Since 1951

# JAMAL MOHAMED COLLEGE (AUTONOMOUS)

**Accredited (3rd Cycle) with 'A' Grade by NAAC**

**Affiliated to Bharathidasan University**

**TIRUCHIRAPPALLI – 620 020**

## DEPARTMENT OF COMPUTER SCIENCE

# COMPUTER ORGANIZATION AND ARCHITECTURE

**20UCS5CC11**

**Prepared By: Dr. T. Abdul Razak**

| Semester | Code | Course | Title of the Course | Hours | Credits | Max. Marks | Internal Marks | External Marks |
|---|---|---|---|---|---|---|---|---|
| **V** | **20UCS5CC11** | **Core – XI** | **COMPUTER ORGANIZATION AND ARCHITECTURE** | **5** | **5** | **100** | **25** | **75** |

**Course Outcomes (COs):**

**On completion of the course, students will be able to**

CO1. Understand the various types of number systems and the usage of binary codes.
CO2. Apply Boolean laws and theorems to simplify and implement Boolean expressions.
CO3. Design and analyse combinational circuits.
CO4. Design and analyse sequential circuits.
CO5. Understand the architecture and functionality of a central processing unit.

**UNIT I** **15 hours**

Number Systems – Decimal, Binary, Octal and Hexadecimal Systems – Conversion from one system to another – Addition, Subtraction, Multiplication and Division of Binary, Octal and Hexadecimal Numbers – Binary Codes – 8421, 2421, Excess-3, Gray – Weighted and Non-weighted codes, Reflected Code, Self-complementary Codes – BCD Codes – Alphanumeric Codes.

**UNIT II** **15 hours**

Basic Logic Gates – Universal Logic – Boolean Laws and Theorems – Boolean Expressions – Sum of Products – Product of Sums – Simplification of Boolean Expressions – Karnaugh Map Method (up to 4 Variables) – Implementation of Boolean Expressions using Gate Networks.

**UNIT III** **15 hours**

Combinational Circuits – Multiplexers – Demultiplexers – Decoders – Encoders – Arithmetic Building Blocks – Half and Full Adders – Half and Full Subtractors – Parallel adder – 2's Complement Adder-Subtractor.

**UNIT IV** **15 hours**

Sequential Circuits – Flip Flops – RS, Clocked RS, D, JK, T and Master-Slave Flip Flops – Shift Register – Counters – Asynchronous and Synchronous counters – Mod *n* Counter – Ring Counter**.**

**UNIT V** **15 hours**

Register Transfer and Microoperations: Register Transfer Language – Register Transfer – Arithmetic Microoperations – Logic Microoperations – Arithmetic Logic Unit – Central Processing Unit: General Register Organization – Stack Organization – Instruction Formats – Addressing Modes – Data Transfer and Manipulation.

**Text Books:**

1. Albert Paul Malvino, Donald P. Leach and Goutam Saha, *Digital Principles and Applications*, TMH, Sixth Edition, 2007.
2. Morris Mano M, *Computer System Architecture*, PHI, Third Edition, 2008.

# UNIT-1

## NUMBER SYSTEMS

The technique to represent and work with numbers is called number system.

**Types of Number Systems:**

1. Decimal system (0 – 9)
2. Binary system (0, 1)
3. Octal system (0 – 7)
4. Hexadecimal system (0 – 9, A – F)

**Base or radix of a number system:**

It is defined as the number of digits available in a number system.

