot C + A \cdot \overline{A} = A \cdot C + 0 = A \cdot C$$

Final Minimized POS Form:

$$F(A, B, C) = A \cdot C$$

Tips for Minimization

1. **Factorize whenever possible** to reduce the number of terms.
2. **Use Consensus Theorem** to eliminate redundant terms.
3. **Apply De Morgan's Laws** when dealing with complements in POS forms.
4. **Cross-check with a Karnaugh map (K-map)** for verification if needed.

K-map (up to four variables)

Karnaugh maps (K-maps) are a graphical method for minimizing Boolean expressions up to four variables. They simplify the process of reducing logic functions to minimal **Sum of Products (SOP)** or **Product of Sums (POS)** forms by grouping adjacent cells representing minterms or maxterms.

Structure of a K-map

1. **Number of Variables and Cells:**
   - A K-map for n variables contains 2^n cells, each corresponding to a minterm (SOP) or maxterm (POS).
   - For 4 variables (A, B, C, D), the K-map has $2^4 = 16$ cells.
2. **Variable Assignments:**
   - The rows and columns are labeled using **Gray Code** to ensure that adjacent cells differ by only one bit.
   - For example, for 4 variables:
     - Rows: $AB = 00, 01, 11, 10$
     - Columns: $CD = 00, 01, 11, 10$

Steps to Minimize Using a K-map

1. Create the K-map:

- Place a **1** in the cells corresponding to the minterms (for SOP) or a **0** for the maxterms (for POS).
- Use the truth table or function representation to map the minterms or maxterms.

2. Group Adjacent 1s or 0s:

- Group **1s** for SOP minimization or **0s** for POS minimization.
- Groups must contain 1,2,4,8….cells (powers of 2).
- Groups can wrap around edges of the K-map.

3. Write the Simplified Expression:

- For SOP: Write each group as a product term with the variables that remain constant within the group.
- For POS: Write each group as a sum term with the variables that remain constant within the group.

Example: SOP Minimization

Given Truth Table:

$$F(A, B, C, D) = \Sigma m(0, 1, 2, 5, 8, 9, 10, 14)$$

K-map Construction:

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

Grouping:

- Group 1: $m(0, 1, 8, 9)$ (4 cells: wraps horizontally in the first row).
- Group 2: $m(2, 10)$ (2 cells).
- Group 3: $m(14)$ (1 cell).

Simplified SOP Expression:

$$F(A, B, C, D) = \overline{B} \cdot \overline{C} + A \cdot \overline{C} + B \cdot C \cdot D$$

Example: POS Minimization

Given Truth Table:

$$F(A, B, C, D) = \Pi M(3, 4, 6, 7, 11, 12, 13, 15)$$

K-map Construction:

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

Grouping:

- Group 1: $M(3, 7, 11, 15)$ (wraps vertically in the last column).
- Group 2: $M(4, 12)$ (2 cells).
- Group 3: $M(6, 13)$ (2 cells).

Simplified POS Expression:

$$F(A, B, C, D) = (A + B + C) \cdot (A + \overline{B} + \overline{D}) \cdot (\overline{A} + B + D)$$

Key Points for K-map Minimization

1. **Wrap-around Groups**:
    - K-maps are cyclic; adjacent edges are considered neighbors for grouping.
2. **Largest Groups First**:
    - Prioritize larger groups to minimize the number of terms.
3. **Overlap is Allowed**:
    - A cell can belong to multiple groups.

Don’t Care Conditions in K-maps

Don’t care conditions are situations where the output of a Boolean function is irrelevant for certain input combinations. These are represented by **X** in truth tables or K-maps and can be used to further simplify logic expressions.

Why Use Don’t Care Conditions?

1. **Flexibility in Simplification**:
   Don’t care conditions allow additional flexibility in grouping 1s (for SOP) or 0s (for POS) in K-maps to achieve greater minimization.
2. **Application Scenarios**:
    - Unused input combinations in digital circuits.
    - Outputs that will never occur due to physical or design constraints.
    - Simplifying state machines with unreachable states.

Steps to Handle Don’t Care Conditions in K-maps

1. Mark Don’t Care Cells as X:
    - Place X in cells corresponding to don’t care conditions.
2. Use X Flexibly:
    - Treat X as 1 when grouping 1s for **SOP** minimization.
    - Treat X as 0 when grouping 0s for **POS** minimization.
    - Leave X ungrouped if it does not help simplify the expression.
3. Simplify the Expression:
    - Write the simplified Boolean expression using the groups.

Example: SOP Minimization with Don't Care

Given Function:

$$F(A, B, C) = \Sigma m(1, 3, 7) + d(0, 2, 5)$$

Here, $d(0, 2, 5)$ are don't care conditions.

K-map Construction:

$AB \backslash C$	0	1
00	X	1
01	X	0
11	X	1
10	0	1

Grouping (Include X to Help Simplify):

1. Group $m(1, 3, 7) + d(0)$ into a group of 4 (wraps horizontally across rows).
2. Group $m(7) + d(5)$ into another group of 2.

Simplified SOP Expression:

$$F(A, B, C) = \overline{B} \cdot C + A \cdot C$$

Example: POS Minimization with Don't Care

Given Function:

$$F(A, B, C) = \Pi M(0, 2, 4) + d(6, 7)$$

Here, $d(6, 7)$ are don't care conditions.

K-map Construction:

$AB \backslash C$	0	1
00	0	X
01	0	0
11	X	1
10	1	1

Grouping (Include X to Help Simplify):

1. Group $M(4) + d(6)$ into a group of 2.
2. Group $M(0) + M(2)$ into another group of 2.

Simplified POS Expression:

$$F(A, B, C) = (A + B) \cdot (\overline{A} + \overline{C})$$

UNIT – III

Combinational Logic Circuits – 1

Design of half adder

Design of a Half Adder

A **Half Adder** is a combinational circuit used to perform the addition of two single-bit binary numbers. It produces two outputs:

1. **Sum** (S): Represents the least significant bit of the addition.
2. **Carry** (C): Represents the carry-out, used in the next higher bit addition in multi-bit operations.

Truth Table of a Half Adder

Input A	Input B	Sum (S)	Carry (C)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Boolean Expressions for Outputs

1. **Sum** (S):
   - The sum is 1 when either A or B is 1, but not both. This is equivalent to an XOR operation:

$$S = A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B$$

2. **Carry** (C):
   - The carry is 1 only when both A and B are 1. This is equivalent to an AND operation:

$$C = A \cdot B$$

Logic Diagram of a Half Adder

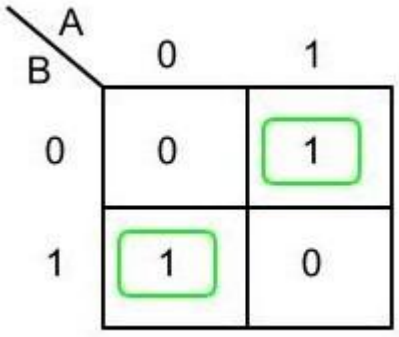
The Half Adder can be implemented using:

1. An XOR gate for the Sum (SSS).
2. An AND gate for the Carry (CCC).

Diagram:

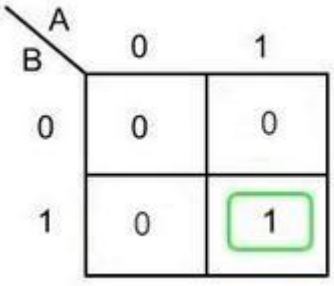
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

For Sum

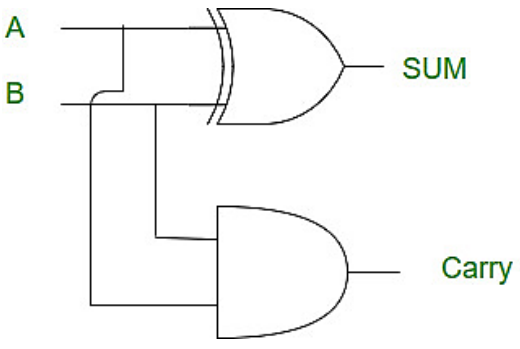


Sum = A XOR B

For Carry



Carry = A AND B



Design Steps

1. **Inputs and Outputs**:
   - Inputs: A, B (two 1-bit binary inputs).
   - Outputs: S (Sum) and C (Carry).
2. **Truth Table**:
   - Use the truth table to derive the expressions for S and C.
3. **Implementation**:
   - Use basic gates (XOR and AND) to implement the Sum and Carry.

Applications of a Half Adder

1. **Building Block**:
   - Used as a building block in constructing **Full Adders** and multi-bit binary adders.
2. **Arithmetic Operations**:
   - Performs addition in simple digital circuits.
3. **ALU Design**:
   - Integrated into Arithmetic Logic Units for computational tasks.

FULL ADDER

Design of a Full Adder

A **Full Adder** is a combinational circuit that performs the addition of three binary bits: two input bits and a carry-in from the previous stage. It produces two outputs:

1. **Sum** (S): The least significant bit of the result.
2. **Carry** (C_{out}): The carry-out, passed to the next stage in multi-bit addition.

Inputs and Outputs

- Inputs:
  - A: First binary input.
  - B: Second binary input.
  - C_{in}: Carry-in from the previous stage.
- Outputs:
  - S: Sum output.
  - C_{out}: Carry output.

Truth Table of a Full Adder

A	B	C_{in}	Sum (S)	Carry (C_{out})
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

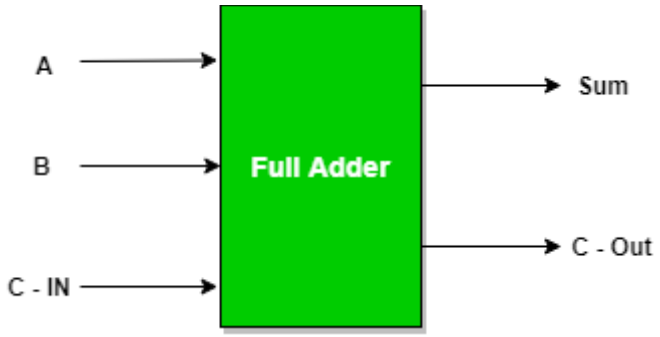
Boolean Expressions

1. **Sum** (S):
   The sum is 1 when an odd number of inputs are 1, which is equivalent to the XOR of the three inputs:
   $$S = A \oplus B \oplus C_{in}$$
2. **Carry** (C_{out}):
   The carry is 1 if at least two of the three inputs are 1. This can be derived using the following expression:
   $$C_{out} = (A \cdot B) + (B \cdot C_{in}) + (A \cdot C_{in})$$

Logic Diagram of a Full Adder

The Full Adder can be constructed using two **Half Adders** and an **OR gate**.

1. **Step 1**: Compute the intermediate sum and carry using the first Half Adder:
   - Inputs: A, B.
   - Outputs: S_1 (Intermediate Sum) and C_1 (Intermediate Carry).
   - $S_1 = A \oplus B, C_1 = A \cdot B$.
2. **Step 2**: Add S_1 and C_{in} using a second Half Adder:
   - Inputs: S_1, C_{in}.
   - Outputs: S (Final Sum) and C_2 (Carry from S_1 and C_{in}).
   - $S = S_1 \oplus C_{in}, C_2 = S_1 \cdot C_{in}$.
3. **Step 3**: Combine the two carries (C_1, C_2) using an OR gate:
   - $C_{out} = C_1 + C_2$.



Inputs			Outputs	
A	B	C – IN	Sum	C - Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Logical Expression for SUM:

= A’ B’ C-IN + A’ B C-IN’ + A B’ C-IN’ + A B C-IN

= C-IN (A’ B’ + A B) + C-IN’ (A’ B + A B’)

= C-IN XOR (A XOR B) = (1,2,4,7)

Logical Expression for C-OUT:

= A’ B C-IN + A B’ C-IN + A B C-IN’ + A B C-IN

= A B + B C-IN + A C-IN = (3,5,6,7)

Another form in which C-OUT can be implemented:

= A B + A C-IN + B C-IN (A + A’)

= A B C-IN + A B + A C-IN + A’ B C-IN

= A B (1 +C-IN) + A C-IN + A’ B C-IN

= A B + A C-IN + A’ B C-IN

= A B + A C-IN (B + B’) + A’ B C-IN

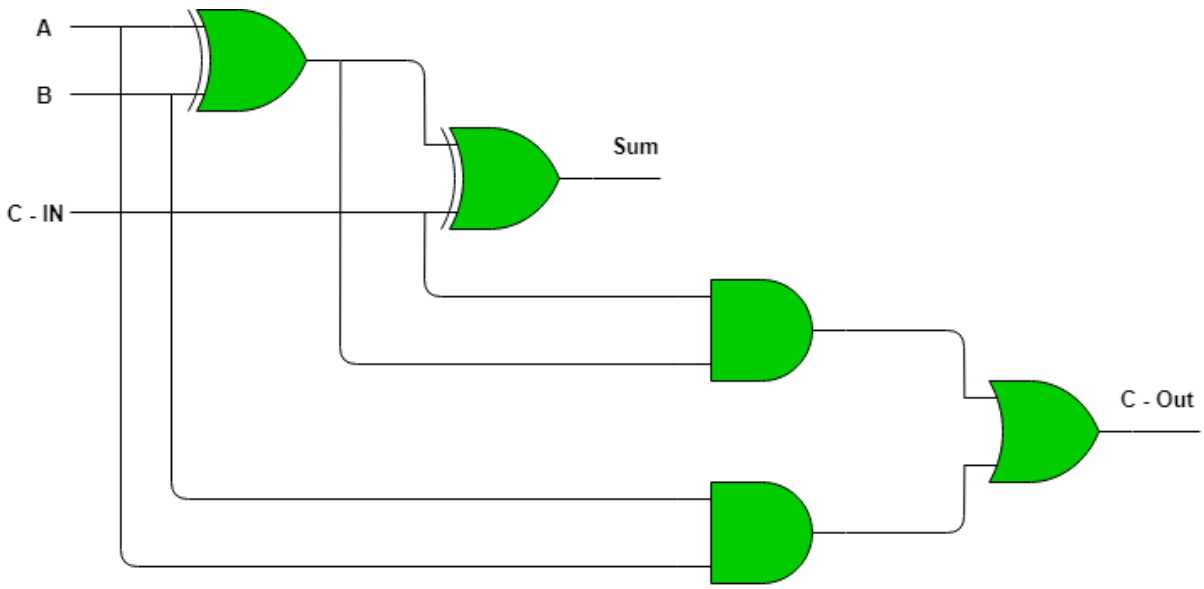
= A B C-IN + A B + A B’ C-IN + A’ B C-IN

= A B (C-IN + 1) + A B’ C-IN + A’ B C-IN

= A B + A B’ C-IN + A’ B C-IN

= AB + C-IN (A’ B + A B’)

Therefore COUT = AB + C-IN (A EX – OR B)



Applications of Full Adder

1. **Multi-bit Adders**:
    - Full Adders are cascaded to form **Ripple Carry Adders** for multi-bit binary addition.
2. **Arithmetic Circuits**:
    - Used in arithmetic logic units (ALUs) for addition and subtraction operations.
3. **Digital Signal Processing**:
    - Core component in digital systems that perform arithmetic computations.

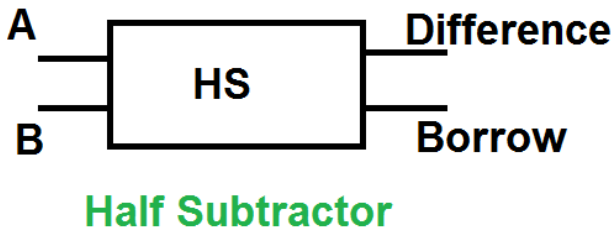
HALF SUBTRACTOR

A half subtractor is a digital logic circuit that performs binary subtraction of two single-bit binary numbers. It has two inputs, A and B, and two outputs, DIFFERENCE and BORROW. The DIFFERENCE output is the difference between the two input bits, while the BORROW output indicates whether borrowing was necessary during the subtraction.

The half subtractor can be implemented using basic gates such as XOR and NOT gates. The DIFFERENCE output is the XOR of the two inputs A and B, while the BORROW output is the NOT of input A and the AND of inputs A and B.

Half Subtractor

Half subtractor is a combination circuit with two inputs and two outputs that are **different** and **borrow**. It produces the difference between the two binary bits at the input and also produces an output (Borrow) to indicate if a 1 has been borrowed. In the subtraction (A-B), A is called a **Minuend bit** and B is called a **Subtrahend bit.**



Half Subtractor

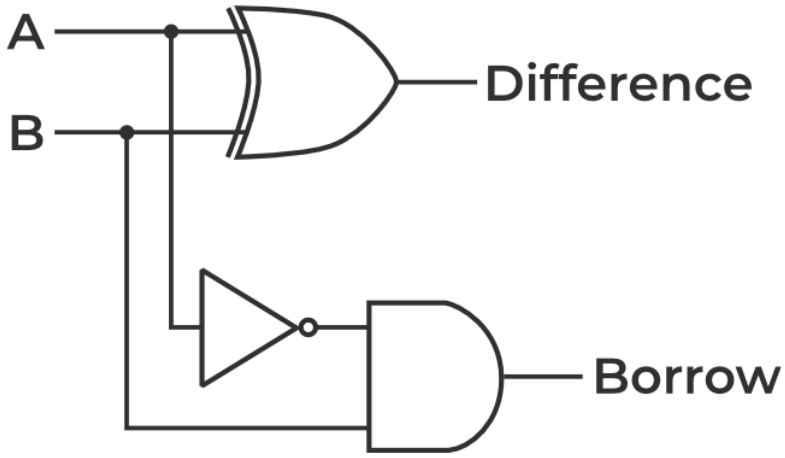
Truth Table

A	B	Diff	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

The SOP form of the Diff and Borrow is as follows:

Diff= A'B+AB'
Borrow = A'B

Implementation



Logical Expression

Difference = A XOR B
Borrow = \overline{A}B

Advantages of Half Adder and Half Subtractor

1. **Simplicity:** The half adder and half subtractor circuits are simple and easy to design, implement, and debug compared to other binary arithmetic circuits.
2. **Building blocks:** The half adder and half subtractor are basic building blocks that can be used to construct more complex arithmetic circuits, such as full adders and subtractors, multiple-bit adders and subtractors, and carry look-ahead adders.
3. **Low cost:** The half adder and half subtractor circuits use only a few gates, which reduces the cost and power consumption compared to more complex circuits.
4. **Easy integration:** The half adder and half subtractor can be easily integrated with other digital circuits and systems.

Disadvantages of Half Adder and Half Subtractor

1. **Limited functionality:** The half adder and half subtractor can only perform binary addition and subtraction of two single-bit numbers, respectively, and are not suitable for more complex arithmetic operations.
2. **Inefficient for multi-bit numbers:** For multi-bit numbers, multiple half adders or half subtractors need to be cascaded, which increases the complexity and decreases the efficiency of the circuit.
3. **High propagation delay:** The propagation delay of the half adder and half subtractor is higher compared to other arithmetic circuits, which can affect the overall performance of the system.

Application of Half Subtractor in Digital Logic:

1.Calculators: Most mini-computers utilize advanced rationale circuits to perform numerical tasks. A Half Subtractor can be utilized in a number cruncher to deduct two parallel digits from one another.

2.Alarm Frameworks: Many caution frameworks utilize computerized rationale circuits to identify and answer interlopers. A Half Subtractor can be utilized in these frameworks to look at the upsides of two parallel pieces and trigger a caution in the event that they are unique.

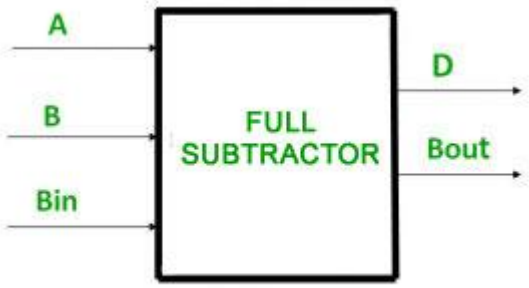
3.Automotive Frameworks: Numerous advanced vehicles utilize computerized rationale circuits to control different capabilities, like the motor administration framework, stopping mechanism, and theater setup. A Half Subtractor can be utilized in these frameworks to perform computations and examinations.

4.Security Frameworks: Advanced rationale circuits are usually utilized in security frameworks to identify and answer dangers. A Half Subtractor can be utilized in these frameworks to look at two double qualities and trigger a caution in the event that they are unique.

5.Computer Frameworks: Advanced rationale circuits are utilized broadly in PC frameworks to perform estimations and examinations. A Half Subtractor can be utilized in a PC framework to deduct two paired values from one another.

FULL SUBTRACTOR

A full subtractor is a **combinational circuit** that performs subtraction of two bits, one is minuend and other is subtrahend, taking into account borrow of the previous adjacent lower minuend bit. This circuit **has three inputs and two outputs**. The three inputs A, B and Bin, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D and Bout represent the difference and output borrow, respectively. Although subtraction is usually achieved by adding the complement of subtrahend to the minuend, it is of academic interest to work out the Truth Table and logic realisation of a full subtractor; x is the minuend; y is the subtrahend; z is the input borrow; D is the difference; and B denotes the output borrow. The corresponding maps for logic functions for outputs of the full subtractor namely difference and borrow.



Here's how a full subtractor works:

1. First, we need to convert the binary numbers to their two's complement form if we are subtracting a negative number.
2. Next, we compare the bits in the minuend and subtrahend at the corresponding positions. If the

subtrahend bit is greater than or equal to the minuend bit, we need to borrow from the previous stage (if there is one) to subtract the subtrahend bit from th**e** minuend bit.
3. We subtract the two bits along with the borrow-in to get the difference bit. If the minuend bit is greater than or equal to the subtrahend bit along with the borrow-in, then the difference bit is 1, otherwise it is 0.
4. We then calculate the borrow-out bit by comparing the minuend and subtrahend bits. If the minuend bit is less than the subtrahend bit along with the borrow-in, then we need to borrow for the next stage, so the borrow-out bit is 1, otherwise it is 0.

The circuit diagram for a full subtractor usually consists of two half-subtractors and an additional OR gate to calculate the borrow-out bit. The inputs and outputs of the full subtractor are as follows:

Inputs:

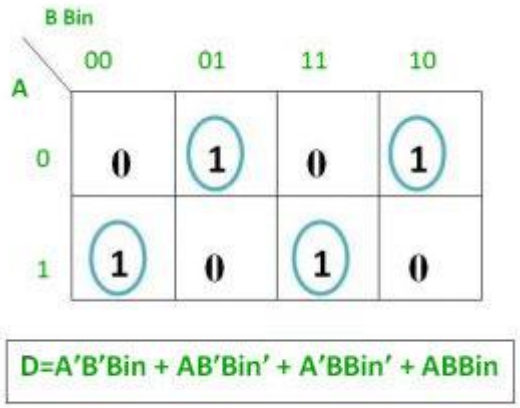
A: minuend bit
B: subtrahend bit
Bin: borrow-in bit from the previous stage
Outputs:

Diff: difference bit
Bout: borrow-out bit for the next stage

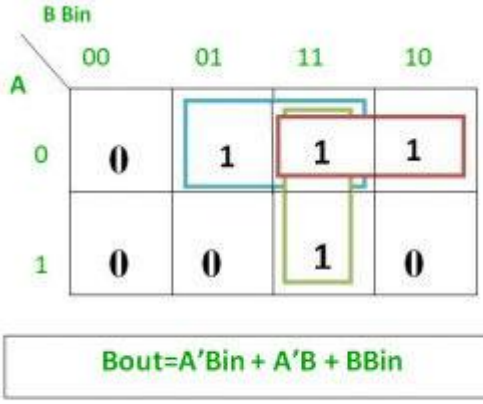
Truth Table –

INPUT			OUTPUT	
A	B	Bin	D	Bout
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

From above table we can draw the K-Map as shown for “difference” and “borrow”.



D=A’B’Bin + AB’Bin’ + A’BBin’ + ABBin



Bout=A’Bin + A’B + BBin

Logical expression for difference –

```
D   = A'B'Bin + A'BBin' + AB'Bin' + ABBin
    = Bin(A'B' + AB)  + Bin'(AB' + A'B)
    = Bin( A XNOR B) + Bin'(A XOR B)
    = Bin (A XOR B)'  +  Bin'(A XOR B)
    = Bin XOR (A XOR B)
    = (A XOR B) XOR Bin
```

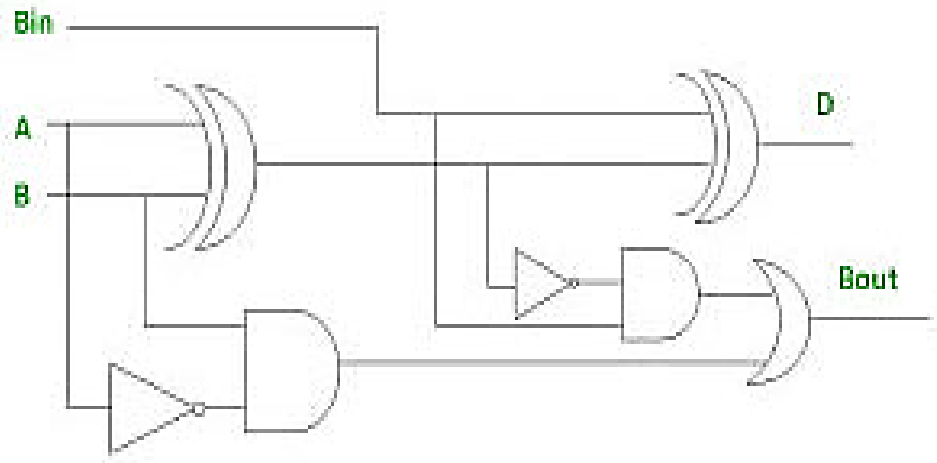
Logical expression for borrow –

```
Bout = A'B'Bin + A'BBin' + A'BBin + ABBin
     = A'B'Bin +A'BBin' + A'BBin + A'BBin + A'BBin + ABBin
     = A'Bin(B + B') + A'B(Bin + Bin') + BBin(A + A')
     = A'Bin + A'B + BBin

OR

Bout = A'B'Bin + A'BBin' + A'BBin + ABBin
     = Bin(AB + A'B') + A'B(Bin + Bin')
     = Bin( A XNOR B) + A'B
     = Bin (A XOR B)' + A'B
```

Logic Circuit for Full Subtractor –



RIPPLE ADDERS AND SUBTRACTORS

Ripple Carry Adder

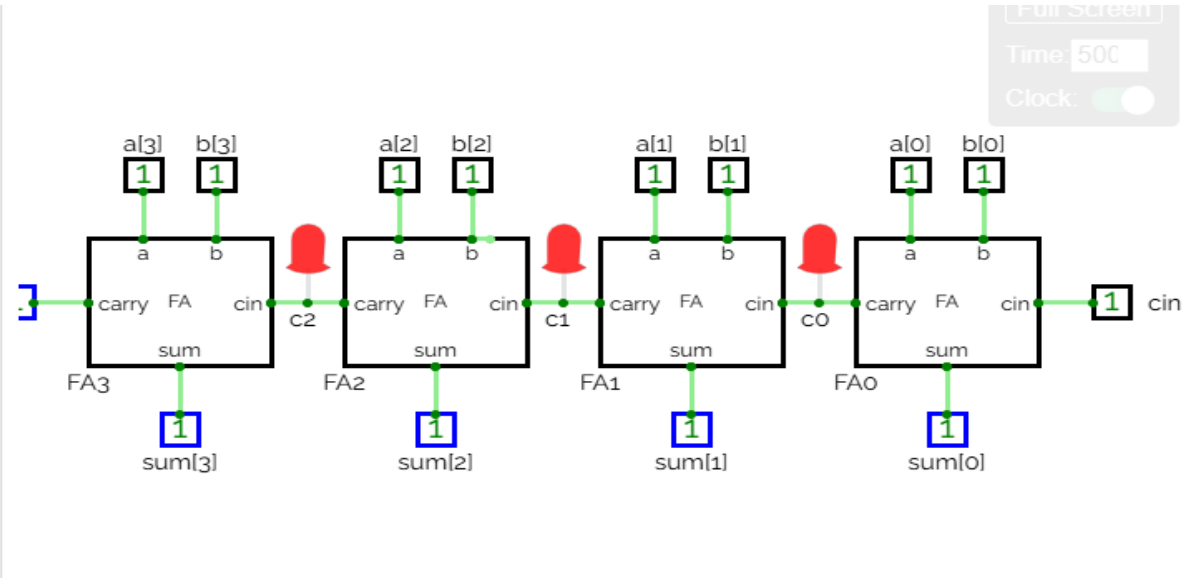
A **Ripple Carry Adder (RCA)** is a type of adder used for performing multi-bit binary addition. It is constructed by chaining together multiple **Full Adders** where the **Carry-out** of one full adder is connected to the **Carry-in** of the next.

Key Features:

- **Simple Design**: It uses Full Adders in series to add multiple bits.
- **Carry Propagation**: The carry propagates through each stage, hence the name "Ripple Carry."
- **Slower Performance**: The carry signal must ripple through all stages, making the RCA slower for large bit-widths, as each stage depends on the carry from the previous stage.

Ripple Carry Adder Design

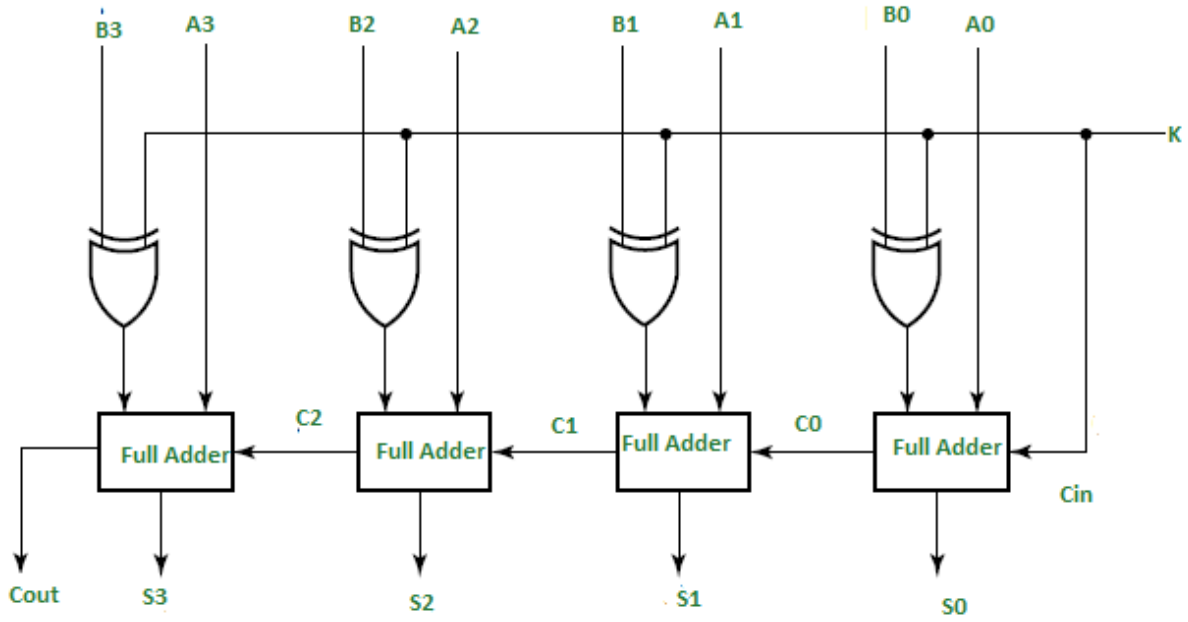
1. Inputs:
   - Two binary numbers $A = A_n A_{n-1} \ldots A_0$ and $B = B_n B_{n-1} \ldots B_0$, where n is the number of bits.
   - Carry-in C_{in} (initially 0 for the first adder).
2. Outputs:
   - Sum: $S = S_n S_{n-1} \ldots S_0$.
   - Carry-out: C_{out} (the final carry bit).
3. Circuit:
   - **Full Adders** are connected in series.
   - Each Full Adder computes the sum and the carry-out. The carry-out from one adder is the carry-in for the next one.



Ripple Carry Subtractor

A **Ripple Carry Subtractor** is designed similarly to an adder but for subtraction. It uses **Full Subtractors** (or a combination of Full Adders and Inverters) to perform the subtraction of two binary numbers.

1. **Inputs**:
   - Two binary numbers $A = A_n A_{n-1} \ldots A_0$ and $B = B_n B_{n-1} \ldots B_0$.
   - Borrow-in B_{in} (initially 0 for the first subtractor).
2. **Outputs**:
   - Difference: $D = D_n D_{n-1} \ldots D_0$.
   - Borrow-out: B_{out} (the final borrow bit).
3. **Circuit**:
   - **Full Subtractors** are connected in series.
   - Each Full Subtractor computes the difference and the borrow-out. The borrow-out from one subtractor is the borrow-in for the next one.



Ripple Carry Adder vs. Ripple Carry Subtractor

- **Ripple Carry Adder**:
  - Designed for addition of binary numbers.
  - Uses Full Adders.
  - Carry propagates through each bit.
  - Slower for larger bit widths due to the carry propagation delay.
- **Ripple Carry Subtractor**:
  - Designed for subtraction of binary numbers.

- Uses Full Subtractors (or Full Adders with inverted inputs).
- Borrow propagates through each bit.
- Similar to the RCA but with borrow rather than carry.

Advantages and Disadvantages

Advantages:

1. **Simple Design**: Both Ripple Carry Adders and Subtractors are simple to design and easy to implement.
2. **Modularity**: Can easily scale to add or subtract more bits by chaining additional Full Adders or Full Subtractors.

Disadvantages:

1. **Slow for Large Bit Widths**: The propagation delay of carry or borrow signals can cause significant delay as the number of bits increases, which makes these adders and subtractors slower compared to more advanced designs like **Carry Look-Ahead Adders** or **Borrow Look-Ahead Subtractors**.
2. **Efficiency**: The ripple effect causes inefficiencies in high-speed applications due to the time it takes for the carry or borrow to ripple through all bits.

Applications

1. **Arithmetic Units**: Used in simple arithmetic operations in ALUs.
2. **Digital Processors**: Basic components for adding or subtracting multi-bit numbers in processors.
3. **Digital Signal Processing**: Useful in circuits requiring basic binary arithmetic.

UNIT – IV

Combinational Logic Circuits – 2

Design of decoders

A binary decoder is a digital circuit that converts a binary code into a set of outputs. The binary code represents the position of the desired output and is used to select the specific output that is active. Binary decoders are the inverse of encoders and are commonly used in digital systems to convert a serial code into a parallel set of outputs.

1. The basic principle of a binary decoder is to assign a unique output to each possible binary code. For example, a binary decoder with 4 inputs and 2^4 = 16 outputs can assign a unique output to each of the 16 possible 4-bit binary codes.
2. The inputs of a binary decoder are usually active low, meaning that only one input is active (low) at any given time, and the remaining inputs are inactive (high). The active low input is used to select the specific output that is active.
3. There are different types of binary decoders, including priority decoders, which assign a priority to each output, and error-detecting decoders, which can detect errors in the binary code and generate an error signal.

In summary, a binary decoder is a digital circuit that converts a binary code into a set of outputs. Binary decoders are the inverse of encoders and are widely used in digital systems to convert serial codes into parallel outputs.

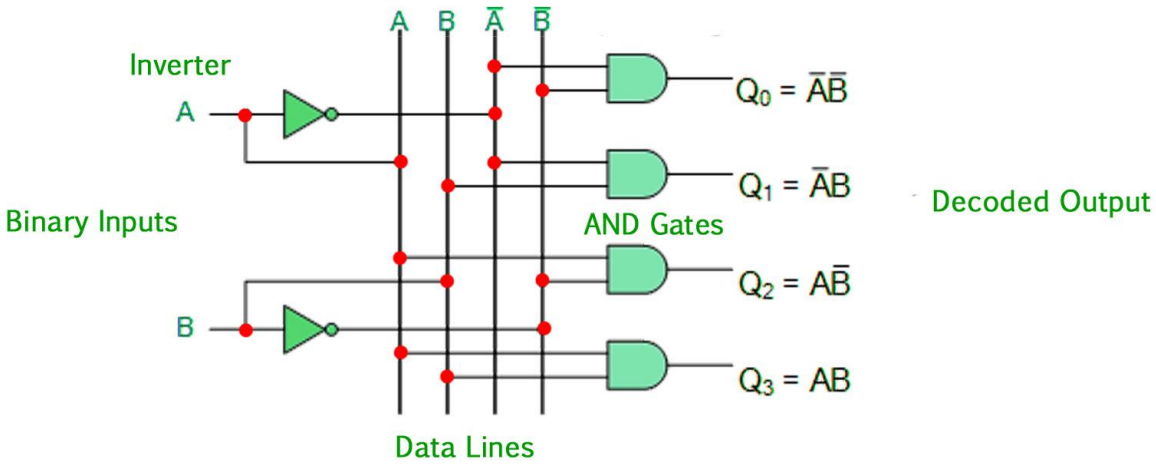
In Digital Electronics, discrete quantities of information are represented by binary codes. A binary code of **n bits** is capable of representing up to **2^n distinct elements** of coded information. The name **"Decoder"** means to translate or decode coded information from one format into another, so a digital decoder transforms a set of digital input signals into an equivalent decimal code at its output. A **decoder** is a **combinational circuit** that converts binary information from **n input lines** to a maximum of **2^n unique output lines**.

Binary Decoder –

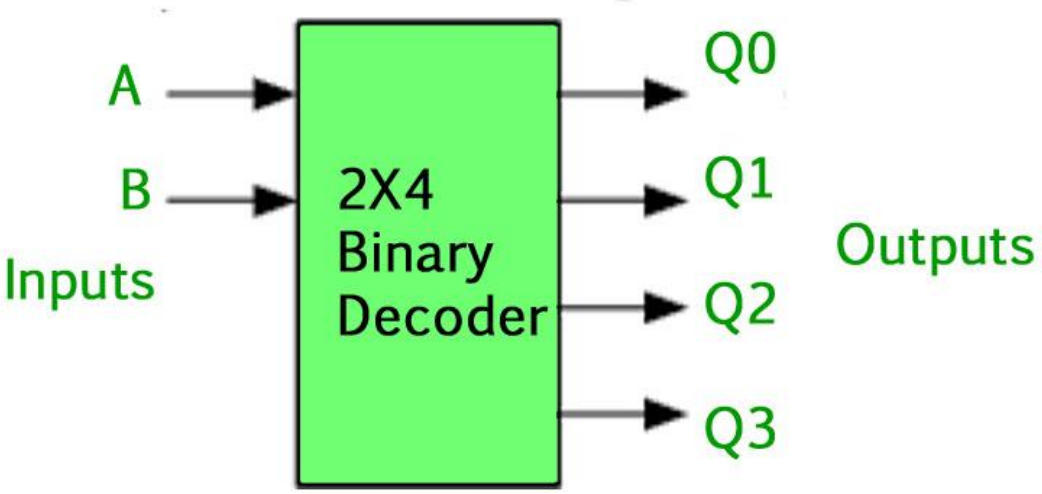
- Binary Decoders are another type of digital logic device that has inputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines, so a decoder that has a set of two or more bits will be defined as having an n-bit code, and therefore it will be possible to represent 2^n possible values.
- If a binary decoder receives n inputs it activates one and only one of its 2^n outputs based on that input with all other outputs deactivated. If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.
- Example, an inverter (NOT-gate) can be classified as a 1-to-2 binary decoder as 1-input and 2-outputs is possible. i.e an input A can give either A or A complement as the output.
- Then we can say that a standard combinational logic decoder is an n-to-m decoder, where m <= 2^n, and whose output, Q is dependent only on its present input states.
- Their purpose is to generate the 2^n (or fewer) minterms of n input variables. Each combination of inputs will assert a unique output.

A Binary Decoder converts coded inputs into coded outputs, where the input and output codes are different and decoders are available to “decode” either a Binary or BCD (8421 code) input pattern to typically a Decimal output code. Practical “binary decoder” circuits include 2-to-4, 3-to-8 and 4-to-16 line configurations.

2-to-4 Binary Decoder –



The 2-to-4 line binary decoder depicted above consists of an array of four AND gates. The 2 binary inputs labeled A and B are decoded into one of 4 outputs, hence the description of a 2-to-4 binary decoder. Each output represents one of the minterms of the 2 input variables, (each output = a minterm).



A	B	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

The output values will be: Qo=A'B' Q1=A'B Q2=AB' Q3=AB The binary inputs A and B determine which output line from Q0 to Q3 is "HIGH" at logic level "1" while the remaining outputs are held "LOW" at logic "0" so only one output can be active (HIGH) at any one time. Therefore, whichever output line is "HIGH" identifies the binary code present at the input, in other words, it "decodes" the binary input. Some binary decoders have an additional input pin labeled "Enable" that controls the outputs from the device. This extra input allows the outputs of the decoder to be turned "ON" or "OFF" as required. The output is only generated when the Enable input has value 1; otherwise, all

outputs are 0. Only a small change in the implementation is required: the Enable input is fed into the AND gates which produce the outputs. If Enable is 0, all AND gates are supplied with one of the inputs as 0 and hence no output is produced. When Enable is 1, the AND gates get one of the inputs as 1, and now the output depends upon the remaining inputs. Hence the output of the decoder is dependent on whether the Enable is high or low.

An encoder is a digital circuit that converts a set of binary inputs into a unique binary code. The binary code represents the position of the input and is used to identify the specific input that is active. Encoders are commonly used in digital systems to convert a parallel set of inputs into a serial code.

The basic principle of an encoder is to assign a unique binary code to each possible input. For example, a 2-to-4 line encoder has 2 input lines and 4 output lines and assigns a unique 4-bit binary code to each of the 2^2 = 4 possible input combinations. The output of an encoder is usually active low, meaning that only one output is active (low) at any given time, and the remaining outputs are inactive (high). The active low output is selected based on the binary code assigned to the active input.

There are different types of encoders, including priority encoders, which assign a priority to each input, and binary-weighted encoders, which use a binary weighting system to assign binary codes to inputs. In summary, an encoder is a digital circuit that converts a set of binary inputs into a unique binary code that represents the position of the input. Encoders are widely used in digital systems to convert parallel inputs into serial codes.

An Encoder is a **combinational circuit** that performs the reverse operation of a Decoder. It has a maximum of **2^n input lines** and **'n' output lines**, hence it encodes the information from 2^n inputs into an n-bit code. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2^n input lines with 'n' bits.

Encoder

Types of Encoders

There are different types of Encoders which are mentioned below.

- 4 to 2 Encoder
- Octal to Binary Encoder (8 to 3 Encoder)
- Decimal to BCD Encoder
- Priority Encoder

4 to 2 Encoder

The 4 to 2 Encoder consists of **four inputs Y3, Y2, Y1 & Y0, and two outputs A1 & A0**. At any time, only one of these 4 inputs can be '1' in order to get the respective binary code at the output. The figure below shows the logic symbol of the 4 to 2 encoder.

4 to 2 Encoder

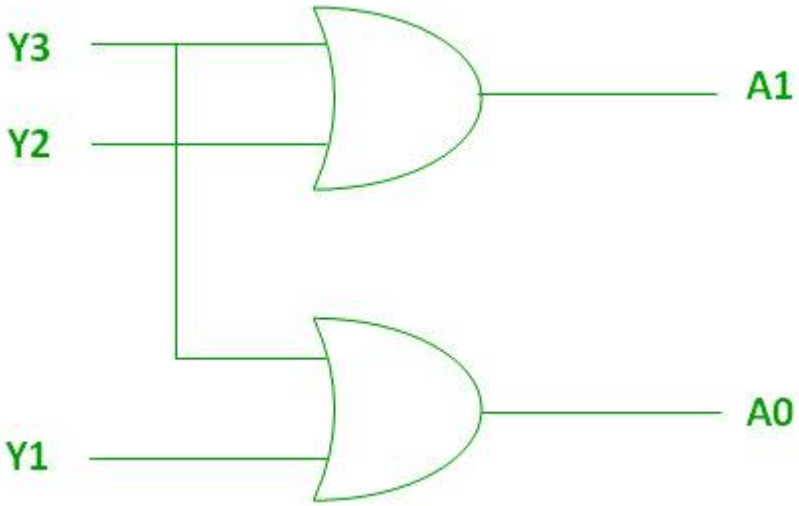
The Truth table of 4 to 2 encoders is as follows.

INPUTS				**OUTPUTS**	
Y3	**Y2**	**Y1**	**Y0**	**A1**	**A0**
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

Logical expression for A1 and A0:

A1 = Y3 + Y2
A0 = Y3 + Y1

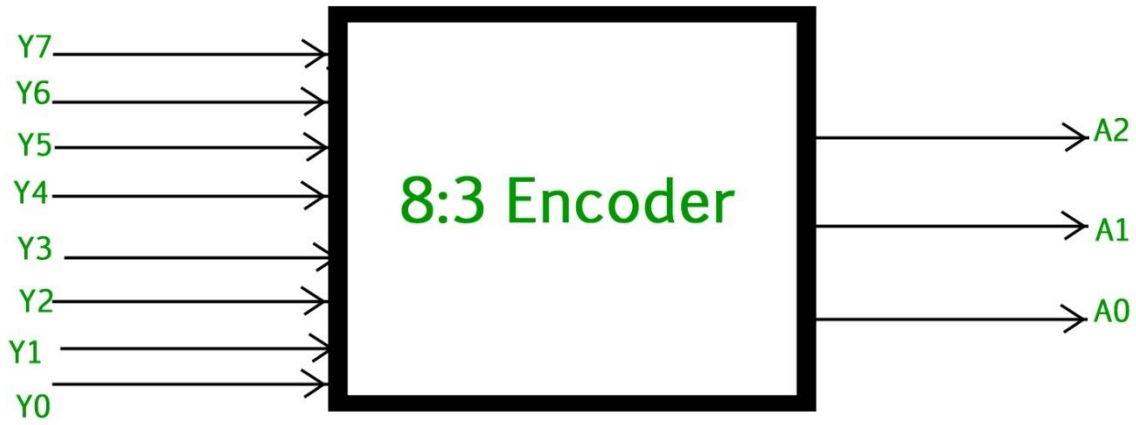
The above two Boolean functions A1 and A0 can be implemented using two input OR gates :



Implementation using OR Gate

Octal to Binary Encoder (8 to 3 Encoder)

The 8 to 3 Encoder or octal to Binary encoder consists of **8 inputs**: Y7 to Y0 and **3 outputs**: A2, A1 & A0. Each input line corresponds to each octal digit and three outputs generate corresponding binary code. The figure below shows the logic symbol of octal to the binary encoder.



Octal to Binary Encoder (8 to 3 Encoder)

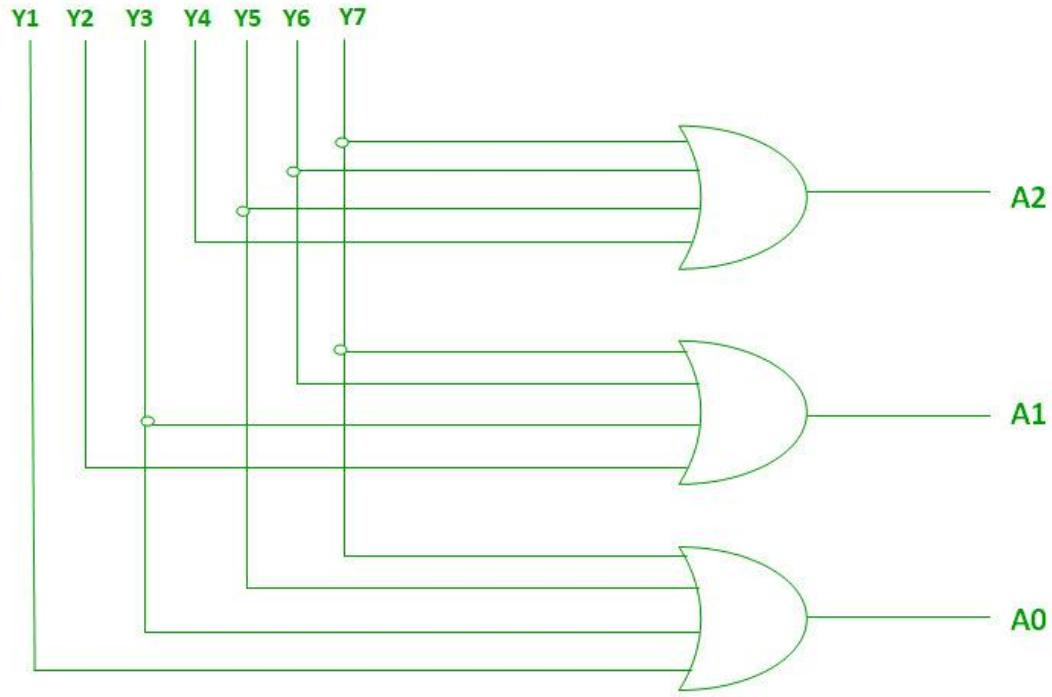
The truth table for the 8 to 3 encoder is as follows.

INPUTS								**OUTPUTS**		
Y7	**Y6**	**Y5**	**Y4**	**Y3**	**Y2**	**Y1**	**Y0**	**A2**	**A1**	**A0**
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

Logical expression for A2, A1, and A0.

A2 = Y7 + Y6 + Y5 + Y4
A1 = Y7 + Y6 + Y3 + Y2
A0 = Y7 + Y5 + Y3 + Y1

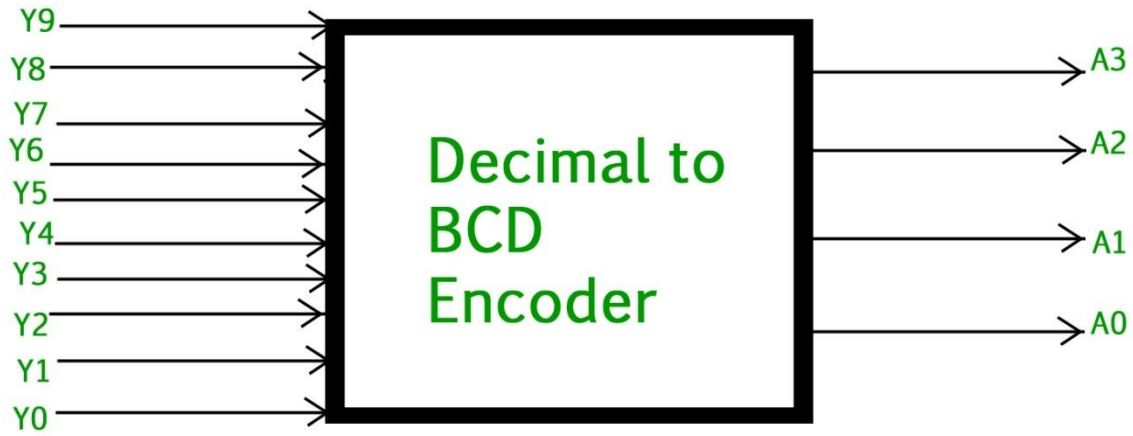
The above two Boolean functions A2, A1, and A0 can be implemented using four input OR gates.



Implementation using OR Gate

Decimal to BCD Encoder

The decimal-to-binary encoder usually consists of **10 input lines** and **4 output lines**. Each input line corresponds to each decimal digit and 4 outputs correspond to the BCD code. This encoder accepts the decoded decimal data as an input and encodes it to the BCD output which is available on the output lines. The figure below shows the logic symbol of the decimal to BCD encoder :



Decimal to BCD Encoder

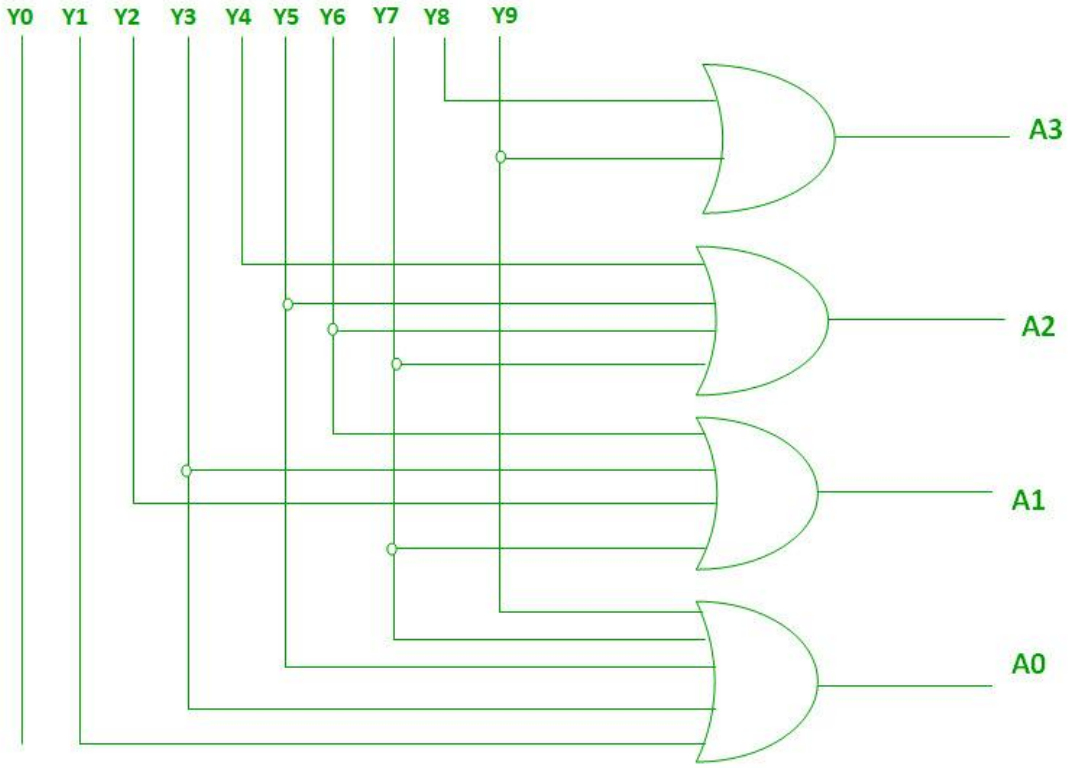
The truth table for decimal to BCD encoder is as follows.

INPUTS										**OUTPUTS**			
Y9	**Y8**	**Y7**	**Y6**	**Y5**	**Y4**	**Y3**	**Y2**	**Y1**	**Y0**	**A3**	**A2**	**A1**	**A0**
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	1	0	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0	1

Logical expression for A3, A2, A1, and A0.

A3 = Y9 + Y8
A2 = Y7 + Y6 + Y5 +Y4
A1 = Y7 + Y6 + Y3 +Y2
A0 = Y9 + Y7 +Y5 +Y3 + Y1

The above two Boolean functions can be implemented using OR gates.



Implementation using OR Gate

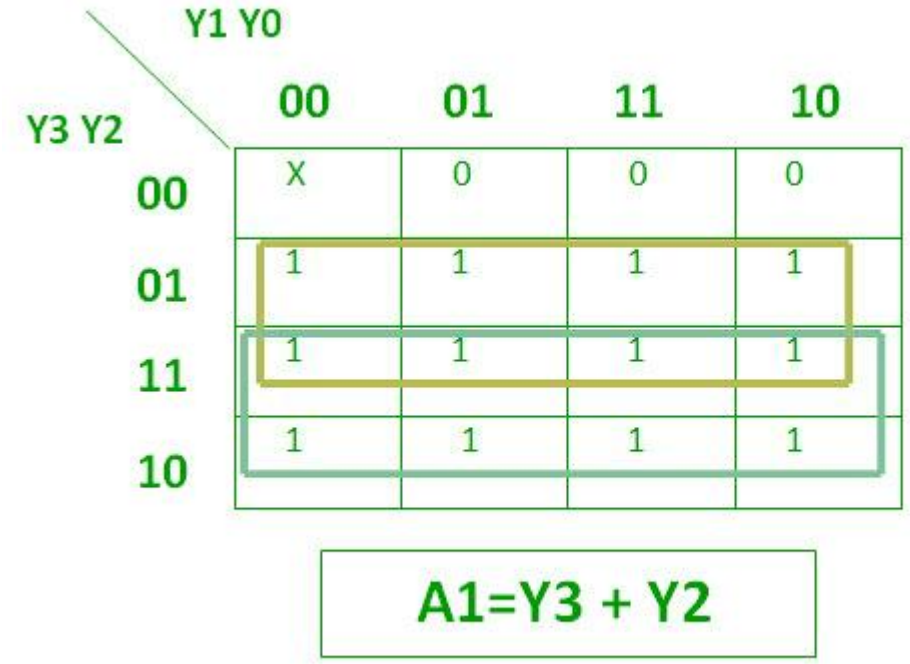
Priority Encoder

A 4 to 2 priority encoder has **4 inputs**: Y3, Y2, Y1 & Y0, and **2 outputs**: A1 & A0. Here, the input, Y3 has the **highest priority**, whereas the input, Y0 has the **lowest priority**. In this case, even if more than one input is '1' at the same time, the output will be the (binary) code corresponding to the input, which is having **higher priority**. The truth table for the priority encoder is as follows.