**Bases of the different number systems**

| Number System | Available Digits | Base / Radix |
|---|---|---|
| Decimal | 0 – 9 | 10 |
| Binary | 0 & 1 | 2 |
| Octal | 0 – 7 | 8 |
| Hexadecimal | 0 – 9, A – F | 16 |

**Binary Number System**

The number system having just these two digits 0 and 1 is called a binary number system. Each binary digit is also called a bit. The base / radix of the binary system is 2. Binary number system is a positional value system, where each digit has a value expressed in powers of 2, as shown below:

| | | | | | |
|---|---|---|---|---|---|
| $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |

**Octal Number System**

Octal number system has eight digits – 0, 1, 2, 3, 4, 5, 6 and 7. The base / radix of the octal system is 8. Octal number system is also a positional value system where each digit has its value expressed in powers of 8, as shown below:

| | | | | | |
|---|---|---|---|---|---|
| $8^5$ | $8^4$ | $8^3$ | $8^2$ | $8^1$ | $8^0$ |

**Hexadecimal Number System**

Hexadecimal number system has 16 digits – 0 to 9, A to F. The base / radix of the hexadecimal system is 16. In hexadecimal number system, each digit has its value expressed in powers of 16, as shown below:

| | | | | | |
|---|---|---|---|---|---|
| $16^5$ | $16^4$ | $16^3$ | $16^2$ | $16^1$ | $16^0$ |

## CONVERSION FROM ONE NUMBER SYSTEM TO ANOTHER

The following number conversions are possible:

1. Decimal to Binary, Octal and Hexadecimal
2. Binary, Octal and Hexadecimal to Decimal
3. Octal to Binary and Binary to Octal
4. Hexadecimal to Binary and Binary to Hexadecimal
5. Octal to Hexadecimal and Hexadecimal to Octal

### Decimal to Binary Conversion

1. First, we divide the integer and successive quotients by 2, till the quotient becomes zero. Then, we write the remainders in the reverse order for getting the integer part of the binary number.
2. Next, we multiply the fractional and successive fractions by 2. The carries are noted until the result is 0 or when the required number of the equivalent digit is obtained. The fractional part of the binary number is obtained by writing the carries in the normal sequence.

Example: $(152.25)_{10} = ( ? )_2$

1. Divide the decimal number 152 and its successive quotients by base 2.

| Operation | Quotient | Remainder |
|---|---|---|
| 152/2 | 76 | 0 (LSB) |
| 76/2 | 38 | 0 |
| 38/2 | 19 | 0 |
| 19/2 | 9 | 1 |
| 9/2 | 4 | 1 |
| 4/2 | 2 | 0 |
| 2/2 | 1 | 0 |
| 1/2 | 0 | 1(MSB) |

Then, we write the remainders in the reverse order for getting the integer part of the binary number as shown below:

$(152)_{10} = (10011000)_2$

2. Now, we multiply the fractional and successive fractions by 2

| Operation | Result | Carry |
|---|---|---|
| 0.25×2 | 0.50 | 0 |
| 0.50×2 | 0 | 1 |

The fractional part of the binary number is obtained by writing the carries in the normal sequence as shown below:
$(0.25)_{10} = (.01)_2$

Answer : $(152.25)_{10} = (10011000.01)_2$

**Decimal to Octal Conversion**

1. First, we divide the integer and successive quotients by base 8, till the quotient becomes zero. Then, we write the remainders in the reverse order for getting the integer part of the octal number.
2. Next, we multiply the fractional and successive fractions by 8. The carries are noted until the result is 0 or when the required number of the equivalent digit is obtained. The fractional part of the octal number is obtained by writing the carries in the normal sequence.

Example: $(152.25)_{10} = ( ? )_8$

1. Divide the decimal number 152 and its successive quotients by base 8.

| **Operation** | **Quotient** | **Remainder** |
|---|---|---|
| 152/8 | 19 | 0 |
| 19/8 | 2 | 3 |
| 2/8 | 0 | 2 |

$(152)_{10} = (230)_8$

2. Now, we multiply the fractional and successive fractions by 8

| **Operation** | **Result** | **Carry** |
|---|---|---|
| 0.25×8 | 0 | 2 |

$(0.25)_{10} = (.2)_8$

Answer : $(152.25)_{10} = (230.2)_8$

**Decimal to hexadecimal conversion**

1. First, we divide the integer and successive quotients by base 16, till the quotient becomes zero. Then, we write the remainders in the reverse order for getting the integer part of the hexadecimal number.
2. Next, we multiply the fractional and successive fractions by 16. The carries are noted until the result is 0 or when the required number of the equivalent digit is obtained. The fractional part of the hexadecimal number is obtained by writing the carries in the normal sequence.