INPUTS				**OUTPUTS**		
Y3	**Y2**	**Y1**	**Y0**	**A1**	**A0**	**V**
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1

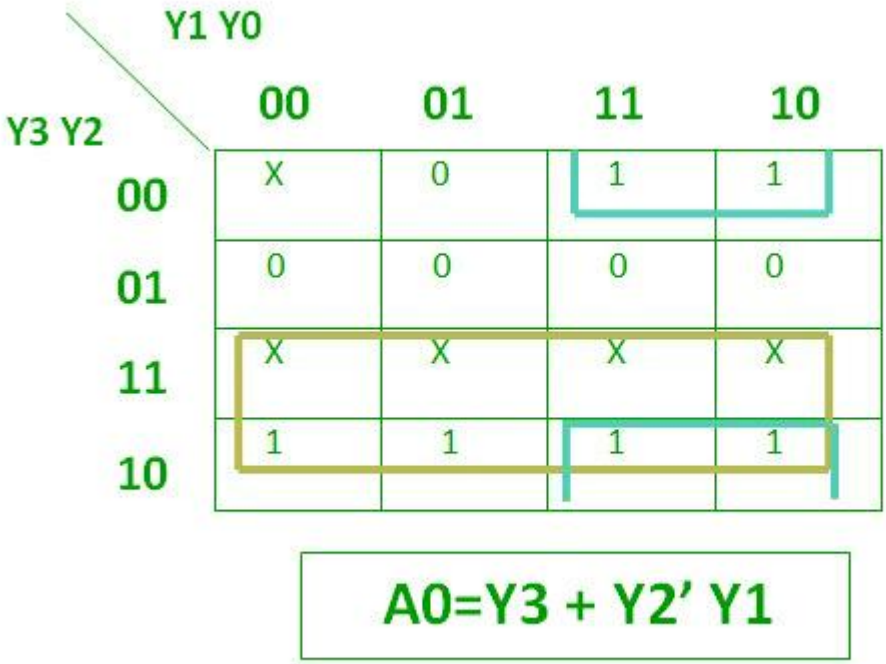
INPUTS				OUTPUTS		
0	1	X	X	1	0	1
1	X	X	X	1	1	1

The logical expression for A1 is shown below.



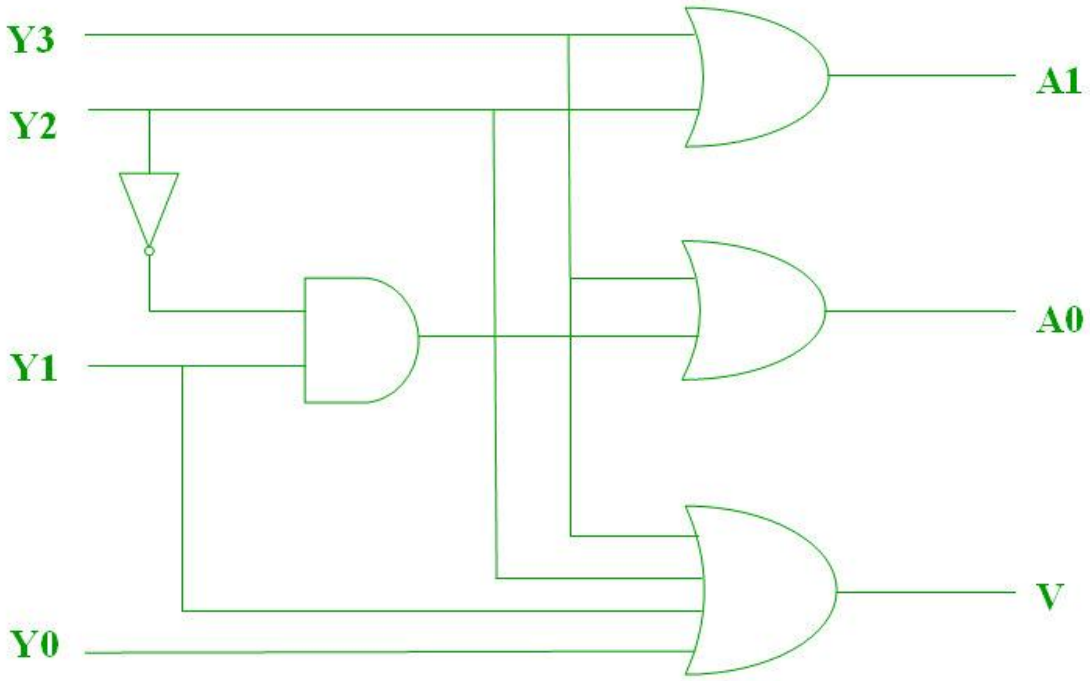
Logical Expression

The Logical Expression for A0 is shown below.



Logical Expression

The above two Boolean functions can be implemented as.



Priority Encoder

There are some errors that usually happen in Encoders are mentioned below.

- There is an ambiguity, when all outputs of the encoder are equal to zero.
- If more than one input is active High, then the encoder produces an output, which may not be the correct code.

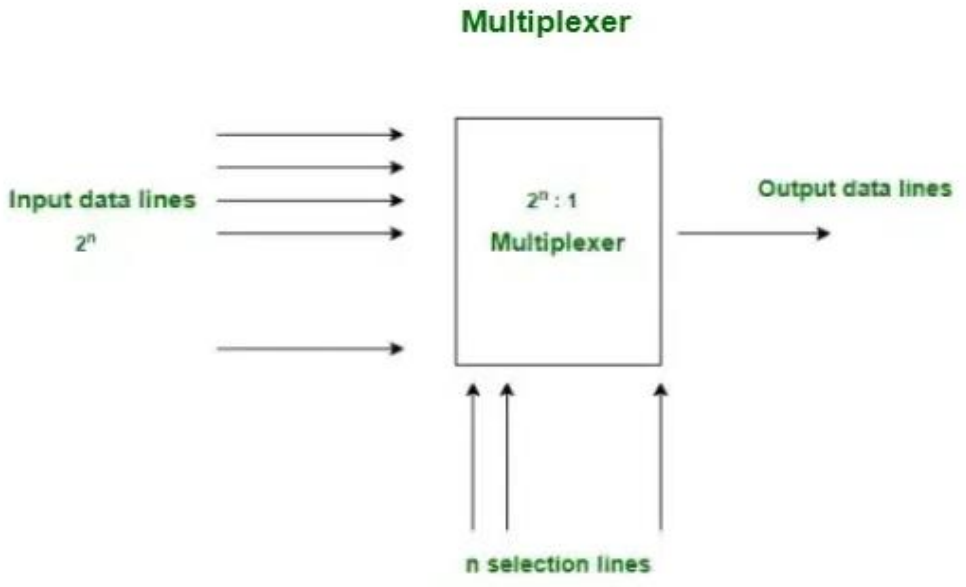
So, to overcome these difficulties, we should assign priorities to each input of the encoder. Then, the output of the encoder will be the code corresponding to the active high inputs, which have higher priority.

Multiplexers

What Are Multiplexers?

A multiplexer is a combinational circuit that has many data inputs and a single output, depending on control or select inputs. For N input lines, log2(N) selection lines are required, or equivalently, for 2n2*n* input lines, n selection lines are needed. Multiplexers are also known as "N-to-1 selectors," parallel-to-serial converters, many-to-one circuits, and universal logic circuits. They are mainly used to increase the amount of data that can be sent over a network within a certain amount of time and bandwidth .

Multiplexer



Types of Mux

The Mux can be of different types based on input but in this article we will go through two major types of mux which are

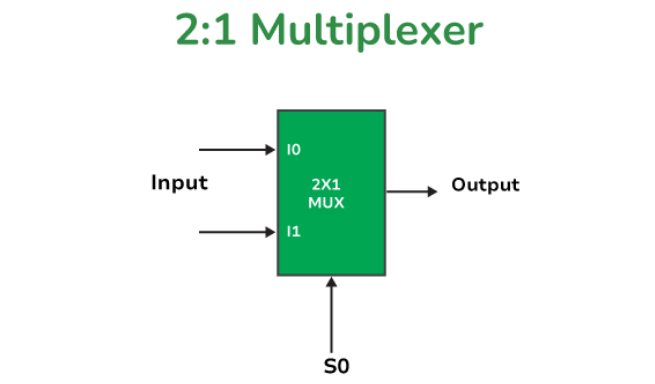
- **2×1 Mux**
- **4×1 Mux**

2×1 Multiplexer

The 2×1 is a fundamental circuit which is also known 2-to-1 multiplexer that are used to choose one signal from two inputs and transmits it to the output. The 2×1 mux has two input lines, one output line, and a single selection line. It has various applications in digital systems such as in microprocessor it is used to select between two different data sources or between two different instructions.

Block Diagram of 2:1 Multiplexer with Truth Table

Given Below is the Block Diagram and Truth Table of 2:1 Mux. In this Block Diagram where I0 and I1 are the input lines ,Y is the output line and S0 is a single select line.



Truth Table

S_0	I_0	I_1	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

Block Diagram of 2:1 Multiplexer with Truth Table

The output of the 2×1 Mux will depend on the selection line S0,

- When S is 0(low), the I0 is selected
- when S0 is 1(High), I1 is selected

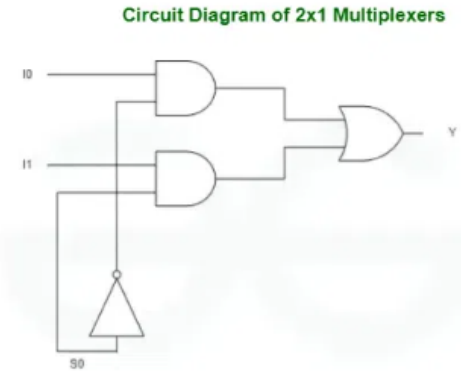
Logical Expression of 2×1 Mux

Using the Truth Table ,the Logical Expression for Mux can be determined as

$$Y = \overline{S_0}.I_0 + S_0.I_1$$

Circuit Diagram of 2×1 Multiplexers

Using truth table the circuit diagram can be given as

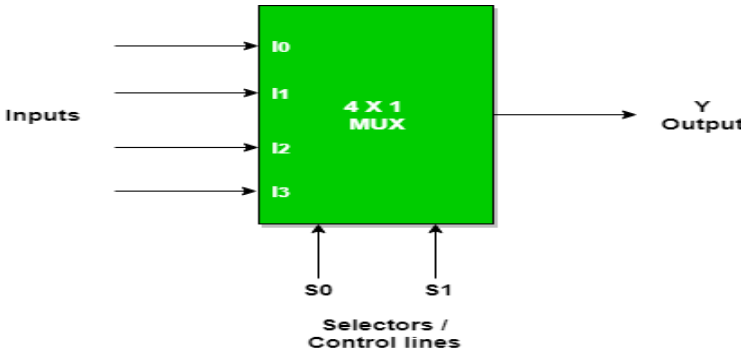


4×1 Multiplexer

The 4×1 Multiplexer which is also known as the 4-to-1 multiplexer. It is a multiplexer that has 4 inputs and a single output. The Output is selected as one of the 4 inputs which is based on the selection inputs. The number of the Selection lines will depend on the number of the input which is determined by the equation log2n*log*2*n* ,In 4×1 Mux the selection lines can be determined as log4=2*log*4=2 ,slo two selections are needed.

Block Diagram of 4×1 Multiplexer

In the Given Block Diagram I0, I1, I2, and I3 are the 4 inputs and Y is the Single output which is based on Select lines S0 and S1.



The output of the multiplexer is determined by the binary value of the selection lines

- When S1S0=00, the input I0 is selected.
- When S1S0=01, the input I1 is selected.
- When S1S0=10, the input I2 is selected.
- When S1S0=11, the input I3 is selected.

Truth Table of 4×1 Multiplexer

Given Below is the Truth Table of 4×1 Multiplexer

Truth Table

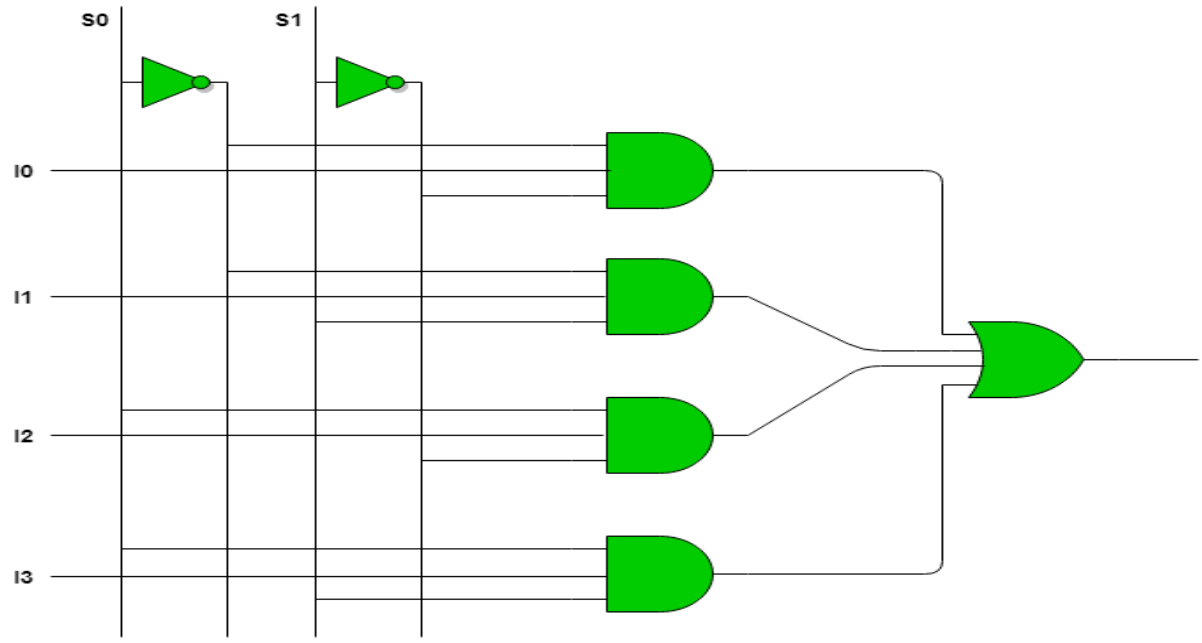
S0	S1	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

So, final equation,

$Y = S0'.S1'.I0 + S0'.S1.I1 + S0.S1'.I2 + S0.S1.I3$

Circuit Diagram of 4×1 Multiplexers

Using truth table the circuit diagram can be given as



Multiplexer can act as universal combinational circuit. All the standard logic gates can be implemented with multiplexers.

Implementation of Different Gates with 2:1 Mux

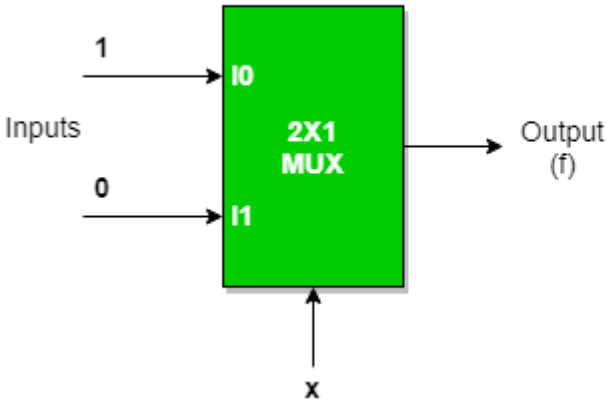
Given below are the Implementation of Different gate using 2:1 Mux

Implementation of NOT gate using 2 : 1 Mux

The Not gate from 2:1 Mux can be obtained by

- Connect the input signal to one of the data input lines(I0).
- Then connect a line (0 or 1) to the other data input line(I1)
- Connect the same input line Select line S0 which is connected to D0.

Given Below is the Diagram for the Logical Representation of **NOT gate using 2 : 1 Mux**



Truth Table

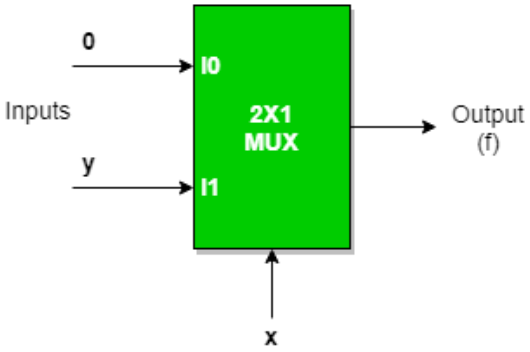
x	f
0	1
1	0

Implementation of AND gate using 2 : 1 Mux

The And gate from 2:1 Mux can be obtained by

- Connect the input Y to I1.
- Connect the input X to the selection line S0.
- Connect a line(0) to I0.

Given Below is the Diagram for the Logical Representation of **AND gate using 2 : 1 Mux**



Truth Table

x	y	f	f→y
0	0	0	f = 0
0	1	0	
1	0	0	f = y
1	1	1	

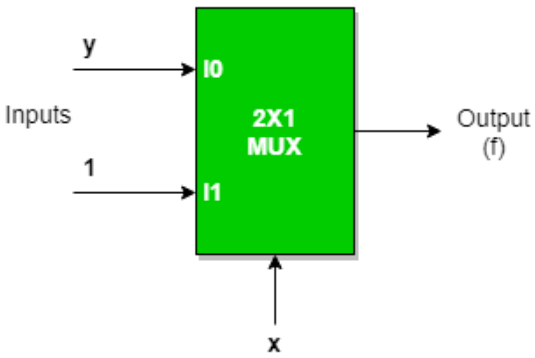
For further more on the **Implementation of AND gate using 2 : 1 Mux**

Implementation of OR gate using 2 : 1 Mux

The OR gate from 2:1 Mux can be obtained by

- Connect input X to the selection line S0.
- Connect input Y to I1.
- Connect Line(1) to I1.

Given Below is the Diagram for the Logical Representation of **OR gate using 2 : 1 Mux**



Truth Table

x	y	f	f→y
0	0	0	f = y
0	1	1	
1	0	1	f = 1
1	1	1	

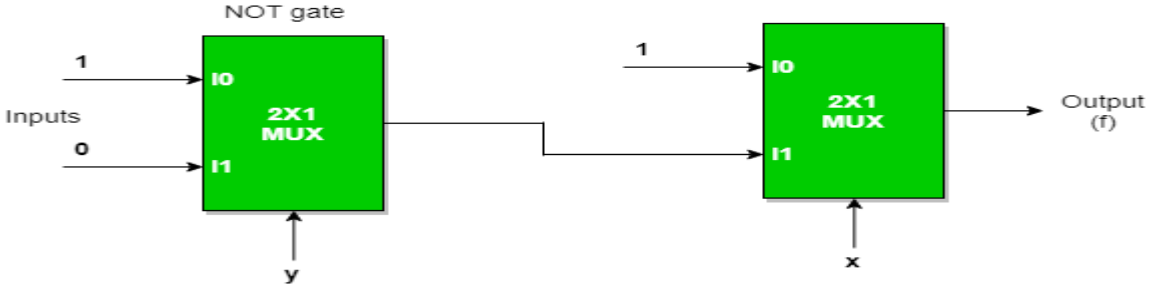
Implementation of NAND, NOR, XOR and XNOR gates requires two 2:1 Mux. First multiplexer will act as NOT gate which will provide complemented input to the second multiplexer.

Implementation of NAND gate using 2 : 1 Mux

The NAND gate from 2:1 Mux can be obtained by

- In first mux take inputs and 1 and 0 and y as selection line.
- In Second MUX the Output from mux is connected to I1.
- line(1) is given to the I0.
- x is given as selection line for the second Mux.

Given Below is the Diagram for the Logical Representation of **NAND gate using 2 : 1 Mux**



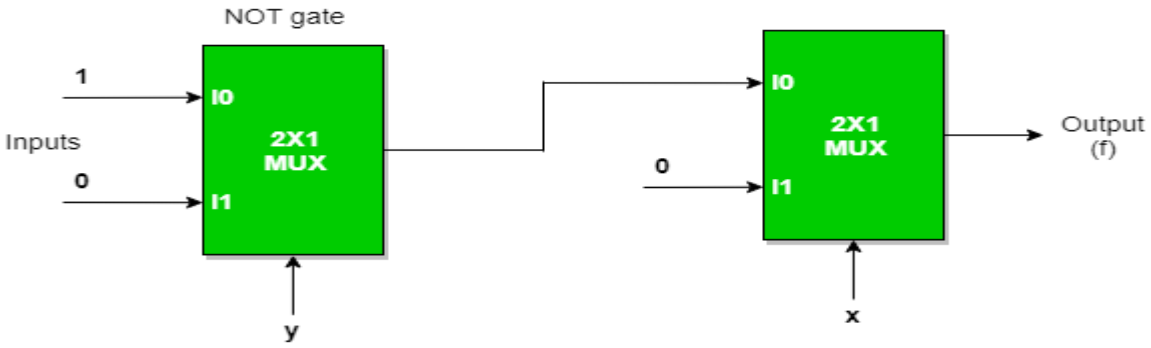
For further more on the **Implementation of NAND gate using 2 : 1 Mux**

Implementation of NOR gate using 2 : 1 Mux

The Nor gate from 2:1 Mux can be obtained by

- In first mux take inputs and 1 and 0 and y as selection line.
- In Second MUX the Output from mux is connected to I0.
- line(0) is given to the I1.
- x is given as selection line for the second Mux.

Given Below is the Diagram for the Logical Representation of **NOR gate using 2 : 1 Mux**



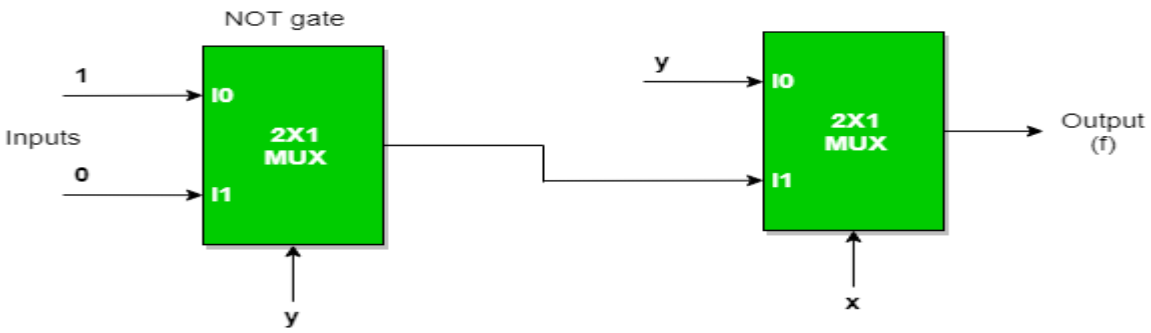
For further more on the **Implementation of NOR gate using 2 : 1 Mux**

Implementation of EX-OR gate using 2 : 1 Mux

The Nor gate from 2:1 Mux can be obtained by

- In first mux take inputs and 1 and 0 and y as selection line.
- In Second MUX the Output from mux is connected to I1.
- y is given to the I0.
- x is given as selection line for the second Mux.

Given Below is the Diagram for the Logical Representation of **EX-OR gate using 2 : 1 Mux**

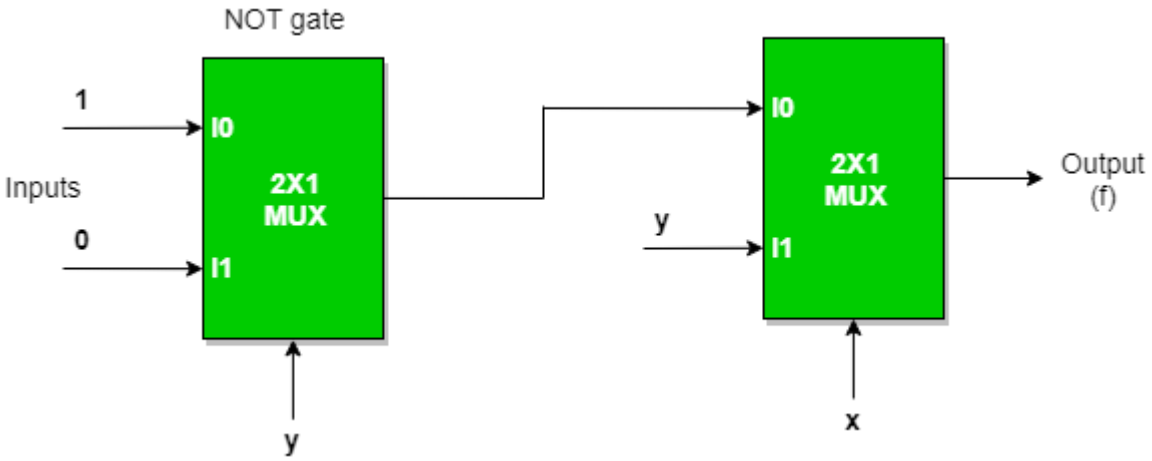


Implementation of EX-NOR gate using 2 : 1 Mux

Given Below is the Diagram for the Logical Representation of **EX-OR gate using 2 : 1 Mux**

The Nor gate from 2:1 Mux can be obtained by

- In first mux take inputs and 1 and 0 and y as selection line.
- In Second MUX the Output from mux is connected to I0.
- y is given to the I1.
- x is given as selection line for the second Mux.

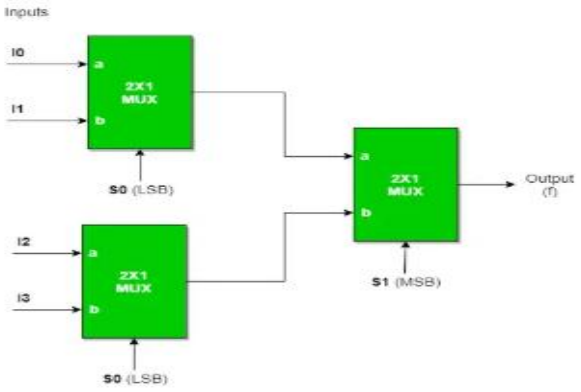


Implementation of Higher Order MUX using Lower Order MUX

Given Below are the Implementation of Higher Order MUX Using Lower Order MUX

4 : 1 MUX using 2 : 1 MUX

Three 2: 1 MUX are required to implement 4 : 1 MUX.



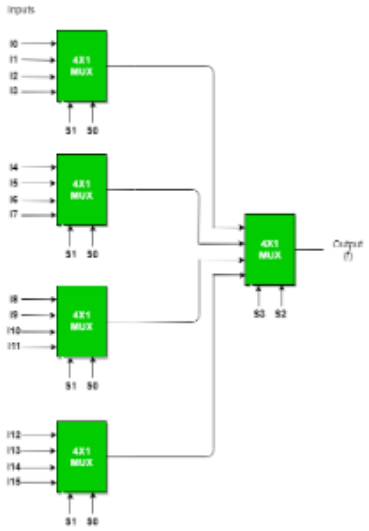
Truth Table

S1	S0	f
0	0	I0
0	1	I1
1	0	I2
1	1	I3

4 : 1 MUX using 2 : 1 MUX

16 : 1 MUX using 4 : 1 MUX

Given Below is the logical Diagram of 16:1 Mux Using 4:1 Mux



DE-Multiplexer

DEMUX or De-Multiplexer is a data distributor combinational circuit. It works in a reverse way of the Multiplexer. The DEMUX has 1 input port and 2^n output lines. Here n signifies the selection line for a DEMUX. As per the selection line value, the DEMUX input lines will be connected to receive the output. Demultiplexer receives digital information from a single source and converts it into several sources.

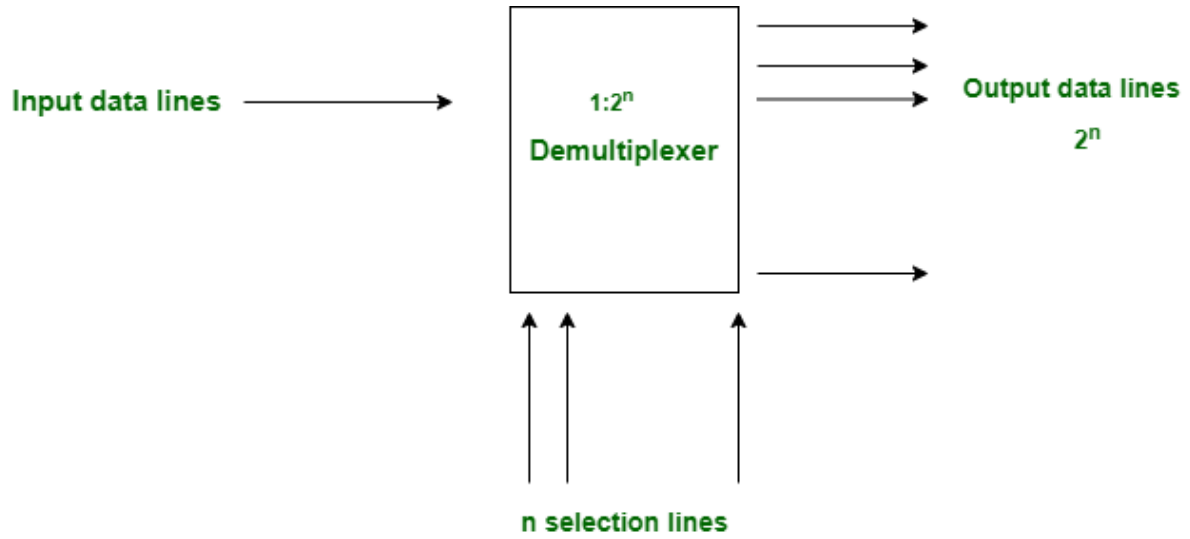
What is a DEMUX?

The DEMUX is a digital information processor. It takes input from one source and also converts the data to transmit towards various sources. The demultiplexer has one data input line. The demultiplexer has several control lines (also known as select lines). These lines determine to which output the input data should be sent. The number of control lines determines the number of output lines.

Let us discuss the DEMUX with the attached information.

General Block Diagram Of A DEMUX

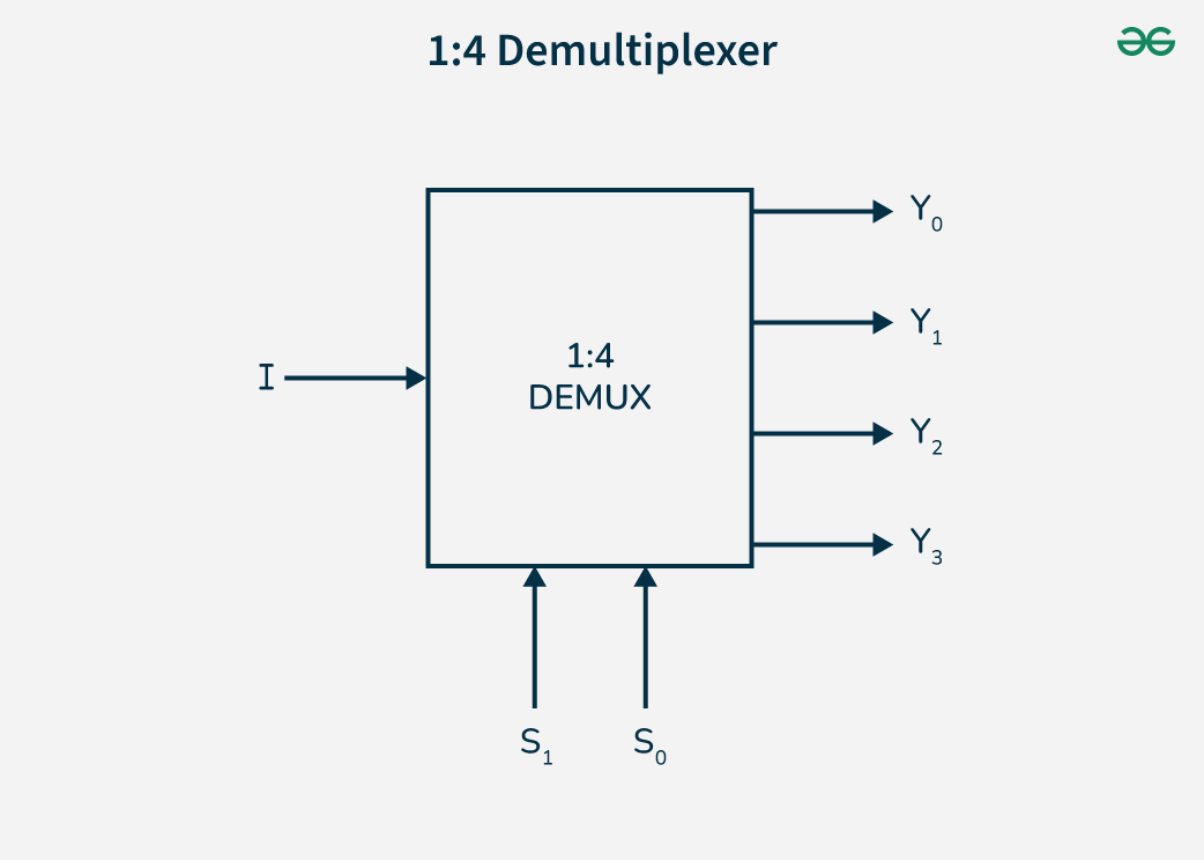
Here is the basic block diagram of a DEMUX as mentioned below.



DEMUX

Truth Table Of A 1X4 DEMUX

A 1x4 DEMUX has only one input which is denoted as I. There are two selection lines i.e. S1 and S0. At last, the DEMUX has output lines including Y3, Y2, Y1 &Y0. Here is the 1x4 DEMUX with diagram as mentioned below.

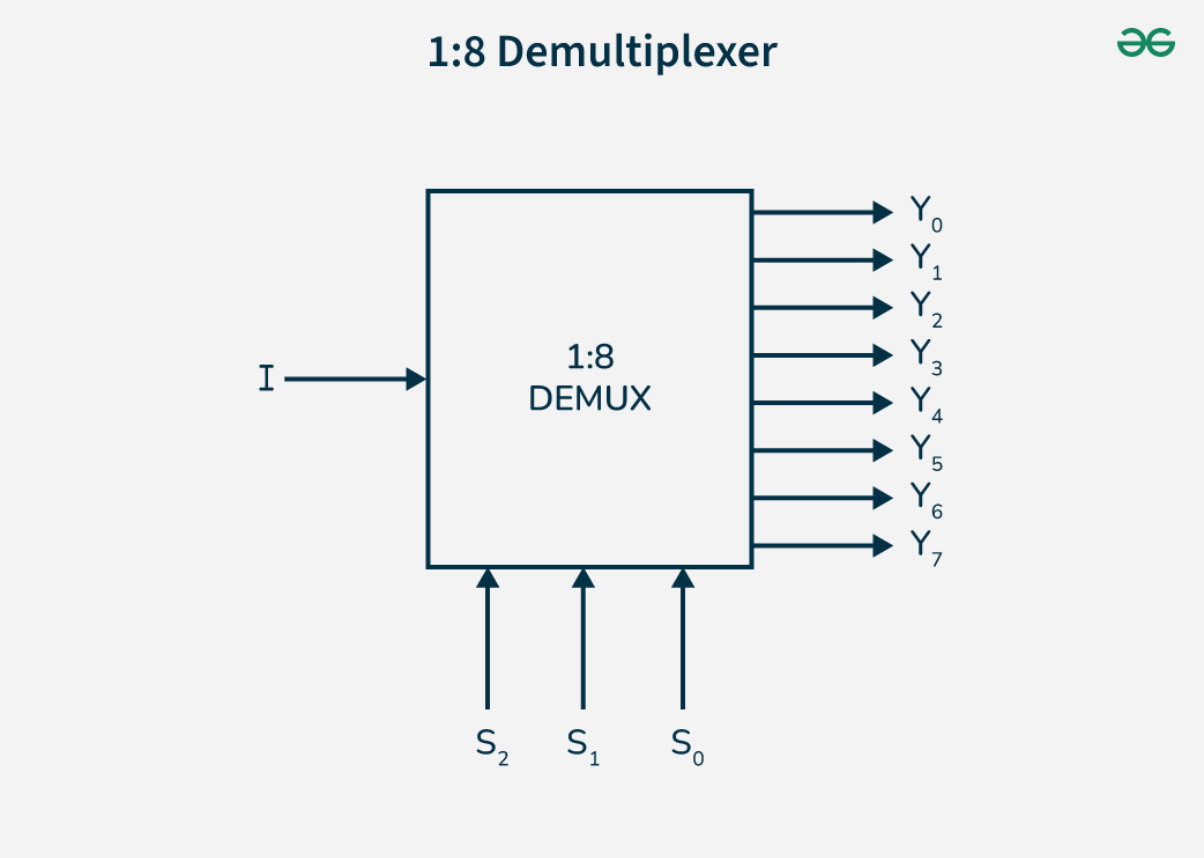


the truth table of the 1x4 DEMUX as mentioned below.

Selection Inputs		Outputs			
S1	**S0**	**Y3**	**Y2**	**Y1**	**Y0**
0	0	0	0	0	**I**
0	1	0	0	**I**	0
1	0	0	**I**	0	0
1	1	**I**	0	0	0

Truth Table Of A 1x8 De-Multiplexer

The 1x8 DEMUX was designed by using two DEMUX. They are the two 1x4 DEMUX and one 1x2 DEMUX. The 1x8 DEMUX contains two input lines with four outputs. Let us see the block diagram of the 1x8 DEMUX as mentioned below.

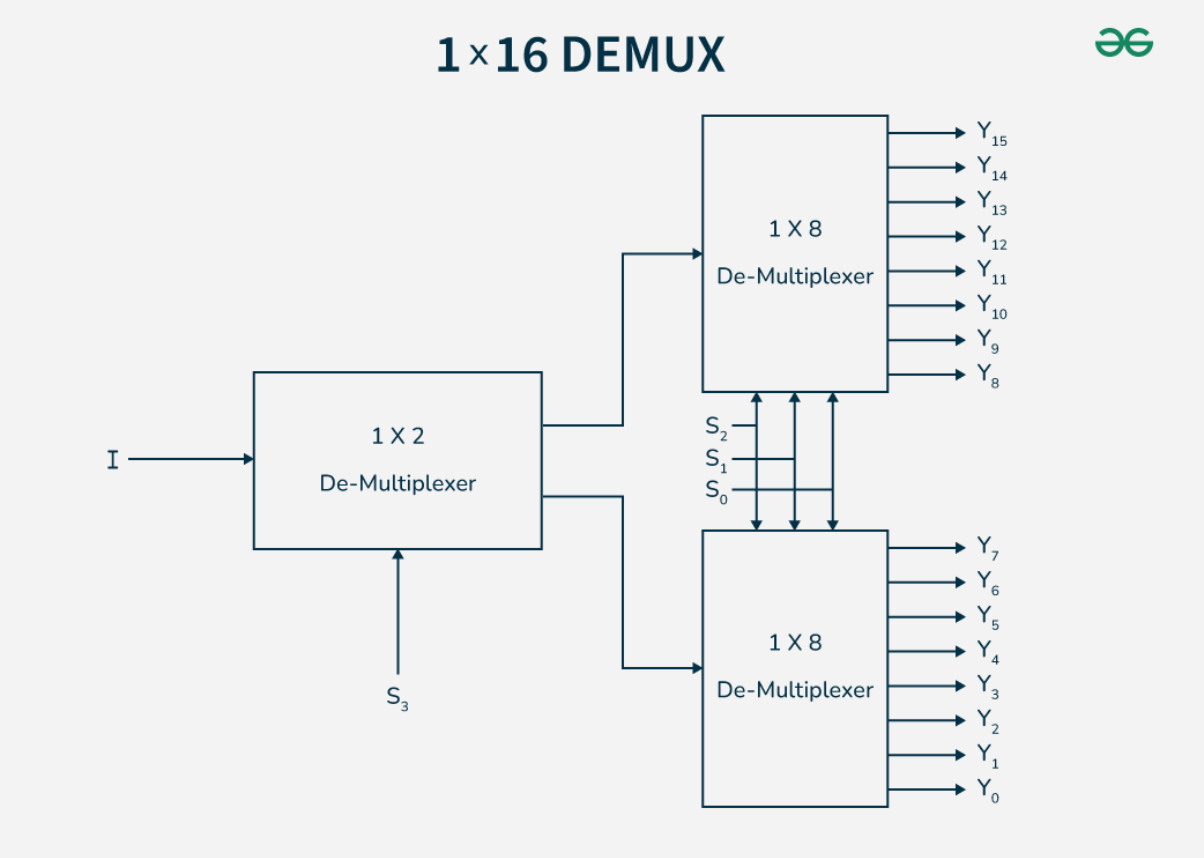


Here is the 1x8 DEMUX truth table as mentioned below.

Selection Inputs			Outputs							
S2	**S1**	**S0**	**Y7**	**Y6**	**Y5**	**Y4**	**Y3**	**Y2**	**Y1**	**Y0**
0	0	0	0	0	0	0	0	0	0	**I**
0	0	1	0	0	0	0	0	0	**I**	0
0	1	0	0	0	0	0	0	**I**	0	0
0	1	1	0	0	0	0	**I**	0	0	0
1	0	0	0	0	0	**I**	0	0	0	0
1	0	1	0	0	**I**	0	0	0	0	0
1	1	0	0	**I**	0	0	0	0	0	0
1	1	1	**I**	0	0	0	0	0	0	0

Truth Table Of A 1x16 De-Multiplexer

The 1x16 DEMUX was designed by using the two 1x8 DEMUX and one 1x2 DEMUX. The 1x16 DEMUX have two input lines. It has eight outputs. Let us see the block diagram of the 1x16 DEMUX as mentioned below.



1:16 DEMUX

Here is the 1x16 DEMUX truth table as mentioned below:

SELECTION INPUTS				**OUTPUTS**															
S3	S2	S1	S0	Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	A	0	0
0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	A	0	0	0

SELECTION INPUTS				OUTPUTS															
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	0	A	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	A	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	A	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	A	0	0	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	A	0	0	0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	A	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	A	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	A	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Advantages and Disadvantages of the DEMUX

Now, we are going to discuss the advantages and disadvantages of the DEMUX as mentioned below.

Advantages of the DEMUX

- The DEMUX increases the efficiency of the particular communication system as it takes data from a specific input source and distributes it to different sources.

- The DEMUX helps to separate the different signals from the mixed data sources. Then it distributes these data to different sources.
- DEMUX can decode the signal outputs of the multiplexer, as the system works in a reverse way of the MUX.

Disadvantages of the DEMUX

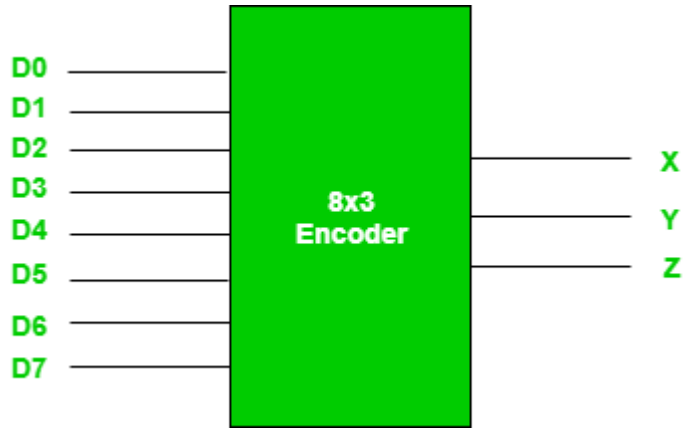
- The DEMUX may cause a wastage of bandwidth as it distributes the refined data in different channels. These channels can overlap with each other which leads to the loss of signal.
- The DEMUX may cause problems in the synchronization of signals. The data channels can overlap with each other which leads to the delay in the whole process.

Encoders and Decoders in Digital Logic

Binary code of N digits can be used to store 2N distinct elements of coded information. This is what encoders and decoders are used for.

Encoders convert 2N lines of input into a code of N bits and **Decoders** decode the N bits into 2N lines.

1. Encoders – An encoder is a combinational circuit that converts binary information in the form of a 2N input lines into N output lines, which represent N bit code for the input. For simple encoders, it is assumed that only one input line is active at a time. As an example, let's consider **Octal to Binary** encoder. As shown in the following figure, an octal-to-binary encoder takes 8 input lines and generates 3 output lines.



Truth Table –

D7	D6	D5	D4	D3	D2	D1	D0	X	Y	Z
0	0	0	0	0	0	0	1	0	0	0

D7	D6	D5	D4	D3	D2	D1	D0	X	Y	Z
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

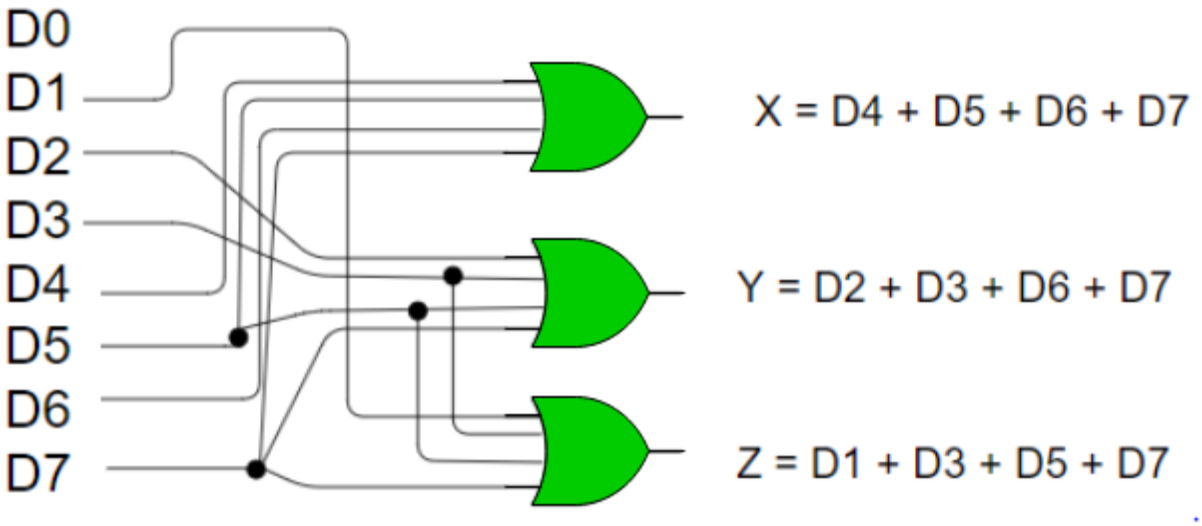
As seen from the truth table, the output is 000 when D0 is active; 001 when D1 is active; 010 when D2 is active and so on.

Implementation –

From the truth table, the output line Z is active when the input octal digit is 1, 3, 5 or 7. Similarly, Y is 1 when input octal digit is 2, 3, 6 or 7 and X is 1 for input octal digits 4, 5, 6 or 7. Hence, the Boolean functions would be:

X = D4 + D5 + D6 + D7
Y = D2 +D3 + D6 + D7
Z = D1 + D3 + D5 + D7

Hence, the encoder can be realised with OR gates as follows:



One limitation of this encoder is that only one input can be active at any given time. If more than one inputs are active, then the output is undefined. For example, if D6 and D3 are both active, then, our output would be 111 which is the output for D7. To overcome this, we use Priority Encoders. Another ambiguity arises when all inputs are 0. In this case, encoder outputs 000 which actually is the output for D0 active. In order to avoid this, an extra bit can be added to the output, called the valid bit which is 0 when all inputs are 0 and 1 otherwise.

Priority Encoder –

A priority encoder is an encoder circuit in which inputs are given priorities. When more than one inputs are active at the same time, the input with higher priority takes precedence and the output corresponding to that is generated. Let us consider the 4 to 2 priority encoder as an example. From the truth table, we see that when all inputs are 0, our V bit or the valid bit is zero and outputs are not used. The x's in the table show the don't care condition, i.e, it may either be 0 or 1. Here, D3 has highest priority, therefore, whatever be the other inputs, when D3 is high, output has to be 11. And D0 has the lowest priority, therefore the output would be 00 only when D0 is high and the other input lines are low. Similarly, D2 has higher priority over D1 and D0 but lower than D3 therefore the output would be 010 only when D2 is high and D3 are low (D0 & D1 are don't care).

Truth Table –

D3	D2	D1	D0	X	Y	V
0	0	0	0	x	x	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1

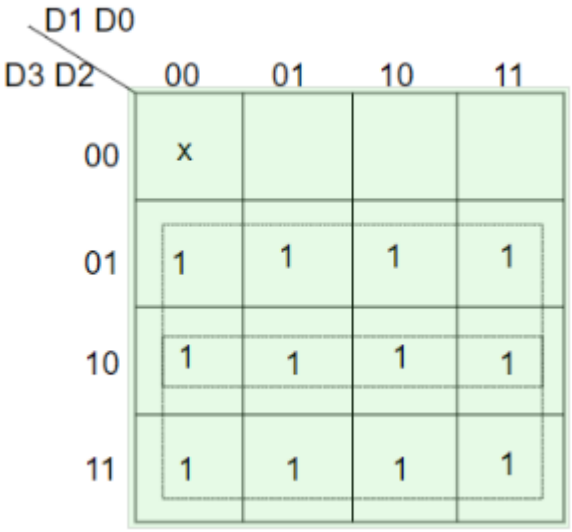
D3	D2	D1	D0	X	Y	V
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Implementation –

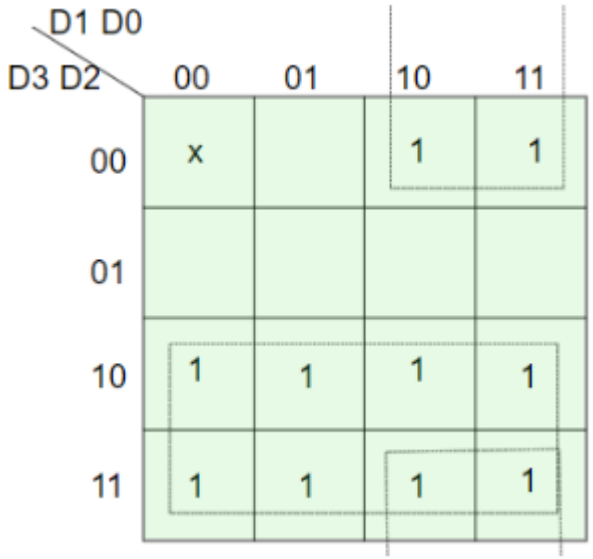
It can clearly be seen that the condition for valid bit to be 1 is that at least any one of the inputs should be high. Hence,

V = D0 + D1 + D2 + D3

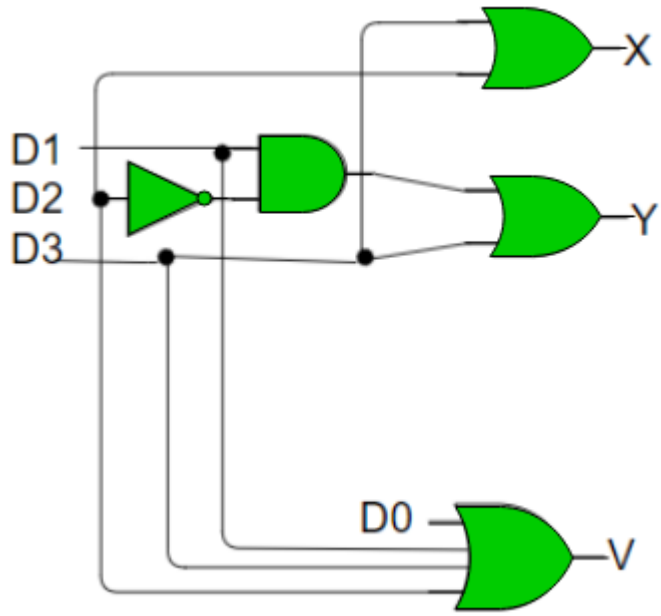
For X:



=> X = D2 + D3 For Y:



=> Y = D1 D2’ + D3 Hence, the priority 4-to-2 encoder can be realized as follows:



2. Decoders –

A decoder does the opposite job of an encoder. It is a combinational circuit that converts n lines of input into 2

n

lines of output. Let’s take an example of 3-to-8 line decoder.

Truth Table –

X	Y	Z	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0

X	Y	Z	D0	D1	D2	D3	D4	D5	D6	D7
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Implementation –

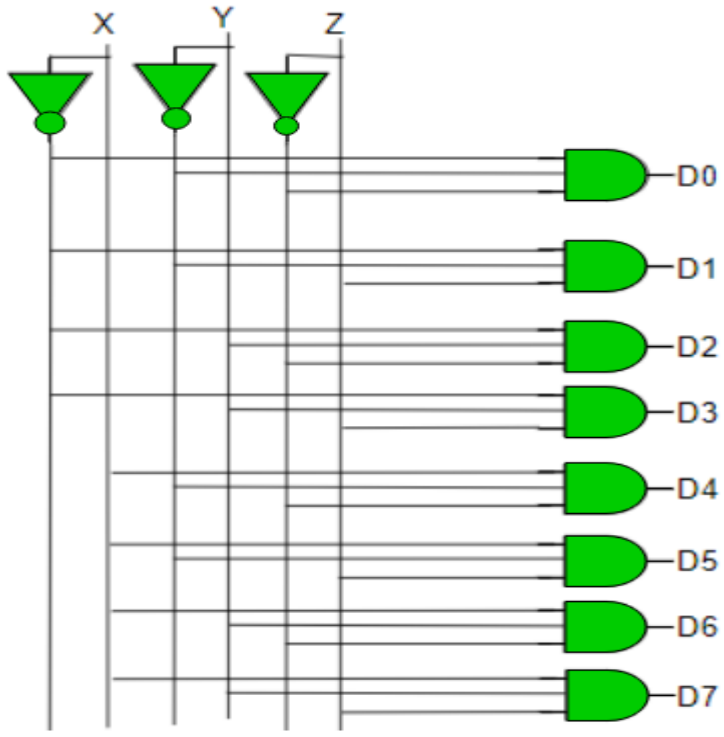
D0 is high when X = 0, Y = 0 and Z = 0. Hence,

D0 = X' Y' Z'

Similarly,

D1 = X' Y' Z
D2 = X' Y Z'
D3 = X' Y Z
D4 = X Y' Z'
D5 = X Y' Z
D6 = X Y Z'
D7 = X Y Z

Hence,

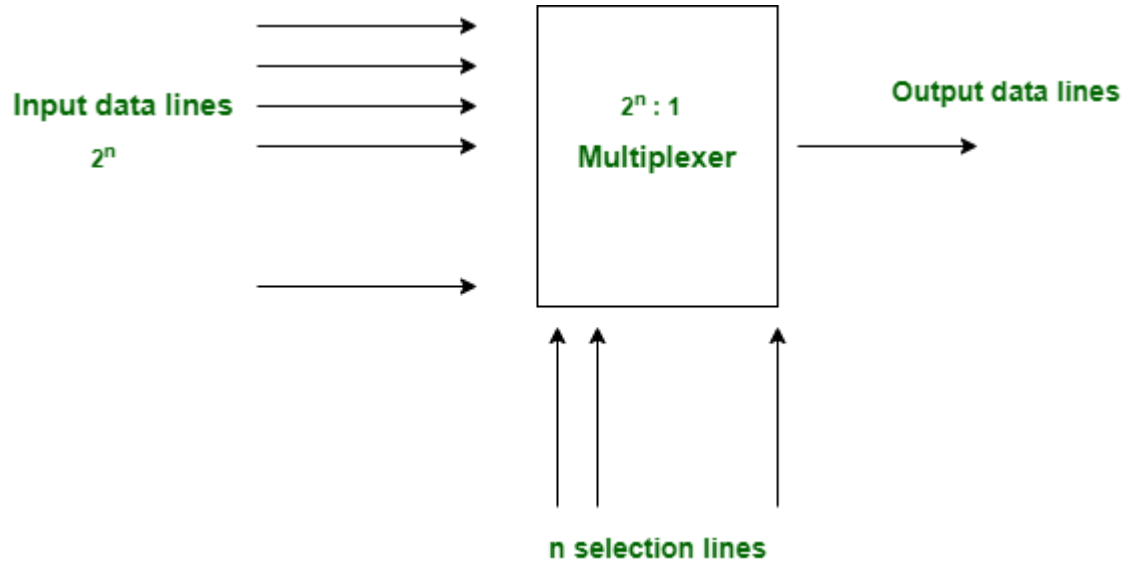


Difference between Multiplexer and Demultiplexer

What is a Multiplexer?

A multiplexer is a data selector which takes several inputs and gives a single output. In multiplexer, we have 2n Input lines and 1 output lines where n is the number of selection lines.