Example: $(152.25)_{10} = ( ? )_{16}$

1. Divide the decimal number 152 and its successive quotients by base 16.

| **Operation** | **Quotient** | **Remainder** |
|---|---|---|
| 152/16 | 9 | 8 |
| 9/16 | 0 | 9 |

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

2. Now, we multiply the fractional and successive fractions by 16.

| **Operation** | **Result** | **Carry** |
|---|---|---|
| 0.25×16 | 0 | 4 |

$(0.25)_{10} = (4)_{16}$

Answer : $(152.25)_{10} = (230.2)_8$

**Binary to Decimal Conversion**

The process of converting binary to decimal is quite simple. The process starts from multiplying the bits of binary number with its corresponding positional weights. And lastly, we add all those products.

Example 1: $(10110.001)_2$

We multiply each bit of the binary number 10110.001 with its respective positional weight, and then we add the products of all the bits with its weight.

$(10110.001)_2=(1\times2^4)+(0\times2^3)+(1\times2^2)+(1\times2^1)+(0\times2^0)+(0\times2^{-1})+(0\times2^{-2})+(1\times2^{-3})$
$(10110.001)_2=(1\times16)+(0\times8)+(1\times4)+(1\times2)+(0\times1)+(0\times1⁄2)+(0\times1⁄4)+(1\times1⁄8)$
$(10110.001)_2=16+0+4+2+0+0+0+0.125$
$(10110.001)_2=(22.125)_{10}$

The decimal equivalent of the binary number 10110.001 is 22.125

**Octal to Decimal Conversion**

The process of converting octal to decimal is carried out by multiplying the digits of the octal number with its corresponding positional weights and then adding all the products.

Example: $(152.25)_8 = ( ? )_{10}$

We multiply each digit of the octal number 152.25 with its respective positional weight, and then add the products.

$(152.25)_8=(1\times8^2)+(5\times8^1)+(2\times8^0)+(2\times8^{-1})+(5\times8^{-2})$
$(152.25)_8=64+40+2+(2\times1⁄8)+(5\times1⁄64)$
$(152.25)_8=64+40+2+0.25+0.078125$
$(152.25)_8=(106.328125)_{10}$

The decimal equivalent of the octal number 152.25 is 106.328125

**Hexadecimal to Decimal Conversion**

The process of converting hexadecimal to decimal is carried out by multiplying the digits of the hexadecimal number with its corresponding positional weights and then adding all the products.

Example: $(152A.25)_{16} = ( ? )_{10}$

We multiply each digit of the hexadecimal number 152A.25 with its respective positional weight, and then add the products.

$(152A.25)_{16}=(1\times16^3)+(5\times16^2)+(2\times16^1)+(A\times16^0)+(2\times16^{-1})+(5\times16^{-2})$
$(152A.25)_{16}=(1\times4096)+(5\times256)+(2\times16)+(10\times1)+(2\times16^{-1})+(5\times16^{-2})$
$(152A.25)_{16}=4096+1280+32+10+(2\times1⁄16)+(5\times1⁄256)$
$(152A.25)_{16}=5418+0.125+0.125$
$(152A.25)_{16}=(5418.14453125)_{10}$

The decimal equivalent of the hexadecimal number 152A.25 is 5418.14453125.

**Binary to Octal Conversion**

Binary numbers can be converted into equivalent octal numbers by making groups of 3 bits on both sides of the binary point and then replacing each group of 3 bits by its octal representation. If there will be one or two bits left in a group of 3 bits, we add the required number of zeros on extreme sides.

Example: $(111110101011.0011)_2$

First, we make group of 3 bits on both sides of the binary point.

111 110 101 011.001 1

On the right side of the binary point, the last pair has only one bit. To make it a complete group of 3 bits, we add two 0's on the extreme right.