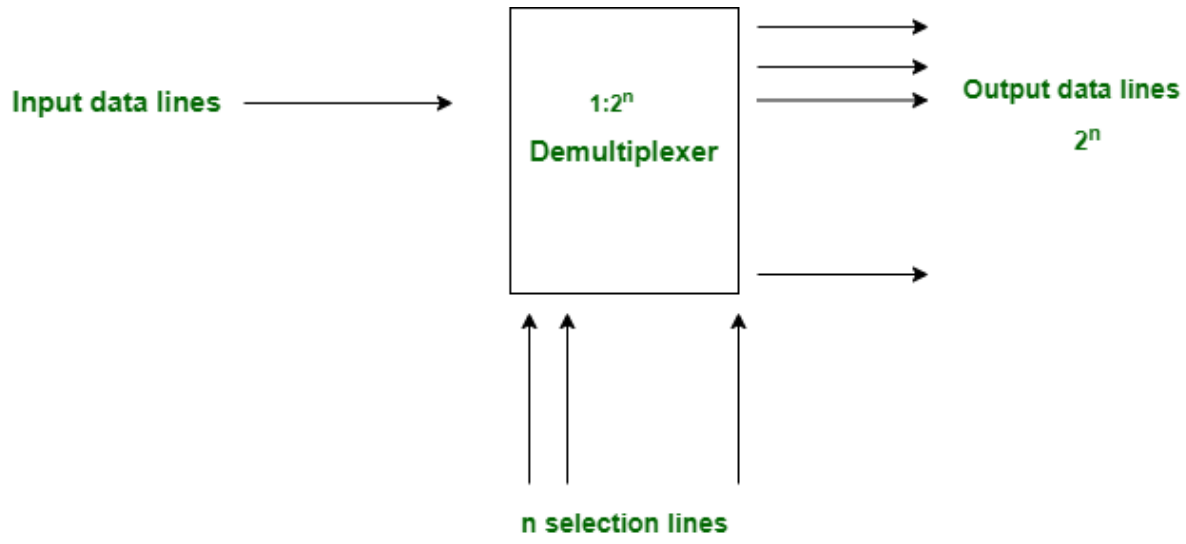
Given Below is the Block Diagram of the Multiplexer, It will have 2n Input lines and will select output based on the Select line.



What is Demultiplexer ?

Demultiplexer is a data distributor which takes a single input and gives several outputs. In demultiplexer we have 1 input and 2n output lines where n is the selection line.

Given below is the block diagram of the Demultiplexer, It will have one Input line and will give 2n output lines.



Difference Between of Multiplexer and Demultiplexer

Given Below is the Table for the Difference between Multiplexer and Demultiplexer

Multiplexer	**Demultiplexer**
Multiplexer processes the digital information from various sources into a single source.	Demultiplexer receives digital information from a single source and converts it into several sources
It is known as Data Selector	It is known as Data Distributor
Multiplexer is a digital switch	Demultiplexer is a digital circuit
It follows combinational logic type	It also follows combinational logic type
It has 2n input data lines	It has single input line
It has a single output data line	It has 2n output data lines
It works on many to one operational principle	It works on one to many operational principle
In time division Multiplexing, multiplexer is used at the transmitter end	In time division Multiplexing, demultiplexer is used at the receiver end

REALIZATION OF BOOLEAN FUNCTIONS USING DECODERS

Realizing Boolean functions using **decoders** involves leveraging the decoders' capability to generate minterms or truth table rows corresponding to input combinations. A **decoder** is a combinational circuit that converts an n-bit input to a unique output among 2n2^n2n outputs, with only one active at a time. This makes it ideal for implementing Boolean functions. Here's how Boolean functions can be realized:

Steps to Realize Boolean Functions Using Decoders:

1. **Determine the Inputs and Outputs of the Function:**
   - Identify the number of variables (n) in the Boolean function.
   - Choose a 2^n-line decoder that matches the number of variables.
2. **Connect the Inputs of the Decoder:**
   - The inputs of the decoder correspond to the variables of the Boolean function.
   - For example, if there are 3 variables (A, B, C), use a $3 \rightarrow 8$-line decoder.
3. **Find the Minterm Representation:**
   - Express the given Boolean function in terms of its **sum of minterms (SOP)** form.
4. **Select Relevant Outputs of the Decoder:**
   - Each output of the decoder corresponds to a minterm (truth table row). Identify the outputs that are included in the SOP form of the function.
5. **Combine Outputs Using OR Gates:**
   - Use an OR gate to combine the selected outputs of the decoder.
   - The output of the OR gate gives the realization of the Boolean function.

Example:

Realize $F(A, B, C) = \Sigma(1, 3, 5, 7)$ Using a 3-to-8 Decoder

1. **Inputs and Decoder:**
   - The function has 3 variables (A, B, C), so use a $3 \rightarrow 8$-line decoder.
2. **Connect the Decoder Inputs:**
   - Connect A, B, and C as inputs to the decoder.
3. **Identify Minterms:**
   - $F(A, B, C) = \Sigma(1, 3, 5, 7)$ corresponds to the decoder outputs Y_1, Y_3, Y_5, and Y_7.
4. **Connect Outputs to an OR Gate:**
   - Combine Y_1, Y_3, Y_5, and Y_7 using a 4-input OR gate.
5. **Final Circuit:**
   - The OR gate output represents $F(A, B, C)$.

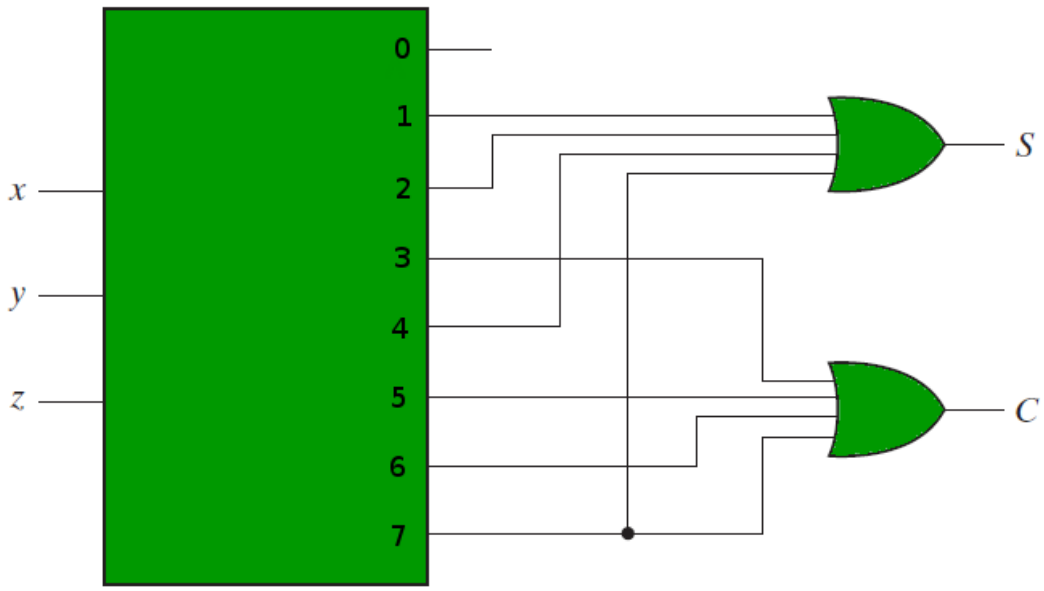
Circuit Diagram:

- **Decoder:** A $3 \rightarrow 8$-line decoder with inputs A, B, C and outputs Y_0 to Y_7.
- **OR Gate:** Combines outputs Y_1, Y_3, Y_5, and Y_7.

Advantages of Using Decoders:

1. **Efficient Implementation:** Decoders simplify the design of Boolean functions, especially for SOP forms.
2. **Scalability:** Suitable for implementing functions with multiple variables.
3. **Ease of Expansion:** Functions can be updated by modifying OR gate connections.

The following circuit diagram shows the implementation of Full adder using a 3:8 Decoder and OR gates.



REALIZATION OF BOOLEAN FUNCTIONS USING MULTIPLEXER

Realization of Boolean Functions Using Multiplexers involves implementing a Boolean function by connecting its variables and constants to a multiplexer (MUX). A MUX can directly implement any Boolean function by selecting the appropriate inputs based on the truth table of the function.

Steps to Realize Boolean Functions Using Multiplexers:

1. **Determine the Number of Variables:**
   - Identify the number of variables (n) in the Boolean function.
   - Use a 2^n-to-1 multiplexer or a smaller one by treating some variables as selection lines and others as direct inputs.
2. **Assign Variables to Selection Lines:**
   - Connect n variables to the selection lines of the MUX.
3. **Find the Truth Table or Canonical Form:**
   - Write the Boolean function in **truth table** form or **Sum of Minterms (SOP)**.
4. **Configure the MUX Inputs:**
   - For a 2^n-to-1 MUX, each input corresponds to a minterm.
   - Connect the inputs of the MUX ($I_0, I_1, \ldots, I_{2^n-1}$) to 0, 1, or the desired variable based on the truth table.
5. **Output Connection:**
   - The MUX output gives the result of the Boolean function.

Example 1: Realizing $F(A, B) = A + B$ Using a 2-to-1 MUX

1. **Expression Simplification:**
   - The function $F(A, B) = A + B$ can be written as:

     $$F = B \cdot I_0 + \overline{B} \cdot I_1$$

     Here, $I_0 = 1$ and $I_1 = A$.
2. **Use a 2-to-1 MUX:**
   - Selection line: B.
   - Inputs: $I_0 = 1, I_1 = A$.
3. **MUX Configuration:**
   - Connect B to the selection line.
   - Set $I_0 = 1$ and $I_1 = A$.
   - The output is $F(A, B)$.

Example 2: Realizing $F(A, B, C) = \Sigma(1, 3, 5, 7)$ Using a 4-to-1 MUX

1. **Function in SOP Form:**
   - The minterms are $1(001), 3(011), 5(101), 7(111)$.
2. **Connect Variables:**
   - Selection lines: B and C.
   - The remaining variable A is used to configure inputs $I_0, I_1, \ldots, I_3$.
3. **MUX Inputs:**
   - Based on $F(A, B, C)$, determine inputs:
     - $I_0 = 0$ (minterm 0 is not present).
     - $I_1 = A$ (minterm 1).
     - $I_2 = A$ (minterm 2).
     - $I_3 = A$ (minterm 3).
4. **MUX Output:**
   - The 4-to-1 MUX combines the inputs to produce $F(A, B, C)$.

Example 3: Using a Larger MUX for Simplicity

For $F(A, B, C, D) = \Sigma(1, 5, 8, 13)$:

1. Use an $8 - to - 1$-MUX:
   - Selection lines: A, B, C.
   - Configure inputs I_0 to I_7 based on the truth table of $F(A, B, C, D)$.
   - Inputs are connected to either D, $\overline{D}$, 1, or 0.

General Approach for Any Boolean Function:

1. **Truth Table or Minterm Expression:** Identify which minterms are 1 (function's SOP form).
2. **Select MUX Size:** Use a 2^n-to-1 MUX or optimize by reducing variables via inputs.
3. **Configure Inputs:** Assign each input (I_k) to 0, 1, or a variable based on the function.
4. **Connect Selection Lines:** The selection lines determine which input is passed to the output.

UNIT – V

Sequential Logic Circuits

Classification of sequential circuits

Sequential circuits are digital circuits where the output depends not only on the current inputs but also on the history of past inputs. This is achieved using memory elements like flip-flops or latches, which store the previous state of the circuit.

Sequential circuits are broadly classified into the following categories:

---

1. Synchronous Sequential Circuits

- **Definition:** The state of the circuit changes only at discrete times, synchronized with a clock signal.
- **Key Features:**
    - State transitions occur only at clock edges (rising or falling).
    - Use flip-flops as memory elements.
- **Advantages:**
    - Easier to design and analyze due to predictable state changes.
    - Less sensitive to noise compared to asynchronous circuits.
- **Examples:**
    - Counters (e.g., binary counter, ring counter).
    - Shift registers.
    - Synchronous finite state machines (FSM).

---

2. Asynchronous Sequential Circuits

- **Definition:** The state of the circuit can change at any time, as soon as there is a change in the input.
- **Key Features:**
    - Do not rely on a clock signal.
    - State transitions depend solely on input changes and internal states.
    - More prone to timing issues like glitches or race conditions.
- **Advantages:**
    - Faster than synchronous circuits, as they are not restricted by the clock.
- **Examples:**

- Asynchronous FSM.
- Some types of digital communication circuits.

---

3. Classification Based on State Machine Type

Sequential circuits can also be classified based on how their outputs are generated:

a. Mealy Machine

- **Definition:** Output depends on both the current state and the current input.
- **Characteristics:**
  - Outputs may change immediately with changes in inputs.
  - Typically requires fewer states than a Moore machine for the same functionality.
- **Example:** Traffic light controller where the output depends on input sensors.

b. Moore Machine

- **Definition:** Output depends only on the current state, not directly on the input.
- **Characteristics:**
  - Outputs are updated only at state transitions.
  - Simpler to design than a Mealy machine but may require more states.
- **Example:** Elevator control system where the output is determined by the current state of the system.

---

4. Classification Based on Memory Elements

Sequential circuits can also be classified based on the type of memory elements they use:

a. Latch-Based Circuits

- Use latches (e.g., SR latch, D latch) as memory elements.
- Typically used in asynchronous designs.

b. Flip-Flop-Based Circuits

- Use flip-flops (e.g., D flip-flop, JK flip-flop, T flip-flop) as memory elements.
- Commonly used in synchronous designs.

---

5. Classification Based on Application

a. Counters

- Sequential circuits designed to count events or clock pulses.

- Examples: Up counter, down counter, modulo counter.

b. Shift Registers

- Circuits that shift binary data left or right.
- Examples: Serial-in serial-out (SISO), serial-in parallel-out (SIPO).

c. Finite State Machines (FSMs)

- Abstract models used to design sequential circuits.
- Can be Mealy or Moore machines.

d. Sequence Generators and Detectors

- Circuits designed to produce or detect a specific sequence of bits.

---

6. Other Classifications

a. Clocked vs. Unclocked

- **Clocked:** Use a clock signal for state transitions (e.g., flip-flop-based circuits).
- **Unclocked:** Do not use a clock signal (e.g., purely combinational circuits with feedback).

b. Synchronous vs. Asynchronous Reset

- **Synchronous Reset:** Reset happens at a clock edge.
- **Asynchronous Reset:** Reset happens immediately, irrespective of the clock.

Summary Table

Type	**Key Features**	**Examples**
Synchronous Sequential Circuits	Changes at clock edges	Counters, FSMs
Asynchronous Sequential Circuits	Changes immediately based on inputs	Asynchronous FSMs
Mealy Machine	Output depends on state and inputs	Traffic light controller
Moore Machine	Output depends only on state	Elevator controller
Latch-Based Circuits	Use latches (e.g., SR, D)	Basic storage circuits
Flip-Flop-Based Circuits	Use flip-flops (e.g., JK, T, D)	Shift registers, counters

LATCH AND FLIP-FLOP

Both **latches** and **flip-flops** are basic building blocks of sequential circuits and are used as memory elements to store one bit of data. They differ primarily in how and when they change their state.

---

1. Latches

- **Definition:** A latch is a bistable device that can store one bit of information and is **level-sensitive**, meaning its output changes as long as the input changes and the enable signal is active.
- **Key Characteristics:**
    - Operates continuously when enabled.
    - Changes state immediately in response to input changes (asynchronous behavior).
    - Often used in asynchronous circuits or as intermediate storage.

Types of Latches:

1. SR (Set-Reset) Latch:
    - Two inputs: Set (S) and Reset (R).
    - Behavior:
        - $S = 1, R = 0$: Sets output $Q = 1$.
        - $S = 0, R = 1$: Resets output $Q = 0$.
        - $S = 0, R = 0$: Maintains the previous state.
        - $S = 1, R = 1$: Undefined state (not allowed).
2. D (Data) Latch:
    - Single input D and an enable signal.
    - Behavior:
        - When enabled, $Q = D$.
        - When not enabled, Q retains its value.

2. Flip-Flops

- **Definition:** A flip-flop is a bistable device that stores one bit of information and is **edge-triggered**, meaning its output changes only at the edge (rising or falling) of a clock signal.
- **Key Characteristics:**
    - Operates synchronously with the clock signal.
    - Output changes only at specific clock edges, making it less sensitive to noise.
    - Used in synchronous circuits.

Types of Flip-Flops:

1. SR (Set-Reset) Flip-Flop:
   - Similar to the SR latch but triggered by a clock edge.
   - Has the same truth table as the SR latch, including the undefined state for $S = 1, R = 1$.
2. D (Data) Flip-Flop:
   - Single input D and a clock signal.
   - Behavior:
     - At the clock edge, $Q = D$.
     - Avoids the undefined state of the SR flip-flop.
     - Acts as a delay element (stores input until the next clock edge).
3. JK Flip-Flop:
   - Two inputs: J (Set) and K (Reset).
   - Behavior:
     - $J = 0, K = 0$: No change.
     - $J = 0, K = 1$: Resets $Q = 0$.
     - $J = 1, K = 0$: Sets $Q = 1$.
     - $J = 1, K = 1$: Toggles Q (complements its value).
4. T (Toggle) Flip-Flop:
   - Single input T and a clock signal.
   - Behavior:
     - $T = 0$: No change.
     - $T = 1$: Toggles Q.
   - Commonly used in counters.

Differences Between Latches and Flip-Flops

Feature	Latch	Flip-Flop
Triggering Mechanism	Level-sensitive	Edge-triggered
Clock Dependency	May or may not depend on a clock	Requires a clock signal
State Change	Continuous when enabled	Only at clock edges
Design Complexity	Simpler	More complex
Use Cases	Asynchronous circuits	Synchronous circuits
Susceptibility to Noise	Higher (due to continuous operation)	Lower (state changes are clocked)

RS- LATCH USING NAND AND NOR GATES TRUTH TABLES, RS, JK, T AND D FLIP-FLOPS, TRUTH AND EXCITATION TABLES

What are Latches?

Latches are digital circuits that store a single bit of information and hold its value until it is updated by new input signals. They are used in digital systems as temporary storage elements to store binary information. Latches can be implemented using various digital logic gates, such as AND, OR, NOT, NAND, and NOR gates.

Latches are widely used in digital systems for various applications, including data storage, control circuits, and flip-flop circuits. They are often used in combination with other digital circuits to implement sequential circuits, such as state machines and memory elements.

Latches Definition

Latches are basic storage elements that operate with signal levels (rather than signal transitions). Latches controlled by a clock transition are flip-flops. Latches are level-sensitive devices. Latches are useful for the design of the asynchronous sequential circuit. Latches are sequential circuit with two stable states. These are sensitive to the input voltage applied and does not depend on the clock pulse. Flip flops that do not use clock pulse are referred to as latch.

SR Latch

S-R latches i.e., Set-Reset latches are the simplest form of latches and are implemented using two inputs: S (Set) and R (Reset). The S input sets the output to 1, while the R input resets the output to 0. When both S and R inputs are at 1, the latch is said to be in an “undefined” state. They are also known as preset and clear states. The SR latch forms the basic building blocks of all other types of flip-flops.

Truth Table of SR Latch

The below table represents the truth table of SR latch.

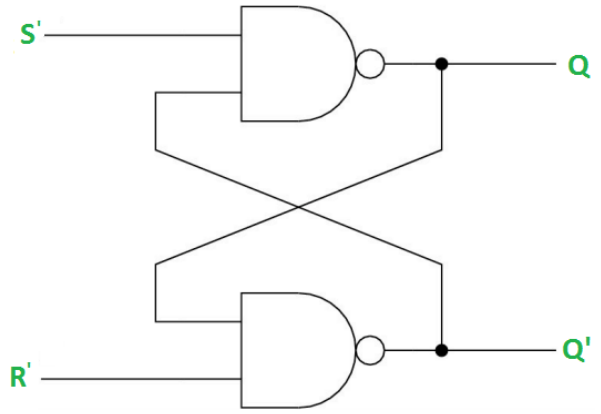
S	R	Q	Q’
0	0	Latch	Latch
0	1	0	1
1	0	1	0
1	1	0	0

Logic Diagram of SR Latch

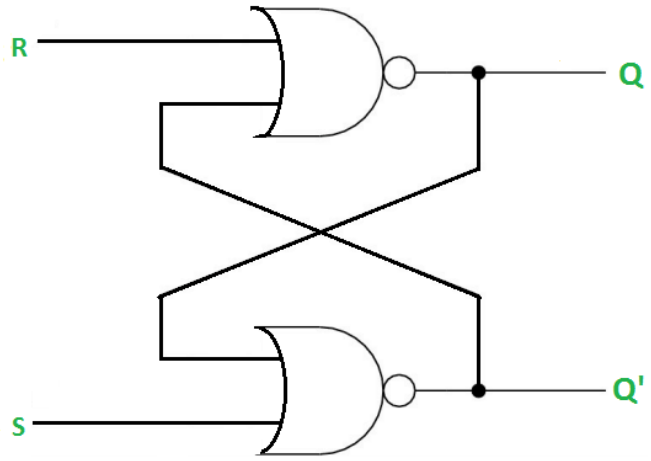
SR Latch is a logic circuit with:

- 2 cross-coupled NOR gate or 2 cross-coupled NAND gate.
- 2 input S for SET and R for RESET
- 2 output Q, Q'.

The below logic diagram represents the SR latch using NAND gate.



The below logic diagram represents SR latch using NOR Gate.



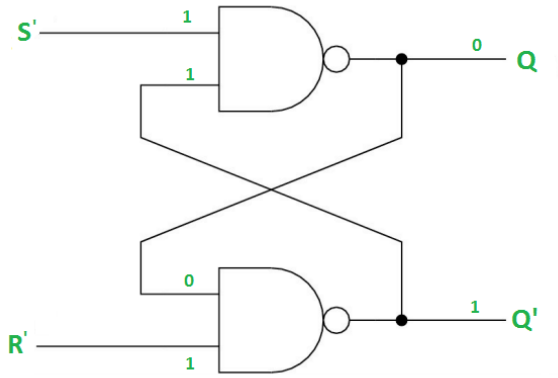
Different Cases of SR Latch

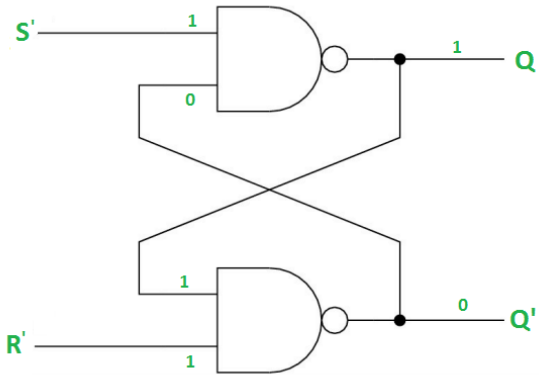
The different cases of SR latch are discussed below.

Case 1: S' = R' = 1 (S = R = 0)

If Q = 1, Q and R' inputs for 2nd NAND gate are both 1.

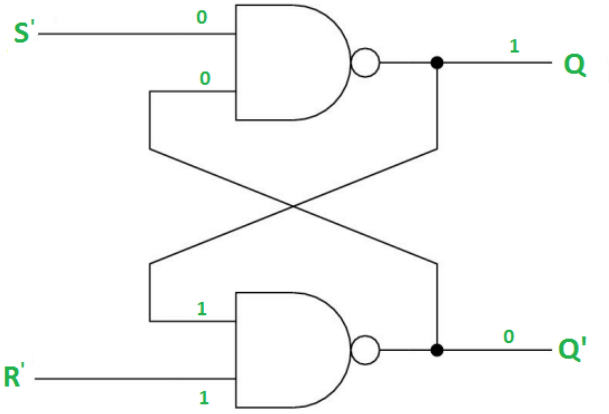
If Q = 0, Q and R' inputs for 2nd NAND gate are 0 and 1 respectively.





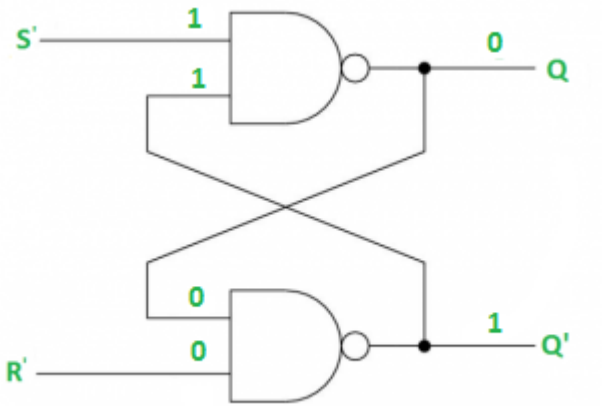
Case 2: S' = 0, R' = 1 (S = 1, R = 0)

- As S' = 0, the output of 1st NAND gate, Q = 1 (**SET state**).
- In second NAND gate, as Q and R' inputs are 1, Q'=0.



Case 3: S' = 1, R' = 0 (S = 0, R = 1)

- As R'=0, the output of 2nd NAND gate, Q' = 1.
- In first NAND gate, as Q and S' inputs are 1, Q = 0 (**RESET state**).



Case 4: S' = R' = 0 (S = R = 1)

When S = R = 1, both Q and Q' becomes 1 which is not allowed. So, the input condition is prohibited.

Gated SR Latch

A Gated SR latch is a SR latch with enable input which works when enable is 1 and retain the previous state when enable is 0.

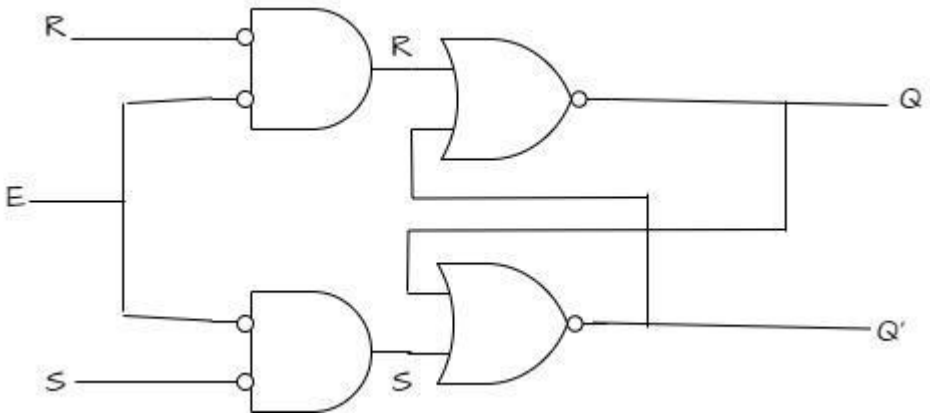
Truth Table of Gated SR Latch

The below table represents the truth table of Gated SR latch.

Enable	S	R	Qn+1
0	X	X	Qn
1	0	0	Qn
1	0	1	0
1	1	0	1
1	1	1	X

Logic Diagram of Gated SR Latch

The below logic diagram represents the gated SR latch.



Logic Diagram of Gated SR Latch

D Latch

D latches are also known as transparent latches and are implemented using two inputs: D (Data) and a clock signal. The output of the latch follows the input at the D terminal as long as the clock signal is high. When the clock signal goes low, the output of the latch is stored and held until the next rising edge of the clock.

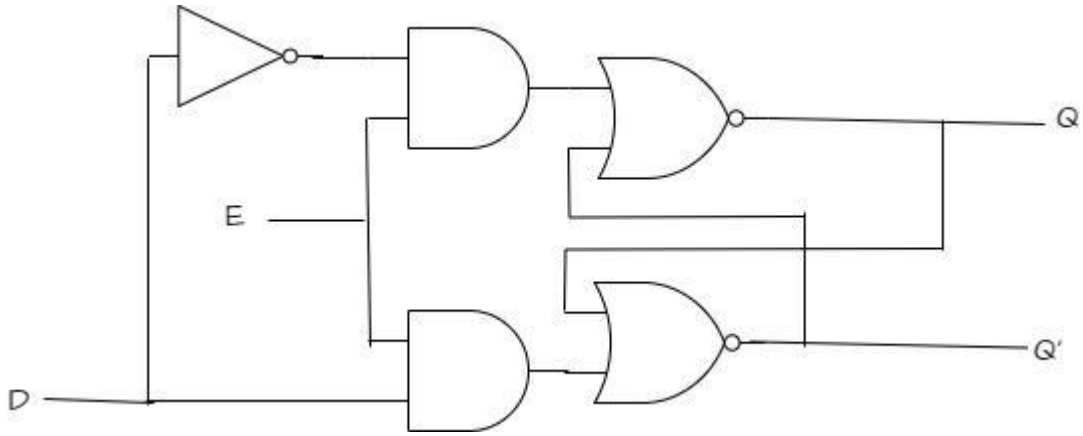
Truth Table of D Latch

The below table represents the truth table of D latch.

E	D	Q	Q’
0	0	Latch	Latch
0	1	Latch	Latch
1	0	0	1
1	1	1	0

Logic Diagram of D Latch

The below logic diagram represents the D latch.



Logic Diagram of D Latch

Gated D Latch

D latch is similar to SR latch with some modifications made. Here, the inputs are complements of each other. The D latch stands for “data latch” as this latch stores single bit temporarily.