111 110 101 011.001 100

Then, we write the octal digits corresponding to each group as shown below.

$(111110101011.0011)_2 = (7653.14)_8$

**Binary to Hexadecimal Conversion**

Binary numbers can be converted into equivalent hexadecimal numbers by making groups of 4 bits on both sides of the binary point. If there will be one, two or three bits left in a group of 4 bits, we add the required number of 0's on extreme sides. Then replace each group of 4 bits by its hexadecimal equivalent.

Example 1: $(10110101011.0011)_2$

First, we make group of 4 bits on both sides of the binary point.

111 1010 1011.0011

On the left side of the binary point, the first group has only bits. To make it a complete pair of 4 bits, add one zero on the extreme left.

0111 1010 1011.0011

2. Then, we write the hexadecimal digits, which correspond to each group as shown below:

$(011110101011.0011)_2 = (7AB.3)_{16}$

**Octal to Binary Conversion**

The process of converting octal to binary is the reverse process of binary to octal. The binary equivalent of the octal number is obtained by converting each octal digit into 3-bit binary.

Example: $(152.25)_8 = ( ? )_2$

We write the 3-bit binary equivalent for each digit of the above given number as shown below:

$(152.25)_8=(001\ 101\ 010.010\ 101)_2$

The binary equivalent of the octal number 152.25 is 001101010.010101.

**Hexadecimal to Binary Conversion**

The process of converting hexadecimal to binary is the reverse process of binary to hexadecimal. The binary equivalent of the octal number is obtained by converting each hexadecimal digit into 4-bit binary.

Example: $(152.A25)_{16}$ = ( ? $)_2$

We write the 4-bit binary equivalent for each digit of the hexadecimal number as shown below:

$(152A.25)_8$ = $(0001\ 0101\ 1010.0010\ 0101)_2$

The binary equivalent of the octal number 152.25 is 000101011010.00100101.

**Octal to hexadecimal conversion**

1. Find the binary equivalent of the octal number by converting each octal digit into 3-bit binary
2. Convert the binary number into hexadecimal equivalent by grouping the binary number into group of 4-bits on both sides of the binary point. If there will be one, two or three bits left in a group of 4 bits, we add the required number of 0's on extreme sides. Then replace each group of 4 bits by its hexadecimal equivalent.

Example: $(152.25)_8$ = ( ? $)_{16}$
Covert each octal digit into 3-bit binary as shown below:

$(152.25)_8$=$(001101010.010101)_2$

Group the binary number into group of 4 bits on both sides of the binary point as shown below:

0 0110 1010 . 0101 01

On the left side of the binary point, the first group has only one digit, and on the right side, the group has only two digits. To make them complete group of 4 bits, add 0's on both extreme sides as shown below:

0000 0110 1010 . 0101 0100

Write the hexadecimal digits for each group of 4 bits as shown below:

(0000 0110 1010 . 0101 $0100)_2$ = $(6A.54)_{16}$

The hexadecimal equivalent of the given octal number 152.25 is 6A.54

Hexadecimal to other Number Systems

**Hexadecimal to Octal Conversion**

1. Find the binary equivalent of the hexadecimal number by converting each octal digit into 4-bit binary
2. Convert the binary number into octal equivalent by grouping the binary number into group of 3-bits on both sides of the binary point. If there will be one, two or three bits left in a group of 3 bits, we add the required number of 0's on extreme sides. Then replace each group of 3 bits by its octal equivalent.

Example: $(152A.25)_{16} = ( ? )_8$

Covert each hexadecimal digit into 4-bit binary as shown below:

$(152A.25)_8=(0001\ 0101\ 0010\ 1010\ .\ 0010\ 0101)_2$

Group the binary number into group of 3 bits on both sides of the binary point as shown below:

0 001 010 100 101 010 . 001 001 01

On the left side of the binary point, the first group has only one digit, and on the right side, the last group has only two digits. To make them complete group of 3 bits, add 0's on both extreme sides as shown below:

000 001 010 100 101 010 . 001 001 010

Write the octal digits for each group of 3 bits as shown below:

$(001\ 010\ 100\ 101\ 010\ .\ 001\ 001\ 010)_2 = (12452.112)_8$

The octal equivalent of the given hexadecimal number 152A.25 is 12452.112

**ADDITION, SUBTRACTION, MULTIPLICATION & DIVISION OF BINARY, OCTAL & HEXADECIMAL NUMBERS**

**Binary Addition:**

0 + 0 = 0,  0 + 1 = 1,  1 + 0 = 1,  1 + 1 = 10

Examples:

```
  1011              10101
  1001 +            11001 +
 --------          ---------
 10100             101110
 --------          ---------
```

**Binary Subtraction:**

Examples:

```
  1011              10101
  1001 -            01011 -
 -------           ---------
  0010              01010
 -------           ---------
```

**Binary Multiplication:**

0 x 0 = 0,    0 x 1 = 0,    1 x 0 = 0,    1 x 1 = 1

**Examples:**

```
     10111                            11001
       101 x                            111 x
  ------------                    ------------
     10111                            11001
    00000                            11001
   10111                            11001
  ------------                    ------------
   1110011                        10101111
  ------------                    ------------
```

**Binary Division:**

```
101 ) 1 1 0 1 0 ( 101
    - 1 0 1
      -------
         1 1 0
        - 1 0 1
        --------
           0 0 1
```

**Octal Addition:**

The following table will help you to handle octal addition.

A

| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 10 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 10 | 11 |
| 3 | 3 | 4 | 5 | 6 | 7 | 10 | 11 | 12 |
| 4 | 4 | 5 | 6 | 7 | 10 | 11 | 12 | 13 |
| 5 | 5 | 6 | 7 | 10 | 11 | 12 | 13 | 14 |
| 6 | 6 | 7 | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

Sum

B

**Example:**

```
4 5 6
1 2 3 +
-------
6 0 1
-------
```

**Octal Subtraction:**

**Example:**

```
4 5 6
1 7 3 -
-------
2 6 3
-------
```

**Octal Multiplication:**

**Example:**

```
  4 4
  1 5  x
-------
2 6 4
4 4
-------
7 2 4
```

**Octal Division:**

Example: $(6573)_8$ / $(16)_8$

First let us make a table for 16 and its multiples.

| | Decimal | Octal |
|---|---|---|
| 16*1 | 16 | 14 |
| 16*2 | 32 | 34 |
| 16*3 | 48 | 52 |
| 16*4 | 64 | 70 |
| 16*5 | 80 | 106 |
| 16*6 | 96 | 124 |

```
16 ) 6 5 7 3 (366.4
     5 2
     1 3 7
     1 2 4
       1 3 3
       1 2 4
         7 0
         7 0
        -----
         0 0
```

**Hexadecimal Addition:**

The following table will help you to handle hexadecimal addition.

| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 |
| 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 |
| 4 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 |
| 5 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 |
| 6 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 7 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 8 | 8 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 9 | 9 | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| A | A | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| B | B | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A |
| C | C | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B |
| D | D | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C |
| E | E | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D |
| F | F | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 1A | 1B | 1C | 1D | 1E |

**Example:**

```
4 A 6
1 B 3 +
-------
6 5 9
-------
```

**Hexadecimal Subtraction:**

**Example:**

```
4 A 6
1 B 3 -
-------
2 F 3
-------
```

**Hexadecimal Multiplication:**

**Example:**

```
     C B
     A 2  x
----------
   1 9 6
7 E E
---------
8 0 7 6
---------
```

**Hexadecimal Division:**

**Example:**

```
A ) B 8 4 F ( 1 2 6 E
     A
    -----
     1 8
     1 4
    -------
       4 4
       3 C
      -------
         8 F
         8 C
        -----
           3
```

## COMPLEMENTS

Complements are used in digital computers to represent signed numbers and hence to simplify subtraction operations. We have two categories of complements, namely 1's and 2's complements for binary numbers and 9's and 10's complements for decimal numbers. These complements are discussed below.