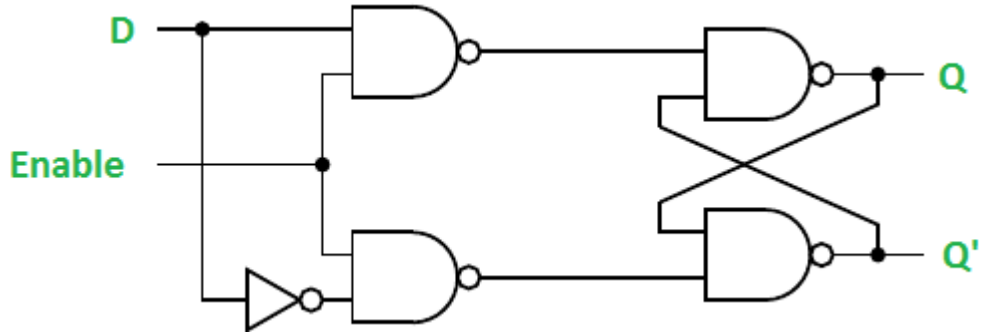
Truth Table of Gated D Latch

The below table represents the truth table of Gated D latch.

Enable	D	Qn	Qn+1	STATE
1	0	x	0	RESET
1	1	x	1	SET
0	x	x	Q(n)	No Change
Characteristics Equation: **Qn+1 = EN.D + EN'.Qn**				

Logic Diagram of Gated D Latch

The below logic diagram represents the gated D latch.



JK Latch

JK latch has two inputs J and K. The output gets toggled when the J and K inputs are high. JK latch is just like SR latch, but it eliminates the undefined state of SR latch.

Truth Table of JK Latch

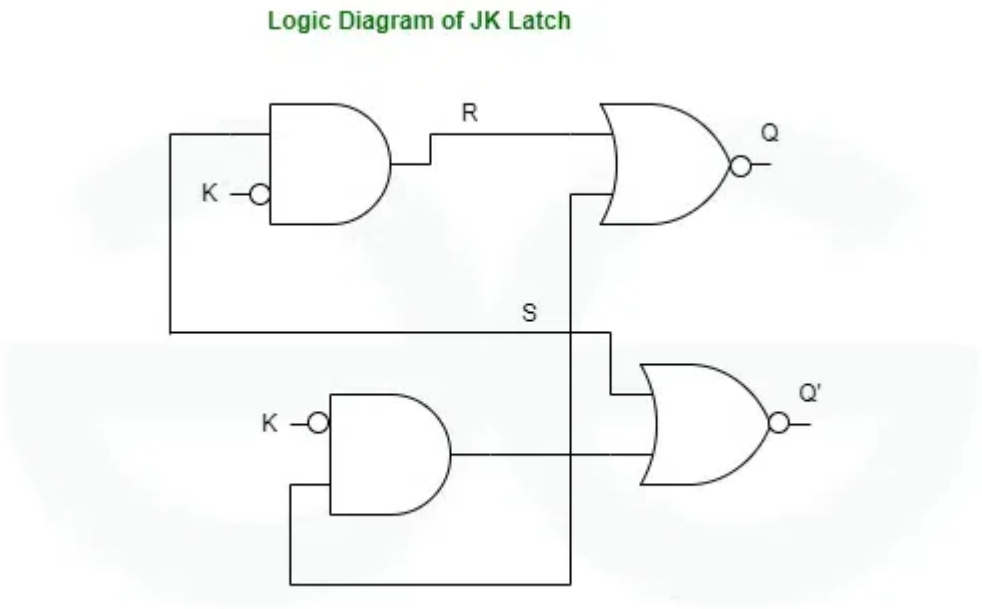
The below table represents the truth table of JK latch.

J	K	Qn+1	Comment
0	0	Q	No change
0	1	0	Reset

J	K	Qn+1	Comment
1	0	1	Set
1	1	Q’	Toggle

Logic Diagram of JK Latch

The below logic diagram represents the JK latch.



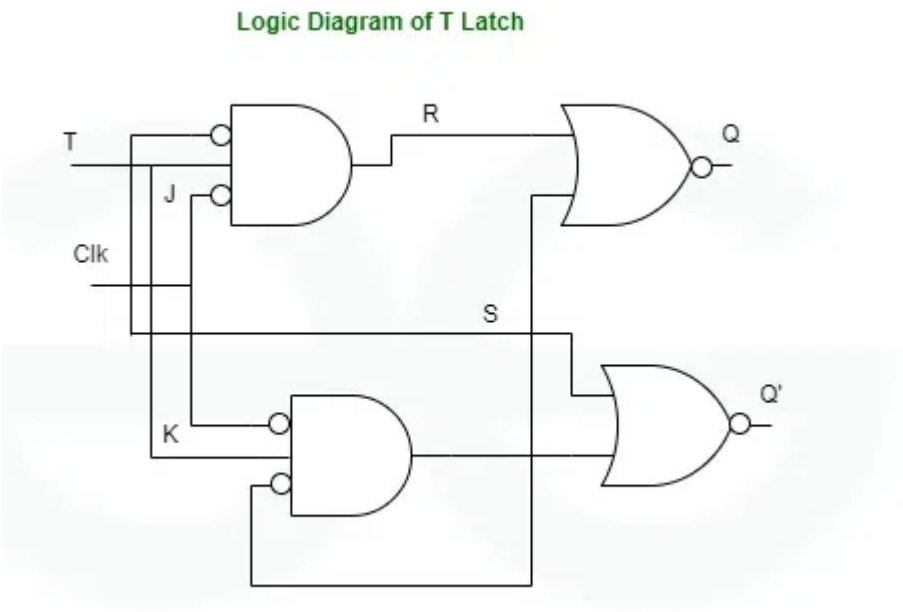
Logic Diagram of JK Latch

T Latch

When the JK inputs of JK latch are shorted we get the T latch. In T latch the outputs are toggled when the inputs are high.

Logic Diagram of T Latch

The below logic diagram represents the T latch.



Logic Diagram of T Latch

Advantages of Latches

Some of the advantages of latches are listed below.

1. **Easy to Implement:** Latches are simple digital circuits that can be easily implemented using basic digital logic gates.
2. **Low Power Consumption:** Latches consume less power compared to other sequential circuits such as flip-flops.
3. **High Speed:** Latches can operate at high speeds, making them suitable for use in high-speed digital systems.
4. **Low Cost:** Latches are inexpensive to manufacture and can be used in low-cost digital systems.
5. **Versatility:** Latches can be used for various applications, such as data storage, control circuits, and flip-flop circuits.

Disadvantages of Latches

Some of the disadvantages of latches are listed below.

1. **No Clock:** Latches do not have a clock signal to synchronize their operations, making their behavior unpredictable.
2. **Unstable State:** Latches can sometimes enter into an unstable state when both inputs are at 1. This can result in unexpected behavior in the digital system.
3. **Complex Timing:** The timing of latches can be complex and difficult to specify, making them less suitable for real-time control applications.

CONVERSION OF FLIP- FLOPS

What is a Flip-Flop?

The flip-flop is a circuit that maintains a state until directed by input to change the state. A basic flip-flop can be constructed using four-NAND or four-NOR gates. Flip-flop is popularly known as the basic digital memory circuit. It has its two states as logic 1(High) and logic 0(low) states. A flip flop is a sequential circuit which consist of single binary state of information or data. The digital circuit is a flip flop which has two outputs and are of opposite states. It is also known as a Bistable Multivibrator.

Types of Flip-Flops

Given Below are the Types of Flip-Flop

- SR Flip Flop
- JK Flip Flop
- D Flip Flop
- T Flip Flop

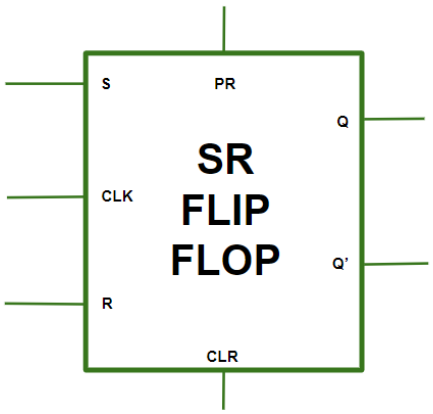
Logic diagrams and truth tables of the different types of flip-flops are as follows:

S-R Flip Flop

In the flip flop, with the help of preset and clear when the power is switched ON, the states of the circuit keeps on changing, that is it is uncertain. It may come to set(Q=1) or reset(Q'=0) state. In many applications, it is desired to initially set or reset the flip flop that is the initial state of the flip flop that needs to be assigned. This thing is accomplished by the preset(PR) and the clear(CLR).

Block Diagram of S-R Flip Flop

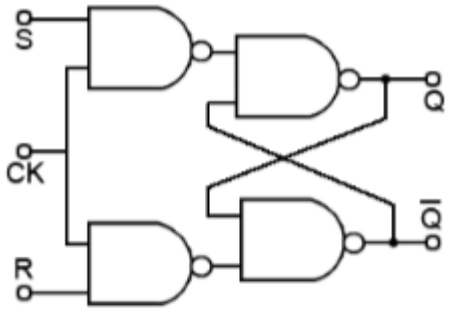
Given Below is the Block Diagram of S-R Flip Flop



S-R Flip Flop

Circuit Diagram and Truth Table of S-R Flip Flop

Given Below is the Diagram of S-R Flip Flop with its Truth Table



TRUTH TABLE

S	R	Q_N	Q_{N+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

Operations of S-R Flip Flop

Given Below is the Operations of S-R Flip Flop

- **Case 1(PR=CLR=1):** The asynchronous inputs are inactive and the flip flop responds freely to the S,R and the CLK inputs in the normal way.
- **Case 2(PR=0 and CLR=1):**This is used when the Q is set to 1.
- **Case 3(PR=1 and CLR=0):**This is used when the Q' is set to 1.
- **Case 4(PR=CLR=0):** This is an invalid state.

Characteristics Equation for SR Flip Flop

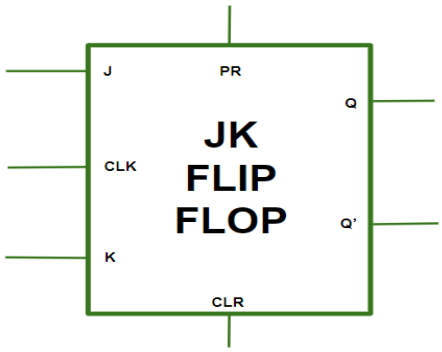
QN+1 = QNR' + SR'

J-K Flip Flop

In JK flip flops, The basic structure of the flip flop which consists of Clock (CLK), Clear (CLR), Preset (PR).

Block Diagram of J-K Flip Flop

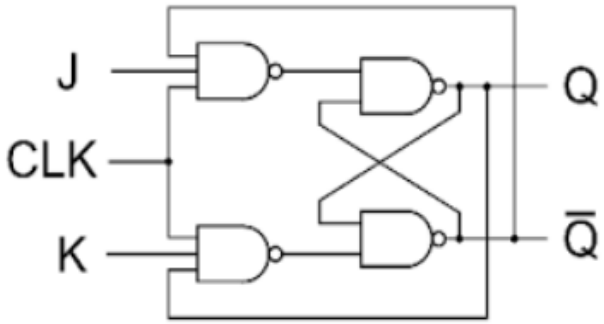
Given Below is Block Diagram of J-K Flip Flop



J-K Flip Flop

Circuit Diagram and Truth Table of J-K Flip Flop

Given Below is the Diagram of J-K Flip Flop with its Truth Table



TRUTH TABLE

J	K	Q_N	Q_{N+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Operations of J-K Flip Flop

Given Below is the Operations of J-K Flip Flop

- **Case 1 (PR=CLR=0):**This condition is in its invalid state.
- **Case 2 (PR=0 and CLR=1):**The PR is activated which means the output in the Q is set to 1. Therefore, the flip flop is in the set state.
- **Case 3 (PR=1 and CLR=0):**The CLR is activated which means the output in the Q' is set to 1. Therefore, the flip flop is in the reset state.
- **Case 4 (PR=CLR=1):**In this condition the flip flop works in its normal way whereas the PR and CLR gets deactivated.

Race Around Condition in J-K Flip Flop

When the J and K both are set to 1, the input remains high for a longer duration of time, then the output keeps on toggling. Toggle means that switching in the output instantly i.e. Q=0, Q'=1 will immediately change to Q=1 and Q'=0 and this continuation keeps on changing. This change in output leads to race around condition.

Characteristics Equation for JK Flip Flop

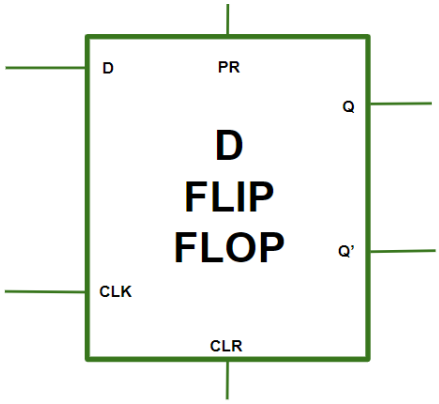
QN+1 = JQ'N + K'QN

D Flip Flop

The D Flip Flop Consists a single data input(D), a clock input(CLK),and two outputs: Q and Q' (the complement of Q).

Block Diagram of D Flip Flop

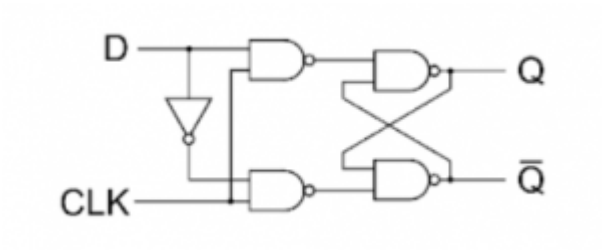
Given Below is the Block Diagram of D Flip Flop



D FLIP FLOP

Circuit Diagram and Truth Table of D Flip Flop

Given Below is the Diagram of D Flip Flop with its Truth Table



Q	D	Q(t+1)
0	0	0
0	1	1
1	0	0
1	1	1

Operation of the D Flip-Flop

Given Below is the operation of D Flip-Flip

- **Case 1 (PR=CLR=0):**This conditions is represents as invalid state where both PR(present) and CLR(clear) inputs are inactive.
- **Case 2 (PR=0 and CLR=1)**:This state is set state in which PR is inactive (0) and CLR is active(1) and the output Q is set to 1.
- **Case 3 (PR=1 and CLR=0):**This state is reset state in which PR is active (1) and CLR is inactive (0) and the complementary output Q' is set to 1.
- **Case 4 (PR=CLR=1):**In This state the flip flop behaves as normal, both PR and CLR inputs are active(1).

Characteristics Equation for D Flip Flop

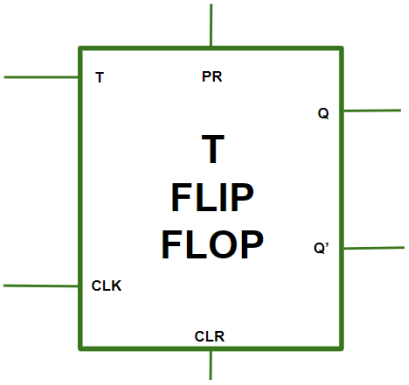
QN+1 = D

T Flip Flop

The T Flip Flop consists of data input (T), a clock input (CLK), and two outputs: Q and Q' (the complement of Q).

Block Diagram of T Flip Flop

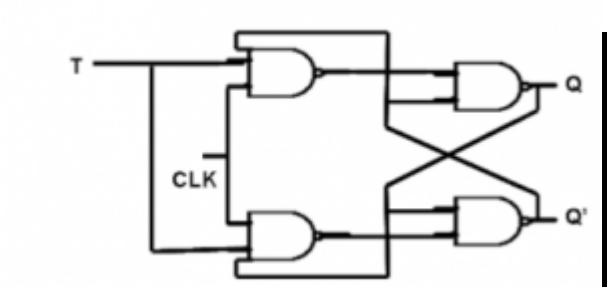
Given Below is the Block Diagram of T Flip Flop



T FLIP FLOP

Circuit Diagram and Truth Table of T Flip Flop

Given Below is the Circuit Diagram and Truth Table of T Flip Flop



T	Q_n	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

Operation of the T Flip-Flop

Given Below is the Operation of T Flip-Flop

- **Case 1 (T=0):**In this condition the **fl**ip-flop remains in its current state regardless of clock input,Also the Output Q will remain unchanged unit the value of T will not change.
- **Case 2 (T=1):**In this condition the flip flop will change when T input is 1,At each rising or falling edge of the clock signal the output Q will be in complementary state.

Characteristics Equation for T Flip Flop

QN+1 = Q'NT + QNT' = QN XOR T

Conversion for Flip Flops

The Excitation Table of the Flip Flop can be given as

EXCITATION TABLE:

Q_N	Q_{N+1}	S	R	J	K	D	T
0	0	0	X	0	X	0	0
0	1	1	0	1	X	1	1
1	0	0	1	X	1	0	1
1	1	X	0	X	0	1	0

Steps To Convert from One Flip Flop to Other

Let there be required flipflop to be constructed using sub-flipflop:

1. Draw the truth table of the required flip-flop.
2. Write the corresponding outputs of sub-flipflop to be used from the excitation table.
3. Draw K-Maps using required flipflop inputs and obtain excitation functions for sub-flipflop inputs.

4. Construct a logic diagram according to the functions obtained.

Convert SR To JK Flip Flop

The Table for the SR To JK is given as

J	K	Q_N	Q_{N+1}	S	R
0	0	0	0	0	X
0	0	1	1	X	0
0	1	0	0	0	X
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	1	X	0
1	1	0	1	1	0
1	1	1	0	0	1

Excitation Functions and Logic Diagram

Function and Logic Diagram for the conversion is given below

$S = JQ_N'$

KQ_N

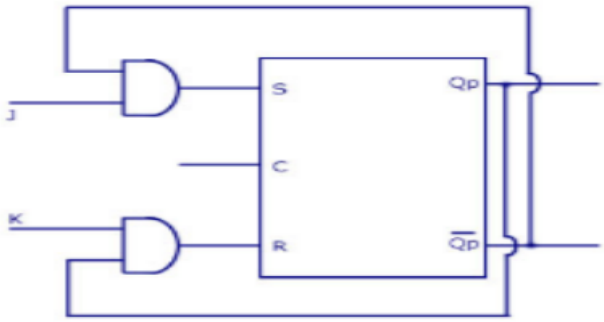
J

0	X	0	0
1	X	0	1

$R = KQ_N$

KQ_N

X	0	1	X
0	0	1	0



Convert SR To D Flip Flop

The Table for the SR To JK is given as

D	Q_N	Q_{N+1}	S	R
0	0	0	0	X
0	1	0	0	1
1	0	1	1	0
1	1	1	X	0

Excitation Functions and Logic Diagram

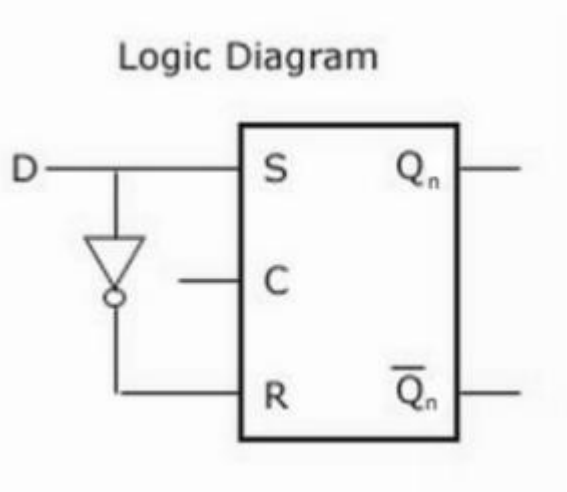
Function and Logic Diagram for the conversion is given below

S:

D,Q_N		
	0	0
	1	X

R:

D,Q_N		
	X	1
	0	0



Applications of Flip-Flops

These are the various types of flip-flops being used in digital electronic circuits and the applications of Flip-flops are as specified below.

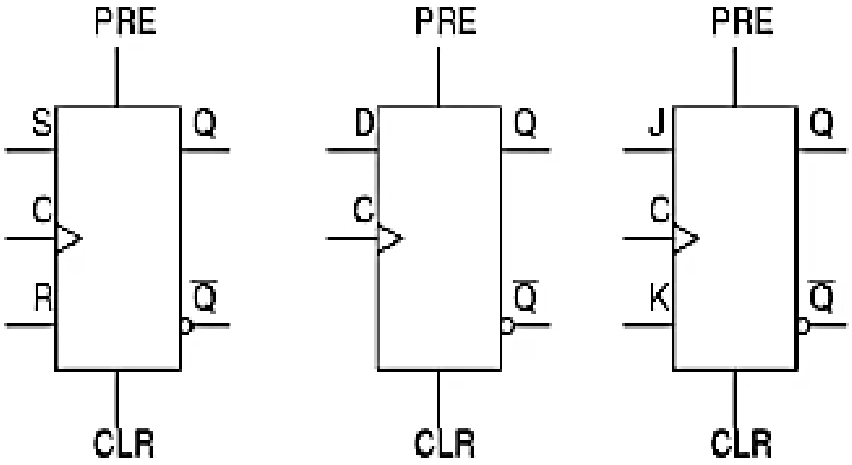
- **Counters:** The Flip Flop are used in the Counter Circuits for Counting pulse or events.
- **Frequency Dividers:** The Flip Flop are used in Frequency Dividers to divide the frequency of a input signal by a specific factor.
- **Shift Registers:** The Shift registers consist of interconnected flip-flops that shift data serially.
- **Storage Registers:** The Storage Resistor uses Flip Flop to store data in binary information.

- **Bounce elimination switch:** The Flip Flop are used in Bounce elimination switch to eliminate the contact bounce.
- **Data storage:** The Flip Flop are used in the Data Storage to store binary data temporarily or permanently.
- **Data transfer:** The Flip Flops are used for data transfer in different electronic parts.
- **Latch:** The Latches are the Sequential circuit which uses Flip Flop for temporary storage of data
- **Registers:** The Registers are mode from the array of flip flop which are used to store data temporarily**.**
- **Memory:** The Flip Flops are the main components in the memory unit for data storage.

FLIP-FLOPS WITH ASYNCHRONOUS INPUTS

Asynchronous flip flop inputs

The normal data inputs to a flip flop (D, S and R, or J and K) are referred to as *synchronous* inputs because they have effect on the outputs (Q and not-Q) only in step, or in sync, with the clock signal transitions. These extra inputs that I now bring to your attention are called *asynchronous* because they can set or reset the flip-flop regardless of the status of the clock signal. Typically, they're called *preset* and *clear*:



When the preset input is activated, the flip-flop will be set (Q=1, not-Q=0) regardless of any of the synchronous inputs or the clock. When the clear input is activated, the flip-flop will be reset (Q=0, not-Q=1), regardless of any of the synchronous inputs or the clock. So, what happens if both preset and clear inputs are activated? Surprise, surprise: we get an invalid state on the output, where Q and not-Q go to the same state, the same as our old friend, the S-R latch! Preset and clear inputs find use when multiple flip-flops are ganged together to perform a function on a multi-bit binary word, and a single line is needed to set or reset them all at once.

Asynchronous inputs, just like synchronous inputs, can be engineered to be active-high or active-low. If they're active-low, there will be an inverting bubble at that input lead on the block symbol, just like the negative edge-trigger clock inputs.

Viva Questions with Answers Based on Syllabus

UNIT I: Number Systems

1. **What are the different types of number systems?**
   Binary, Octal, Decimal, and Hexadecimal.
2. **How do you convert a binary number to decimal?**
   Multiply each bit by 2 raised to its position (starting from 0 on the right) and sum the results.
3. **Convert 10101_2 to decimal.**
   .
4. **How do you convert a decimal number to binary?**
   Repeatedly divide the decimal number by 2 and record the remainders. Read the remainders from bottom to top.
5. **What is r's complement?**
   For a base-r number system, r's complement is calculated by subtracting the number from , where n is the total number of digits.
6. **What is (r-1)'s complement?**
   Subtract each digit from . For binary, subtract each bit from 1.
7. **Find the 2's complement of 1101_2.**
   First, find the 1's complement (0010_2). Add 1 to get 2's complement: .
8. **What are signed binary numbers?**
   Binary numbers that include a sign bit: 0 for positive and 1 for negative.
9. **Perform .**
   .
10. **What is the difference between weighted and unweighted codes?**
    Weighted codes assign a specific weight to each digit position (e.g., BCD), whereas unweighted codes do not (e.g., Gray code).

UNIT II: Logic Gates and Boolean Algebra

11. **What are universal gates?**
    NAND and NOR gates because they can be used to implement any Boolean function.
12. **State De Morgan's theorems.**
    1. $\bar{}(A + B) = \bar{}A \bullet \bar{}B$
    2. $\bar{}(A \bullet B) = \bar{}A + \bar{}B$
13. **What is the truth table of an XOR gate?**

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

14. **How do you implement an OR gate using NAND gates?**
    Combine two NAND gates in series to create NOT gates, then feed the inverted inputs to a third NAND gate.

15. **What is a canonical form?**
    A Boolean function expressed as either a sum of minterms (SOP) or a product of maxterms (POS).

16. **What is the purpose of a K-map?**
    To simplify Boolean expressions and minimize logic circuits.

17. **What are don't care conditions?**
    Input combinations for which the output value does not affect the operation and can be used to simplify expressions.

UNIT III: Combinational Logic Circuits - 1

18. **What is the difference between a half-adder and a full-adder?**
    A half-adder adds two binary numbers, while a full-adder adds three inputs (including carry).

19. **Write the Boolean expressions for the sum and carry in a half-adder.**
    Sum: , Carry: .

20. **How do you implement a full-adder using half-adders?**
    Use two half-adders and an OR gate for the carry output.

21. **What is a ripple-carry adder?**
    A multi-bit adder where the carry bit ripples through successive adder stages.

22. **What are the limitations of a ripple-carry adder?**
    Delay due to carry propagation.

UNIT IV: Combinational Logic Circuits - 2

23. **What is the purpose of a decoder?**
    To convert binary information into a corresponding output line.

24. **What is a multiplexer?**
    A combinational circuit that selects one of several inputs to pass to the output.

25. **How do you realize a Boolean function using a multiplexer?**
    Use the function variables as selection lines and configure the inputs accordingly.

26. **What is a priority encoder?**
    An encoder that outputs the highest-priority input line.

27. **Explain how a demultiplexer differs from a decoder.**
    A demultiplexer routes a single input to one of several outputs, while a decoder activates one of several outputs based on the input combination.

UNIT V: Sequential Logic Circuits

28. **What is the difference between a latch and a flip-flop?**
    A latch is level-triggered, while a flip-flop is edge-triggered.

29. **What is the truth table for an RS latch using NOR gates?**

S	R	Q	Q'
0	0	Q	Q'
0	1	0	1
1	0	1	0
1	1	-	-

30. **What is a JK flip-flop?**
    A flip-flop that eliminates the invalid state of the RS flip-flop by toggling its output when both inputs are 1.

31. **Write the excitation table for a T flip-flop.**

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

32. **What is the function of preset and clear inputs in flip-flops?**
    Preset sets the output to 1, and Clear resets it to 0, regardless of the clock input.

LAB PROGRAMS

1. Realization of Basic Gates Using Universal Gates

Using NAND Gates:

1. **NOT Gate**:
    - Connect both inputs of a NAND gate together.
    - The output acts as a NOT gate.
    - Truth Table:

A	Output
0	1
1	0

2. **AND Gate**:
    - Use two NAND gates.
    - First NAND gate acts as a NOT gate for the output of a standard NAND gate.
    - Circuit:
        - Input A and B go to a NAND gate.
        - The output of the NAND gate goes into a second NAND gate with both inputs tied together.
3. **OR Gate**:
    - Use three NAND gates.
    - First, invert both inputs using two NAND gates (as NOT gates).
    - Then, use a third NAND gate to combine the inverted inputs.
4. **NOR Gate**:
    - Combine OR logic with NOT logic using NAND gates.
    - Use three NAND gates: two for the OR operation and one for inversion.

Using NOR Gates:

1. **NOT Gate**:
    - Connect both inputs of a NOR gate together.
    - The output acts as a NOT gate.
    - Truth Table:

A	Output
0	1
1	0

2. **OR Gate**:
    - Use a single NOR gate.
    - The NOR gate directly performs the OR function with inverted output.
3. **AND Gate**:
    - Use three NOR gates.
    - First, invert both inputs using two NOR gates (as NOT gates).
    - Then, use a third NOR gate to combine the inverted inputs.
4. **NAND Gate**:
    - Combine AND logic with NOT logic using NOR gates.
    - Use three NOR gates: two for the AND operation and one for inversion.