**1's Complement:** The 1's complement of a binary number is obtained by subtracting each bit of the given number from 1. The 1's complement is also obtained by changing all the 0's to 1's and all the 1's to 0's.

Examples: 1. The 1's complement of the binary number 010011 is 101100.

2. The 1's complement of the binary number 110110 is 001001.

**2's Complement:** The 2's complement of a binary number is obtained by finding the 1's complement of the number and then adding a 1 to it.

*2's complement = 1's complement + 1*

Examples: 1. The 2's complement of the binary number 010011 is obtained by the following steps.

1's complement = 101100
2's complement = 101100 + 1 = 101101

2. The 2's complement of the binary number 110110 is obtained by the following steps.

1's complement = 001001
2's complement = 001001 + 1 = 001010

**9’s Complement:** The 9’s complement of a decimal number is obtained by subtracting each digit of the given number from 9.

Examples: 1. The 9’s complement of the decimal number 3549 is 6450

2. The 9’s complement of the decimal number 1287 is 8712

**10’s Complement:** The 10’s complement of a decimal number is obtained by finding the 9’s complement of the number and then adding a 1 to it.

*10’s complement = 9’s complement + 1*

Examples: 1. The 10’s complement of the binary number 3549 is obtained by the following steps.

9’s complement = 6450
10’s complement = 6450 + 1 = 6451

2. The 10’s complement of the binary number 1287 is obtained by the following steps.

9’s complement = 8712
10’s complement = 8712 + 1 = 8713

## BINARY CODES

The digital data is represented, stored and transmitted as group of binary bits. This group is also called as **binary code**. The binary code is represented by the number as well as alphanumeric letter.

### Advantages of Binary Codes

- Binary codes are suitable for the computer applications.
- Binary codes are suitable for the digital communications.
- Since only 0 & 1 are being used, implementation becomes easy.

### Classification of Binary Codes

The codes are broadly categorized into following four categories.

- Weighted codes
- Non-weighted codes
- Binary coded decimal codes
- Self-complementing codes
- Alphanumeric codes

### Weighted Codes

Weighted binary codes are those binary codes which obey the positional weight principle. Each position of the number represents a specific weight. Several systems of the codes are used to express the decimal digits 0 through 9. In these codes each decimal digit is represented by a group of four bits. Examples of weighted codes are 8421 code and 2421 code.

**8421 Code:**

The 8421 code is a weighted code in which each decimal digit 0 through 9 is represented by a 4-bit binary word. Each bit has a weight 8, 4, 2, 1 from left to right. An 8421 code is a BCD code. For example, the decimal number 24 is represented in 8421 code as 0010 0100.

**2421 Code:**

The 2421 code is a weighted code in which each decimal digit 0 through 9 is represented by a 4-bit binary word. Each bit has a weight 2, 4, 2, 1 from left to right. A 2421 code is also a BCD code. The 8421 and 2421 codes of the decimal digits 0 to 9 are shown in the table below:

| Decimal Digit | 8421 Code | 2421 Code |
|---|---|---|
| 0 | 0000 | 0000 |
| 1 | 0001 | 0001 |
| 2 | 0010 | 0010 |
| 3 | 0011 | 0011 |
| 4 | 0100 | 0100 |
| 5 | 0101 | 1011 |
| 6 | 0110 | 1100 |
| 7 | 0111 | 1101 |
| 8 | 1000 | 1110 |
| 9 | 1001 | 1111 |

**Non-Weighted Codes**

In this type of binary codes, the positional weights are not assigned. The examples of non-weighted codes are Excess-3 code and Gray code.

**Excess-3 Code**

The Excess-3 code (also written as XS-3 code) is non-weighted code used to express decimal numbers. The Excess-3 code equivalent of a decimal number is obtained by adding a 3 to each digit of the number and then representing each digit sum as a 4-bit binary word.