2.Design and implementation of half and full adder circuits using logic gates

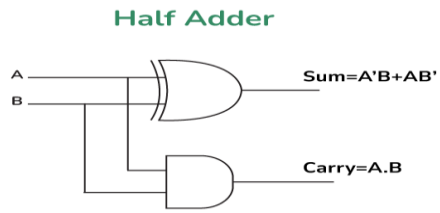
Full Adder:

1. **Definition:**
    - A combinational circuit that adds three binary inputs: A, B, and Carry-in (Cin) and produces a sum (S) and a carry-out (Cout) output.
2. **Boolean Expressions:**
    - Sum (S) = A ⊕ B ⊕ Cin
    - Carry-out (Cout) = (A • B) + (Cin • (A ⊕ B))
3. **Logic Diagram:**
    - Use two XOR gates for the Sum output.
    - Use two AND gates and one OR gate for the Carry-out output.
4. **Truth Table:**

A	B	Cin	Sum (S)	Carry-out (Cout)
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

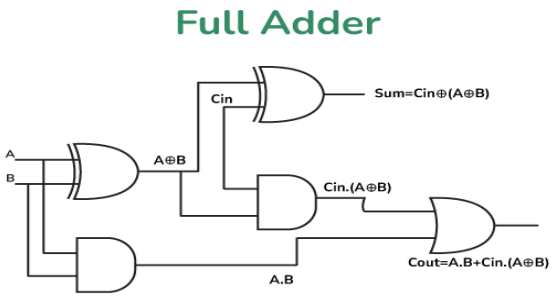
Implementation Notes:

- Half adder circuits are simpler and require fewer gates compared to full adders.
- Full adder circuits can be cascaded to add multi-bit binary numbers.
- These circuits form the fundamental building blocks for more complex arithmetic circuits like ripple-carry adders.



Truth Table

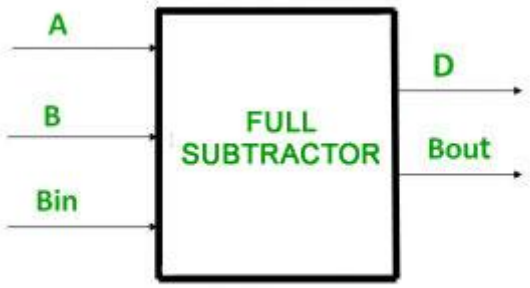
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Inputs			Outputs	
A	B	C	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

3.Design and implementation of half and full subtractor circuits using logic gates

A full subtractor is a **combinational circuit** that performs subtraction of two bits, one is minuend and other is subtrahend, taking into account borrow of the previous adjacent lower minuend bit. This circuit **has three inputs and two outputs**. The three inputs A, B and Bin, denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D and Bout represent the difference and output borrow, respectively. Although subtraction is usually achieved by adding the complement of subtrahend to the minuend, it is of academic interest to work out the Truth Table and logic realisation of a full subtractor; x is the minuend; y is the subtrahend; z is the input borrow; D is the difference; and B denotes the output borrow. The corresponding maps for logic functions for outputs of the full subtractor namely difference and borrow.



Here's how a full subtractor works:

1. First, we need to convert the binary numbers to their two's complement form if we are subtracting a negative number.
2. Next, we compare the bits in the minuend and subtrahend at the corresponding positions. If the subtrahend bit is greater than or equal to the minuend bit, we need to borrow from the previous stage (if there is one) to subtract the subtrahend bit from th**e** minuend bit.
3. We subtract the two bits along with the borrow-in to get the difference bit. If the minuend bit is greater than or equal to the subtrahend bit along with the borrow-in, then the difference bit is 1, otherwise it is 0.
4. We then calculate the borrow-out bit by comparing the minuend and subtrahend bits. If the minuend bit is less than the subtrahend bit along with the borrow-in, then we need to borrow for the next stage, so the borrow-out bit is 1, otherwise it is 0.

The circuit diagram for a full subtractor usually consists of two half-subtractors and an additional OR gate to calculate the borrow-out bit. The inputs and outputs of the full subtractor are as follows:

Inputs:

A: minuend bit
B: subtrahend bit
Bin: borrow-in bit from the previous stage
Outputs:

Diff: difference bit
Bout: borrow-out bit for the next stage

Truth Table –

INPUT			OUTPUT	
A	B	Bin	D	Bout
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

4.Verification of stable tables of RS, JK, T and D flip-flops using NAND and NOR gates

Introduction

A flip flop is an electronic 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.

1. R-S flip flop
2. D flip flop
3. J-K flip flop
4. T flip flop

1) RS flip flop

The basic NAND gate RS flip flop circuit is used to store the data and thus provides feedback from both of its outputs again back to its inputs. The RS flip flop actually has three inputs, SET, RESET and clock pulse.

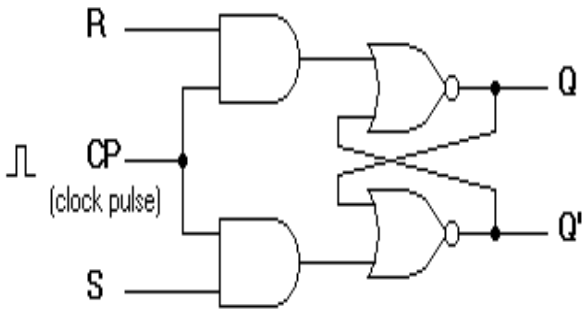


Figure-1:R-S flip flop circuit diagram

INPUTS			OUTPUT	STATE
CLK	S	R	Q	
X	0	0	No Change	Previous
↑	0	1	0	Reset
↑	1	0	1	Set
↑	1	1	-	Forbidden

Figure-2:Characteristics table of R-S flip flop

2) D flip flop

A D flip flop has a single data input. This type of flip flop is obtained from the SR flip flop by connecting the R input through an inverter, and the S input is connected directly to data input. The modified clocked SR flip-flop is known as D-flip-flop and is shown below. From the truth table of SR flip-flop we see that the output of the SR flip-flop is in unpredictable state when the inputs are same and high. In many practical applications, these input conditions are not required. These input conditions can be avoided by making them complement of each other.

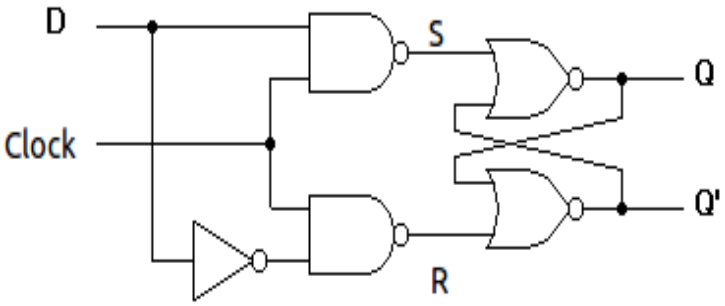


Figure-3:Circuit diagram of D flip flop

Input			Output	
D	reset	clock	Q	Q'
0	0	0	0	1
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

Figure-4:Characteristics table of D flip flop

3) J-K flip flop

In a RS flip-flop the input R=S=1 leads to an indeterminate output. The RS flip-flop circuit may be re-joined if both inputs are 1 than also the outputs are complement of each other as shown in characteristics table below.

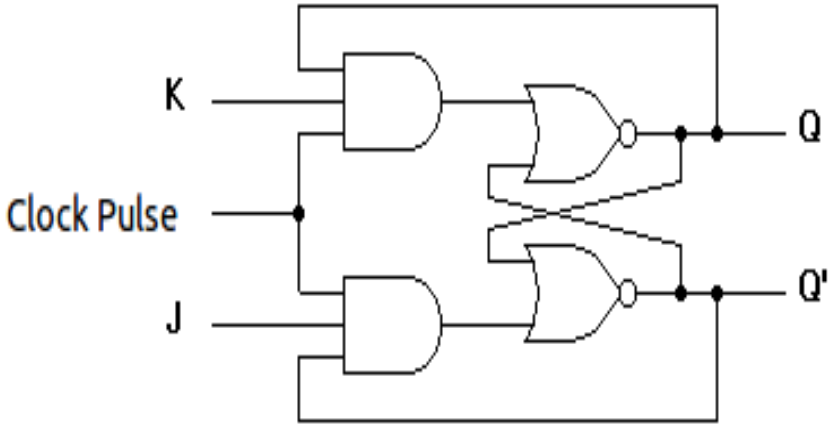


Figure-5:Circuit diagram of J-K flip flop

Trigger	Inputs		Output				Inference
			Present State		Next State		
CLK	J	K	Q	Q'	Q	Q'	
⊠	x	x	-		-		Latched
⊓	0	0	0	1	0	1	No Change
⊓			1	0	1	0	
⊓	0	1	0	1	0	1	Reset
⊓			1	0	0	1	
⊓	1	0	0	1	1	0	Set
⊓			1	0	1	0	
⊓	1	1	0	1	1	0	Toggles
⊓			1	0	0	1	

Figure-6:Characteristics table of J-K flip flop

4) T flip flop

T flip-flop is known as toggle flip-flop. The T flip-flop is modification of the J-K flip-flop. Both the JK inputs of the JK flip – flop are held at logic 1 and the clock signal continuous to change as shown in table below.

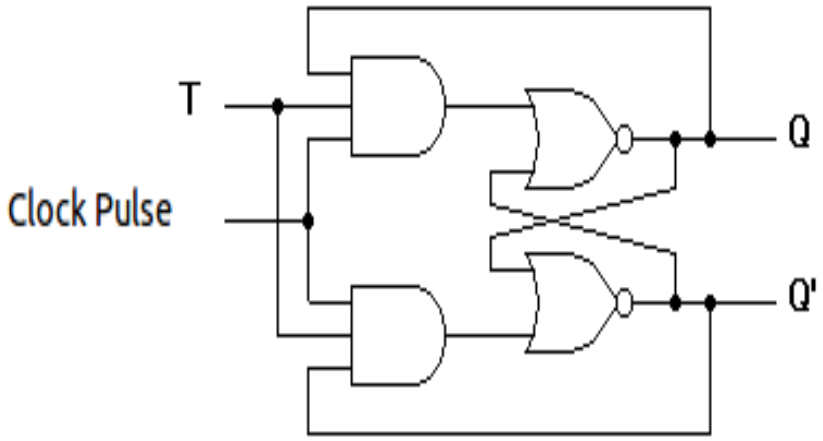


Figure-7:Circuit diagram of T flip flop

T flip-flop

T	Clock	Q	Q'
0	↑	Q	Q'
1	↑	Q'	Q
X	↓	Q	Q'

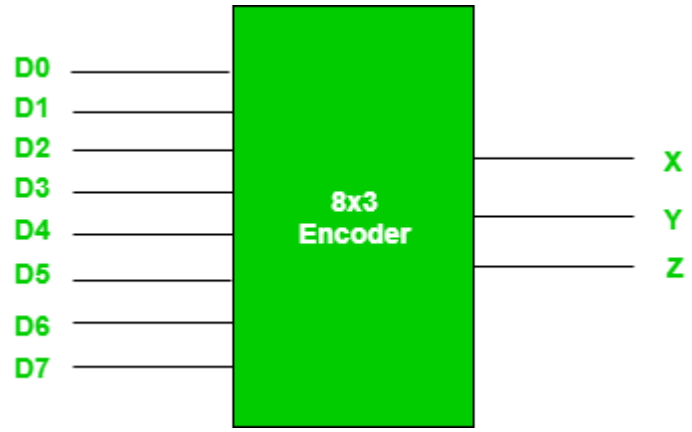
Figure-8:Characteristics table of T flip flop

5.Implementation and verification of Decoder and encoder using logic gates.

Binary code of N digits can be used to store 2N distinct elements of coded information. This is what encoders and decoders are used for.

Encoders convert 2N lines of input into a code of N bits and **Decoders** decode the N bits into 2N lines.

1. Encoders – An encoder is a combinational circuit that converts binary information in the form of a 2N input lines into N output lines, which represent N bit code for the input. For simple encoders, it is assumed that only one input line is active at a time. As an example, let's consider **Octal to Binary** encoder. As shown in the following figure, an octal-to-binary encoder takes 8 input lines and generates 3 output lines.



Truth Table –

D7	D6	D5	D4	D3	D2	D1	D0	X	Y	Z
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1

D7	D6	D5	D4	D3	D2	D1	D0	X	Y	Z
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

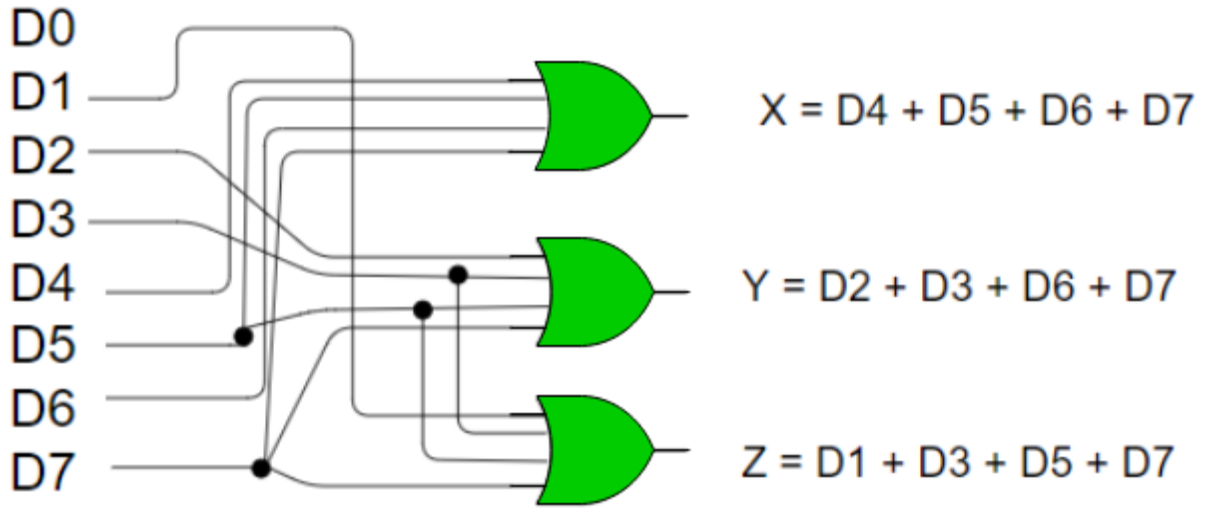
As seen from the truth table, the output is 000 when D0 is active; 001 when D1 is active; 010 when D2 is active and so on.

Implementation –

From the truth table, the output line Z is active when the input octal digit is 1, 3, 5 or 7. Similarly, Y is 1 when input octal digit is 2, 3, 6 or 7 and X is 1 for input octal digits 4, 5, 6 or 7. Hence, the Boolean functions would be:

X = D4 + D5 + D6 + D7
Y = D2 +D3 + D6 + D7
Z = D1 + D3 + D5 + D7

Hence, the encoder can be realised with OR gates as follows:



One limitation of this encoder is that only one input can be active at any given time. If more than one inputs are active, then the output is undefined. For example, if D6 and D3 are both active, then, our output would be 111 which is the output for D7. To overcome this, we use Priority Encoders. Another ambiguity arises when all inputs are 0. In this case, encoder outputs 000 which actually is the output for D0 active. In order to avoid this, an extra bit can be added to the output, called the valid bit which is 0 when all inputs are 0 and 1 otherwise.

Priority Encoder –

A priority encoder is an encoder circuit in which inputs are given priorities. When more than one inputs are active at the same time, the input with higher priority takes precedence and the output corresponding to that is generated. Let us consider the 4 to 2 priority encoder as an example. From the truth table, we see that when all inputs are 0, our V bit or the valid bit is zero and outputs are not used. The x's in the table show the don't care condition, i.e, it may either be 0 or 1. Here, D3 has highest priority, therefore, whatever be the other inputs, when D3 is high, output has to be 11. And D0 has the lowest priority, therefore the output would be 00 only when D0 is high and the other input lines are low. Similarly, D2 has higher priority over D1 and D0 but lower than D3 therefore the output would be 010 only when D2 is high and D3 are low (D0 & D1 are don't care).

Truth Table –

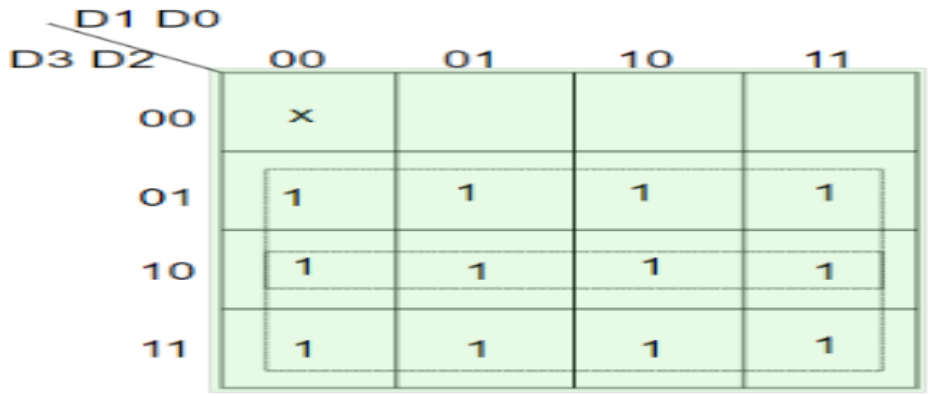
D3	D2	D1	D0	X	Y	V
0	0	0	0	x	x	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Implementation –

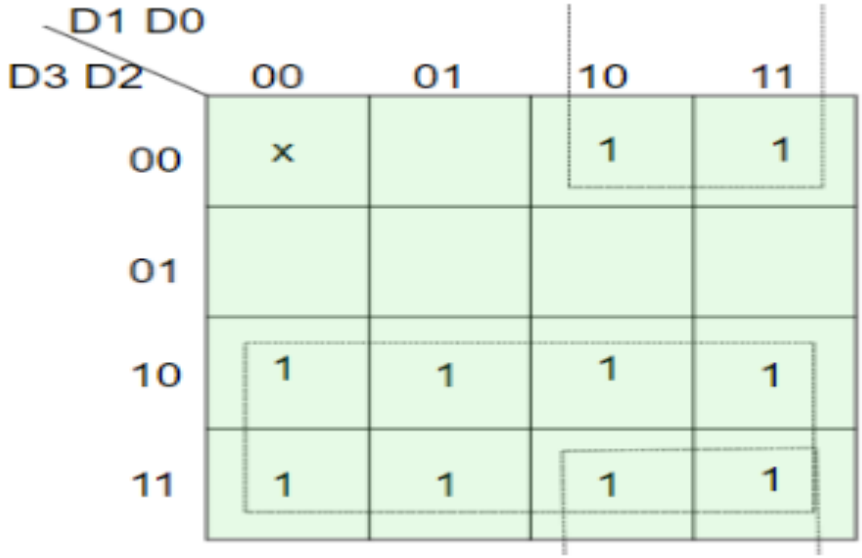
It can clearly be seen that the condition for valid bit to be 1 is that at least any one of the inputs should be high. Hence,

V = D0 + D1 + D2 + D3

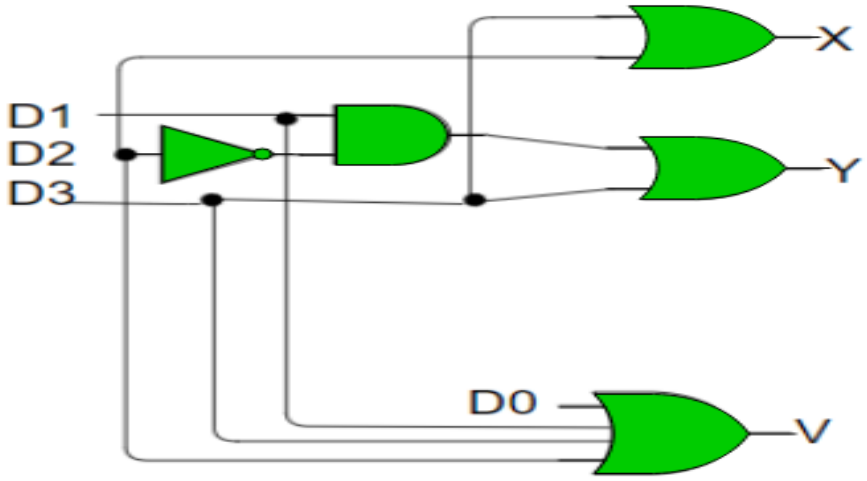
For X:



=> X = D2 + D3 For Y:



=> Y = D1 D2’ + D3 Hence, the priority 4-to-2 encoder can be realized as follows:



2. Decoders –

A decoder does the opposite job of an encoder. It is a combinational circuit that converts n lines of input into 2

n

lines of output. Let’s take an example of 3-to-8 line decoder.

Truth Table –

X	Y	Z	D0	D1	D2	D3	D4	D5	D6	D7
0	0	0	1	0	0	0	0	0	0	0

X	Y	Z	D0	D1	D2	D3	D4	D5	D6	D7
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Implementation –

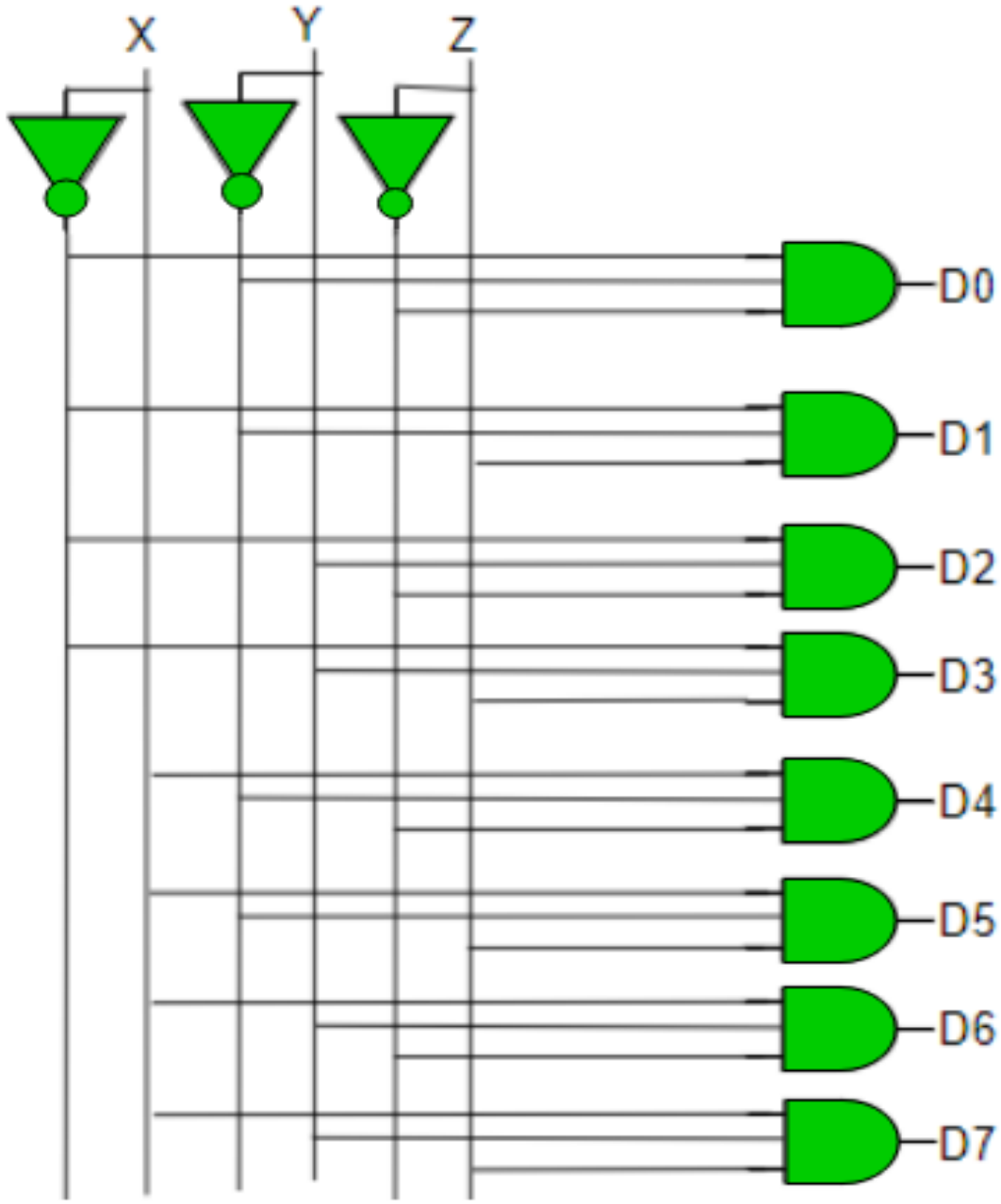
D0 is high when X = 0, Y = 0 and Z = 0. Hence,

D0 = X’ Y’ Z’

Similarly,

D1 = X’ Y’ Z
D2 = X’ Y Z’
D3 = X’ Y Z
D4 = X Y’ Z’
D5 = X Y’ Z
D6 = X Y Z’
D7 = X Y Z

Hence,



6.Implementation of 4X1 MUX and DeMUX using logic gates.

Introduction

The function of a multiplexer is to select the input of any 'n' input lines and feed that to one output line. The function of a de-multiplexer is to inverse the function of the multiplexer and the shortcut forms of the multiplexer. The de-multiplexers are mux and demux. Some multiplexers perform both multiplexing and de-multiplexing operations.

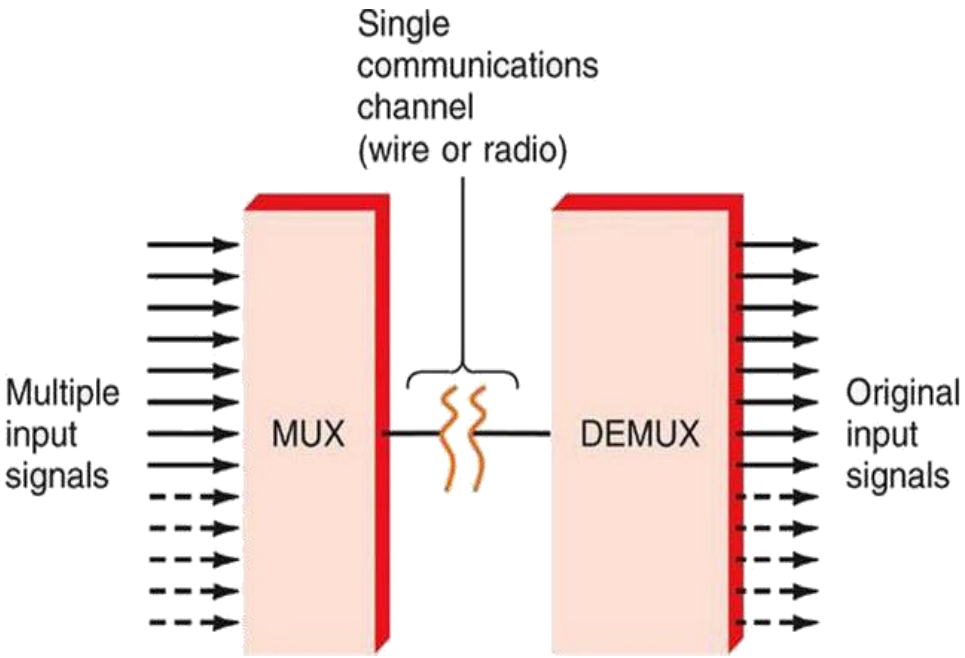


Figure-1:Block diagram of Multiplexer and De-multiplexer

1) Multiplexer

Multiplexer is a device that has multiple inputs and a single line output. The select lines determine which input is connected to the output, and also to increase the amount of data that can be sent over a network within certain time. It is also called a data selector.

Multiplexers are classified into four types:

a) 2-1 multiplexer (1 select line)
b) 4-1 multiplexer (2 select lines)
c) 8-1 multiplexer(3 select lines)
d) 16-1 multiplexer (4 select lines)

1.1) 4x1 Multiplexer

4x1 Multiplexer has four data inputs D0, D1, D2 & D3, two selection lines S0 & S1 and one output Y. The block diagram of 4x1 Multiplexer is shown in the following figure.One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. Truth table of 4x1 Multiplexer is shown below.

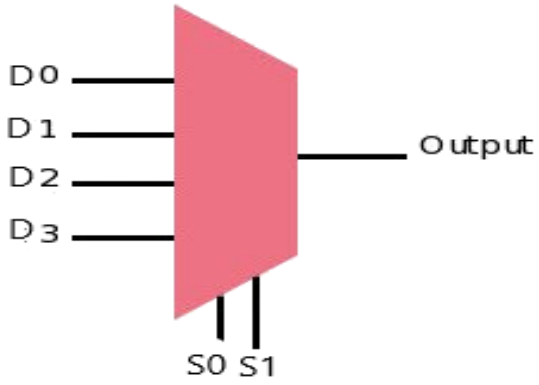


Figure-2:Block diagram of 4x1 Multiplexer

Selection Lines		Output
S0	**S1**	**Y**
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Figure-3:Truth table of 4x1 Multiplexer

2) De-multiplexer

De-multiplexer is also a device with one input and multiple output lines. It is used to send a signal to one of the many devices. The main difference between a multiplexer and a de-multiplexer is that a multiplexer takes two or more signals and encodes them on a wire, whereas a de-multiplexer does reverse to what the multiplexer does.

De-multiplexer are classified into four types:

a)1-2 demultiplexer (1 select line)
b)1-4 demultiplexer (2 select lines)
c)1-8 demultiplexer (3 select lines)
d)1-16 demultiplexer (4 select lines)

2.2) 1x4 De-multiplexer

1x4 De-Multiplexer has one input Data(D), two selection lines, S0 & S1 and four outputs Y0, Y1, Y2 & Y3. The block diagram of 1x4 De-Multiplexer is shown in the following figure.

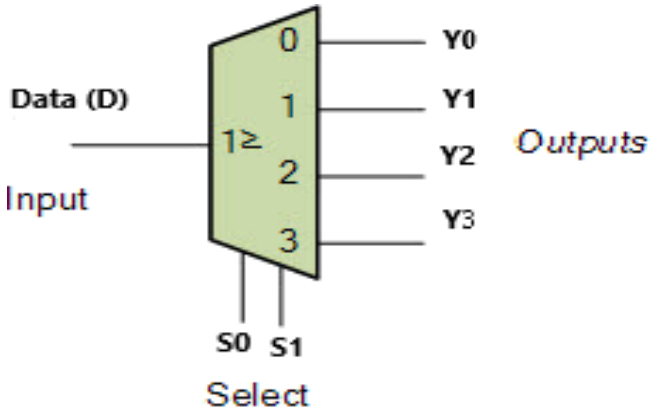


Figure-4:Block diagram of 1x4 De-Multiplexer

Selection Inputs		Outputs			
S0	S1	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

Figure-5:Truth table of 1x4 De-Multiplexer

7.Implementation of 7-segment decoder circuit.

Binary Coded Decimal (BCD)

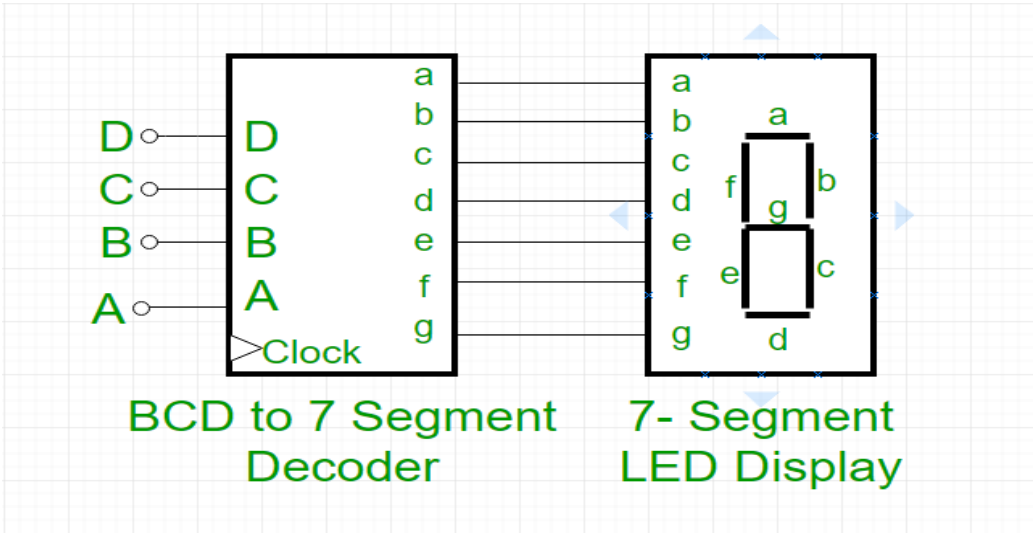
BCD is the encoding scheme each of the decimal numbers(0-9) is represented by its equivalent binary pattern(which is generally of 4-bits).

Seven segment

Seven Segment display is an electronic device which consists of seven Light Emitting Diodes (LEDs) arranged in a some definite pattern (common cathode or common anode type), which is used to display Hexadecimal numerals(in this case decimal numbers, as input is BCD i.e., 0-9). Two types of seven segment LED display:

1. **Common Cathode Type:** In this type of display all cathodes of the seven LEDs are connected together to the ground or -Vcc(hence, common cathode) and LED displays digits when some 'HIGH' signal is supplied to the individual anodes.
2. **Common Anode Type:** In this type of display all the anodes of the seven LEDs are connected to battery or +Vcc and LED displays digits when some 'LOW' signal is supplied to the individual cathodes.

But, seven segment display does not work by directly supplying voltage to different segments of LEDs. First, our decimal number is changed to its BCD equivalent signal then BCD to seven segment decoder converts that signals to the form which is fed to seven segment display. This BCD to seven segment decoder has four input lines (A, B, C and D) and 7 output lines (a, b, c, d, e, f and g), this output is given to seven segment LED display which displays the decimal number depending upon inputs.



Truth Table

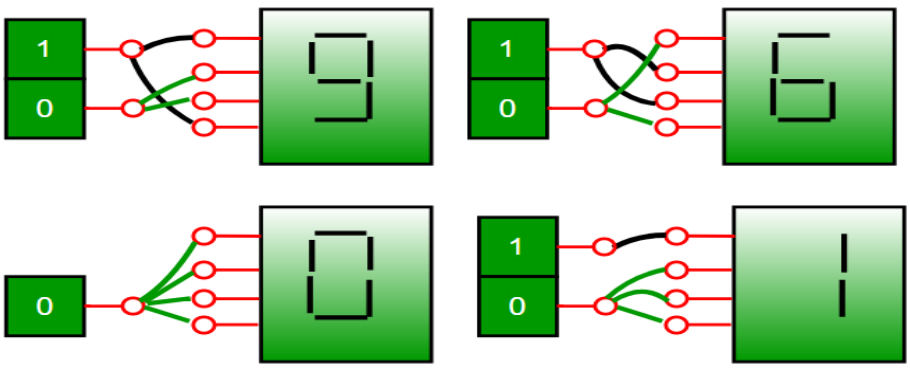
For common cathode type BCD to seven segment decoder:

A	B	C	D	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

Note –

- For Common Anode type seven segment LED display, we only have to interchange all ‘0s’ and ‘1s’ in the output side i.e., (for a, b, c, d, e, f, and g replace all ‘1’ by ‘0’ and vice versa) and solve using K-map.
- Output for first combination of inputs (A, B, C and D) in Truth Table corresponds to ‘0’ and last combination corresponds to ‘9’. Similarly rest corresponds from 2 to 8 from top to bottom.
- BCD numbers only range from 0 to 9,thus rest inputs from 10-F are invalid inputs.

Example –

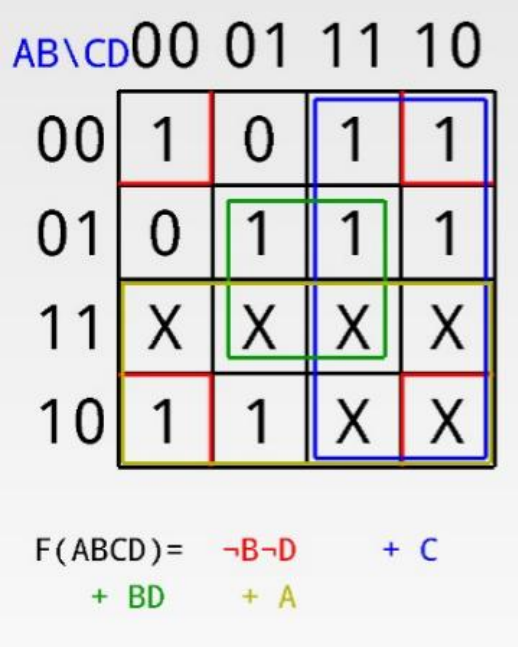


Explanation –

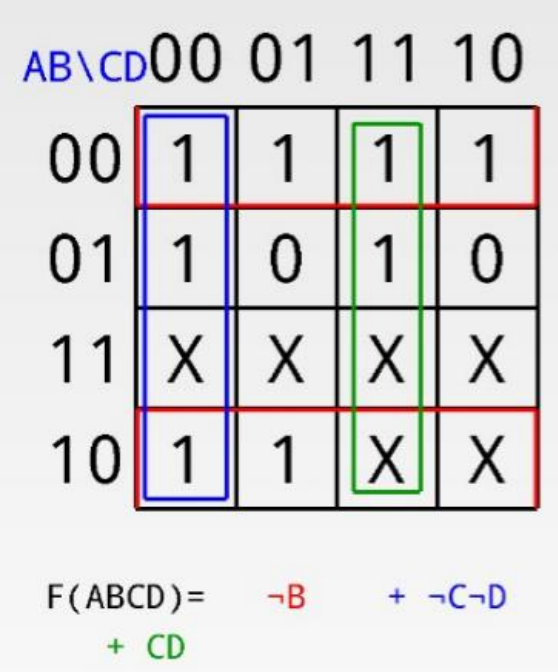
For combination where all the inputs (A, B, C and D) are zero (see Truth Table), our output lines are a = 1, b = 1, c = 1, d = 1, e = 1, f = 1 and g = 0. So 7 segment display shows ‘zero’ as output. Similarly, for combination where one of the input is one (D = 1) and rest are zero, our output lines are a = 0, b = 1, c = 1, d = 0, e = 0, f = 0 and g = 0. So only LEDs ‘b’ and ‘c’ (see diagram above) will glow and 7 segment display shows ‘one’ as output.

K-Maps:

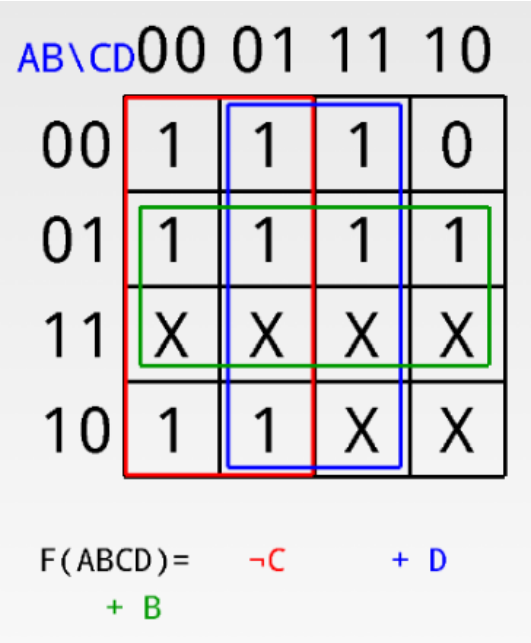
#for a:



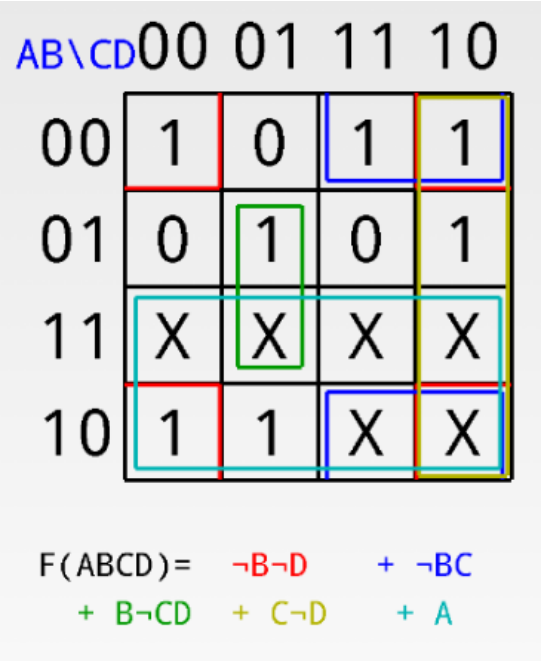
#for b:



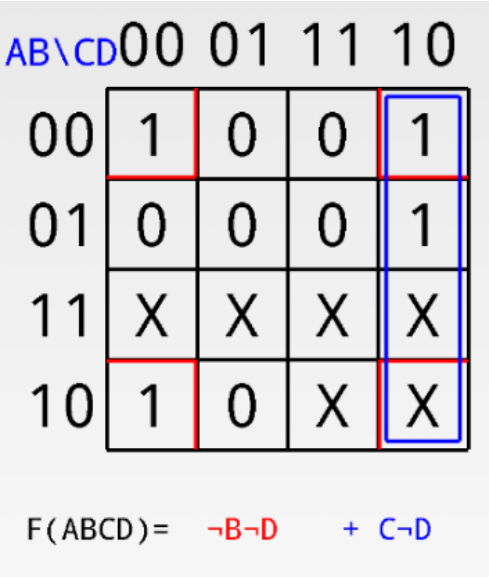
#for c:


AB\CD 00 01 11 10
00 1 1 1 0
01 1 1 1 1
11 X X X X
10 1 1 X X
F(ABCD)= ¬C + D
+ B

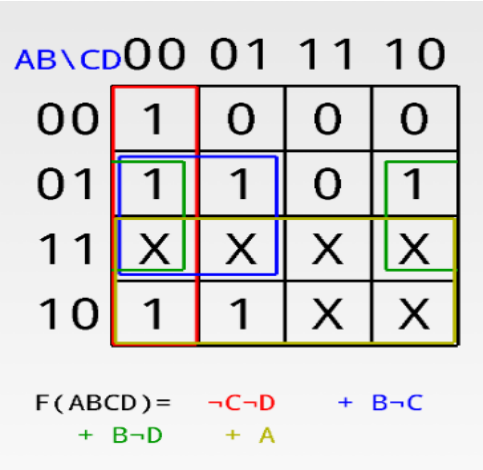
#for d:


AB\CD 00 01 11 10
00 1 0 1 1
01 0 1 0 1
11 X X X X
10 1 1 X X
F(ABCD)= ¬B¬D + ¬BC
+ B¬CD + C¬D + A

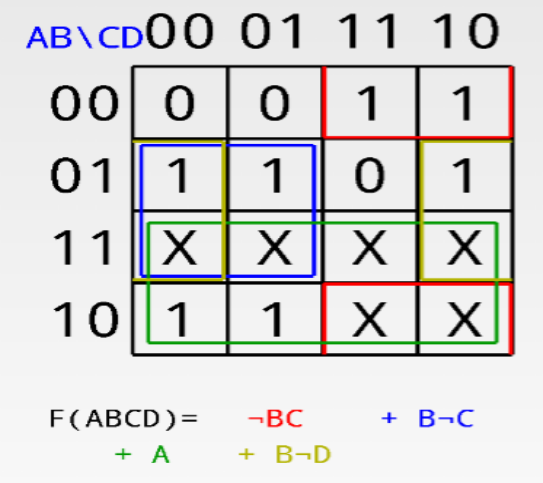
#for e:



#for f:



#for g:



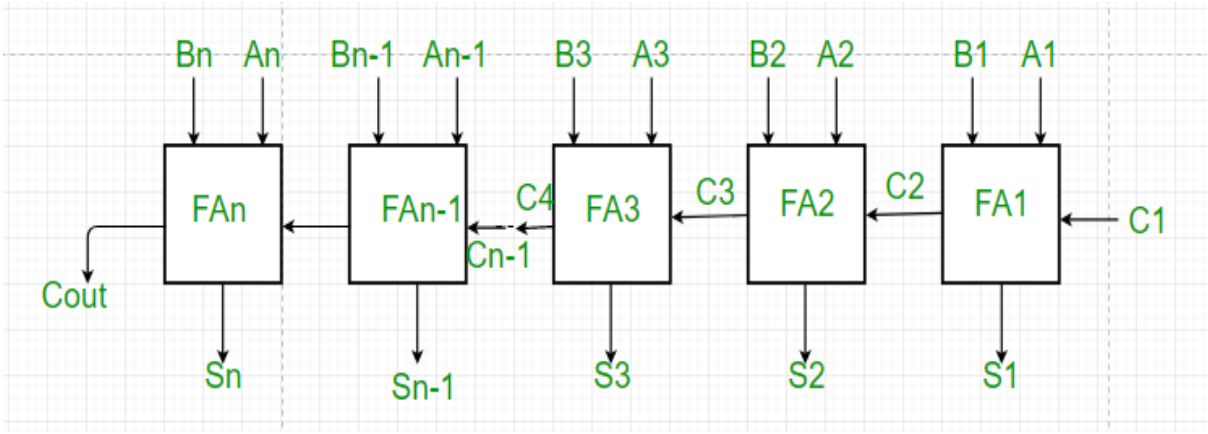
8.Implementation of 4-bit parallel adder

Parallel Adder

A single full adder performs the addition of two one bit numbers and an input carry. But a **Parallel Adder** is a digital circuit capable of finding the arithmetic **sum** of two binary numbers that is **greater than one bit** in length by operating on corresponding pairs of bits in parallel. It consists of **full adders connected in a chain** where the output carry from each full adder is connected to the carry input of the next higher order full adder in the chain.

A n bit parallel adder requires n full adders to perform the operation.

So for the two-bit number, two adders are needed while for four bit number, four adders are needed and so on. Parallel adders normally incorporate carry lookahead logic to ensure that carry propagation between subsequent stages of addition does not limit addition speed.

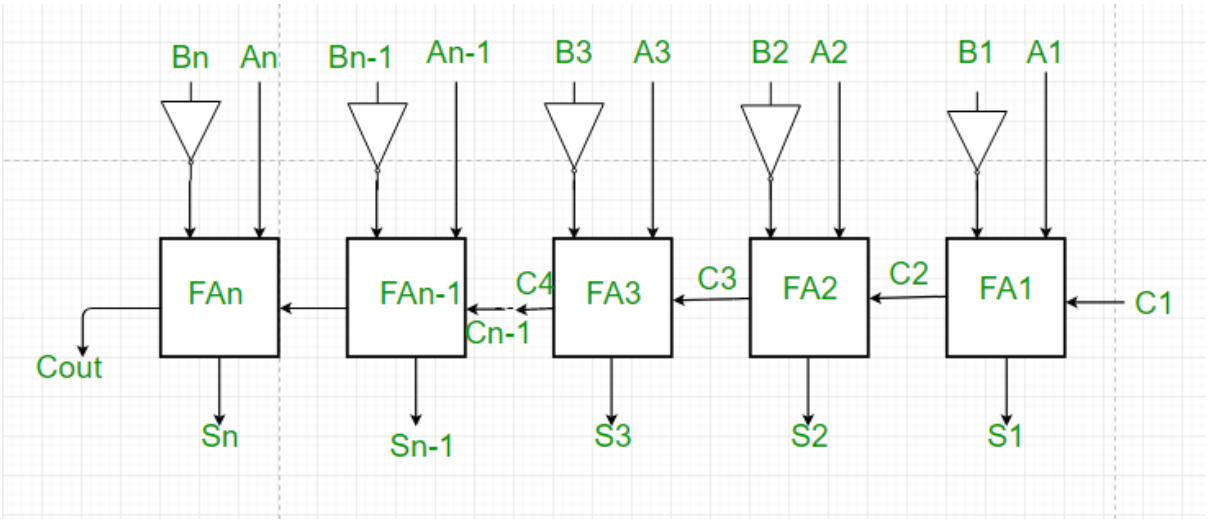


Working of Parallel Adder

1. As you can show in the figure, first of all the full adder FA1 add A1 and B1 along with the carry C1 to generate the sum S1 (the first bit of the output sum) and the carry C2 which is connected to the next adder in chain.
2. Next, the full adder FA2 uses this carry bit C2 to add with the input bits A2 and B2 to generate the sum S2(the second bit of the output sum) and the carry C3 which is again further connected to the next adder in chain and so on.
3. The process continues till the last full adder FAn uses the carry bit Cn to add with its input An and Bn to generate the last bit of the output along last carry bit Cout.

Parallel Subtractor

A Parallel Subtractor is a digital circuit capable of finding the arithmetic difference of two binary numbers that is more than one bit in length by operating on pairs of bits in parallel. The parallel subtractor can be designed in several ways including combination of half and full subtractors, all full subtractors or all full adders with the complement of the number being subtracted input.



Working of Parallel Subtractor

1. As shown in the figure, the parallel binary subtractor is formed by combination of all full adders with subtrahend complement input.
2. This operation considers that the addition of minuend along with the 2's complement of the subtrahend is equal to their subtraction.
3. Firstly the 1's complement of B is obtained by the NOT gate and 1 can be added through the carry to find out the 2's complement of B. This is further added to A to carry out the arithmetic subtraction.
4. The process continues till the last full adder FAn uses the carry bit Cn to add with its input An and 2's complement of Bn to generate the last bit of the output along last carry bit Cout.

9.Design and verification of 4-bit synchronous and asynchronous counter.

Introduction

A counter is a device which stores (and sometimes displays) the number of times a particular event or process has occurred, often in relationship to a clock signal. Counters are used in digital electronics for counting purpose, they can count specific event happening in the circuit. For example, in UP counter a counter increases count for every rising edge of clock.

Classification of Counters

Counters are broadly divided into two categories

1. Asynchronous counter
2. Synchronous counter

1) Asynchronous Counter

In asynchronous counter we don't use universal clock, only first flip flop is driven by main clock and the clock input of rest of the following counters is driven by output of previous flip flops. We can understand it by following diagram-

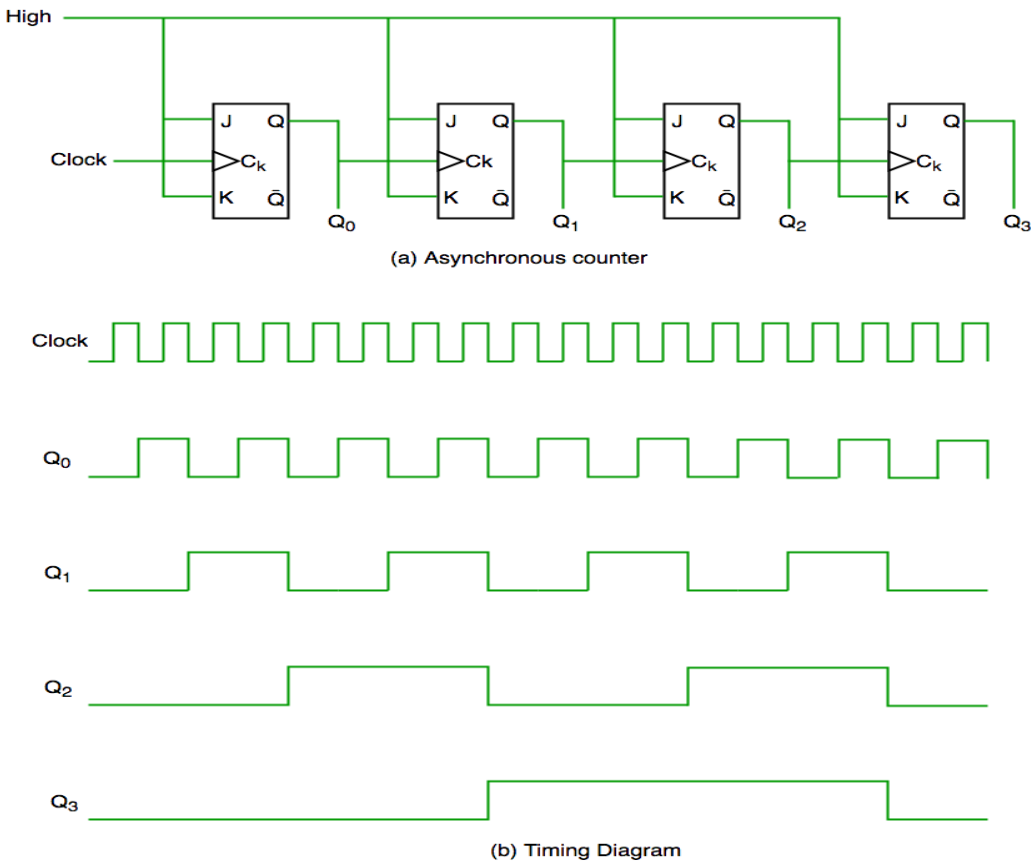
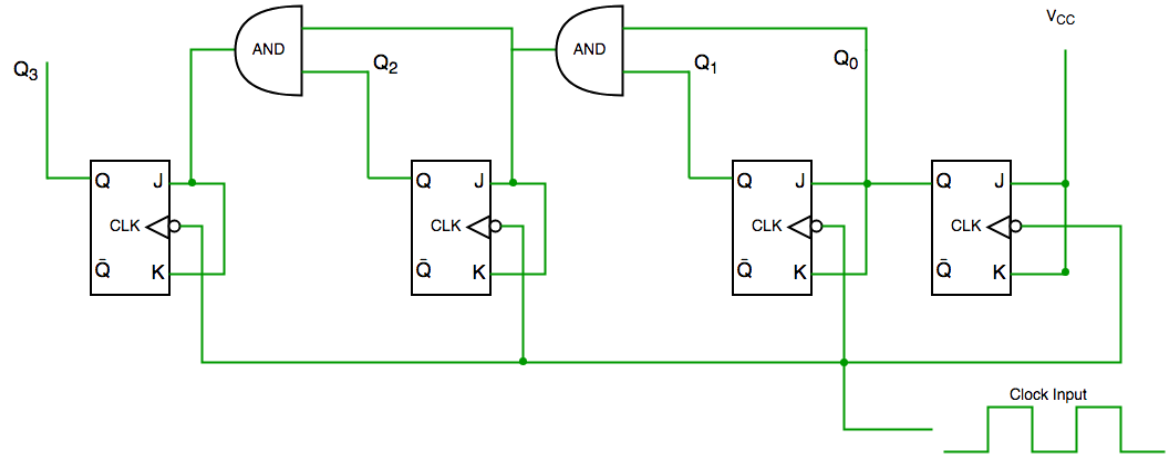


Figure-1: Asynchronous Counter Circuit and Timing Diagram

It is evident from timing diagram that Q0 is changing as soon as the rising edge of clock pulse is encountered, Q1 is changing when rising edge of Q0 is encountered(because Q0 is like clock pulse for second flip flop) and so on. In this way ripples are generated through Q0,Q1,Q2,Q3 hence it is also called RIPPLE counter.

2) Synchronous Counter

Unlike the asynchronous counter, synchronous counter has one global clock which drives each flip flop so output changes in parallel. The one advantage of synchronous counter over asynchronous counter is, it can operate on higher frequency than asynchronous counter as it does not have cumulative delay because of same clock is given to each flip flop.



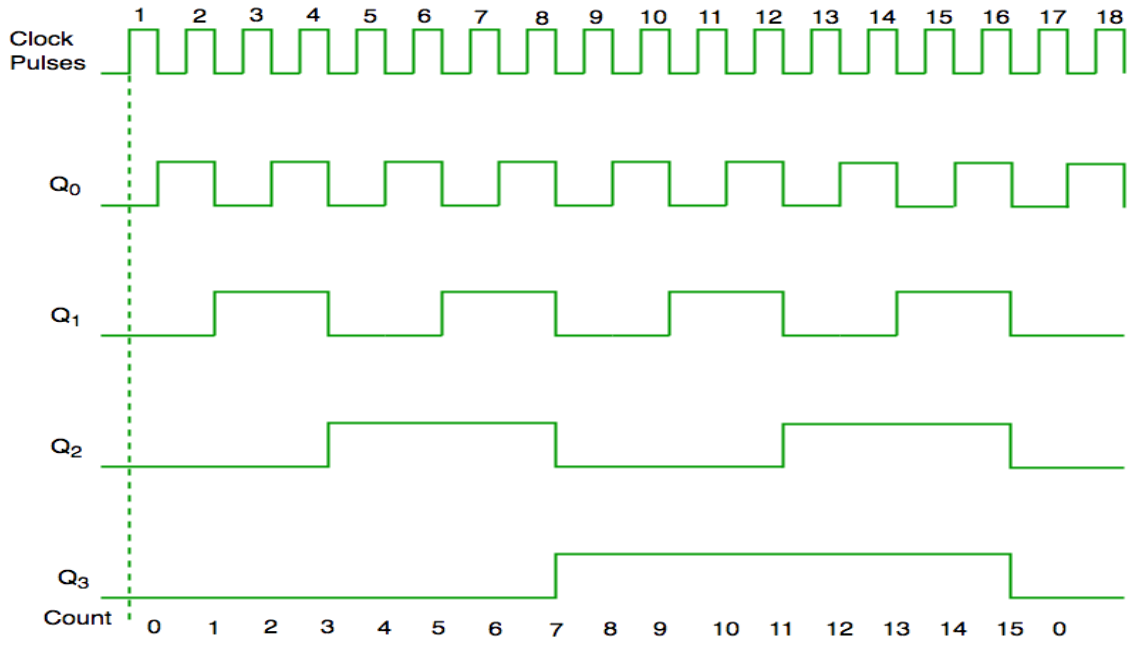


Figure-2: Synchronous Counter Circuit and Timing Diagram

From circuit diagram we see that Q0 bit gives response to each falling edge of clock while Q1 is dependent on Q0, Q2 is dependent on Q1 and Q0 , Q3 is dependent on Q2,Q1 and Q0.