The table below shows the Excess-3 codes for the decimal numbers 0 to 9.

| Decimal Digit | Excess-3 Code |
|---|---|
| 0 | 0011 |
| 1 | 0100 |
| 2 | 0101 |
| 3 | 0110 |
| 4 | 0111 |
| 5 | 1000 |
| 6 | 1001 |
| 7 | 1010 |
| 8 | 1011 |
| 9 | 1100 |

The Excess-3 code of the decimal number 469 is obtained by adding a 3 to each digits 4, 6 & 9. Each digit sum is then represented as a 4-bit binary word. This is illustrated below:

| | | |
|---|---|---|
| 4 | 6 | 9 |
| 3+ | 3+ | 3+ |
| --- | --- | --- |
| 7 | 9 | 12 |
| 0111 | 1001 | 1100 |

The Excess-3 equivalent of the decimal number 469 = 0111 1001 1100

**Gray Code**

The gray code is a code in which two successive numbers differ by one bit position only. For example, decimal numbers 13 and 14 are represented by gray code numbers 1011 and 1001, these numbers differ only in single position that is the second position from the right. In the same way first position on the left changes for 7 and 8 which are 0100 and 1100 (refer to the table below). As only one bit changes at a time, the gray code is also referred to as a **unit distance code** or a **reflected code.**

**Binary Code to Gray Code Conversion**

In the gray code, the 1$^{st}$ bit (MSB) will always be the same as the 1'st bit of the given binary number. So, write the 1$^{st}$ bit as it is. The remaining bits of the gray code is obtained by performing the XOR operation between the successive bits of the binary number. If both the bits are different, the result will be 1, else the result will be 0.

The 2$^{nd}$ bit of the gray number is obtained by performing the XOR operation between the 1$^{st}$ and 2$^{nd}$ bits of the binary number. The 3$^{rd}$ bit of the gray number is obtained by performing the XOR operation between the 2$^{nd}$ and 3$^{rd}$ bits of the binary number, and so on. The gray code equivalents of the binary codes (for the decimal numbers) are given in the table below:

| Decimal | Binary Code | Gray Code |
|---|---|---|
| 0 | 0000 | 0000 |
| 1 | 0001 | 0001 |
| 2 | 0010 | 0011 |
| 3 | 0011 | 0010 |
| 4 | 0100 | 0110 |
| 5 | 0101 | 0111 |
| 6 | 0110 | 0101 |
| 7 | 0111 | 0100 |
| 8 | 1000 | 1100 |
| 9 | 1001 | 1101 |
| 10 | 1010 | 1111 |
| 11 | 1011 | 1110 |
| 12 | 1100 | 1010 |
| 13 | 1101 | 1011 |
| 14 | 1110 | 1001 |
| 15 | 1111 | 1000 |

### Gray Code to Binary Code Conversion

In the binary code, the 1st bit (MSB) will always be the same as the 1'st bit of the given gray number. So, write the 1st bit as it is. The 2nd bit of the binary number is obtained by performing the XOR operation between the 1st bit of the binary number and the 2nd bit of the gray number. The 3rd bit of the binary number is obtained by performing the XOR operation between the 2nd bit of the binary number and the 3rd bit of the gray number. The 4th bit of the binary number is obtained by performing the XOR operation between the 3rd bit of the binary number and the 4th bit of the gray number, and so on.

Example:

The binary equivalent of the gray code 1101 is 1001

### Binary Coded Decimal (BCD) Code

The BCD code of a decimal number is obtained by representing each digit of the number by a 4-bit binary code. Examples of BCD codes are 8421 code, 2421 code and Excess-3 code. The various BCD code equivalents of the decimal numbers 0 through 15 are shown in the table below:

| Decimal | 8421 Code | 2421 Code | XS-3 Code |
|---|---|---|---|
| 0 | 0000 | 0000 | 0011 |
| 1 | 0001 | 0001 | 0100 |
| 2 | 0010 | 0010 | 0101 |
| 3 | 0011 | 0011 | 0110 |
| 4 | 0100 | 0100 | 0111 |
| 5 | 0101 | 1011 | 1000 |
| 6 | 0110 | 1100 | 1001 |
| 7 | 0111 | 1101 | 1010 |
| 8 | 1000 | 1110 | 1011 |
| 9 | 1001 | 1111 | 1100 |
| 10 | 0001 0000 | 0001 0000 | 0100 0011 |
| 11 | 0001 0001 | 0001 0001 | 0100 0100 |
| 12 | 0001 0010 | 0001 0010 | 0100 0101 |
| 13 | 0001 0011 | 0001 0011 | 0100 0110 |
| 14 | 0001 0100 | 0001 0100 | 0100 0111 |
| 15 | 0001 0101 | 0001 1011 | 0100 1000 |

### Self-complementing Code:

A self-complementing code is a code in which the 9's complement of a number is obtained by complementing each bit of the number. Examples of self-complementing code are 2421 code, 5211 code and Excess-code. The different self-complementing codes are shown in the table below:

| Decimal | 2421 Code | 5211 Code | XS-3 Code |
|---|---|---|---|
| 0 | 0000 | 0000 | 0011 |
| 1 | 0001 | 0001 | 0100 |
| 2 | 0010 | 0011 | 0101 |
| 3 | 0011 | 0101 | 0110 |
| 4 | 0100 | 0111 | 0111 |
| 5 | 1011 | 1000 | 1000 |
| 6 | 1100 | 1010 | 1001 |
| 7 | 1101 | 1100 | 1010 |
| 8 | 1110 | 1110 | 1011 |
| 9 | 1111 | 1111 | 1100 |

**Alphanumeric Codes:**

A binary digit or bit can represent only two symbols as it has only two states '0' or '1'. But this is not enough for communication between two computers because there we need many more symbols for communication. These symbols are required to represent 26 alphabets with capital and small letters, numbers from 0 to 9, punctuation marks and other special characters.

The alphanumeric codes are the codes that represent numbers, alphabetic and other special characters. The following two alphanumeric codes are very commonly used for the data representation.

- American Standard Code for Information Interchange (ASCII).
- Extended Binary Coded Decimal Interchange Code (EBCDIC).

ASCII code is a 7-bit code whereas EBCDIC is an 8-bit code. ASCII code is more commonly used worldwide while EBCDIC is used primarily in large IBM computers. The following tables show the ASCII and EBCDIC codes for a few alphanumeric characters.

| ASCII | Symbol | ASCII | Symbol | ASCII | Symbol | ASCII | Symbol | ASCII | Symbol |
|---|---|---|---|---|---|---|---|---|---|
| 010 0000 | Space | 011 0000 | 0 | 100 0111 | G | 101 0111 | W | 110 1101 | m |
| 010 0001 | ! | 011 0001 | 1 | 100 1000 | H | 101 1000 | X | 110 1110 | n |
| 010 0010 | " | 011 0010 | 2 | 100 1001 | I | 101 1001 | Y | 110 1111 | o |
| 010 0011 | # | 011 0011 | 3 | 100 1010 | J | 101 1010 | Z | 111 0000 | p |
| 010 0100 | $ | 011 0100 | 4 | 100 1011 | K | 110 0001 | a | 111 0001 | q |
| 010 0101 | % | 011 0101 | 5 | 100 1100 | L | 110 0010 | b | 111 0010 | r |
| 010 0110 | & | 011 0110 | 6 | 100 1101 | M | 110 0011 | c | 111 0011 | s |
| 010 0111 | ' | 011 0111 | 7 | 100 1110 | N | 110 0100 | d | 111 0100 | t |
| 010 1000 | ( | 011 1000 | 8 | 100 1111 | O | 110 0101 | e | 111 0101 | u |
| 010 1001 | ) | 011 1001 | 9 | 101 0000 | P | 110 0110 | f | 111 0110 | v |
| 010 1010 | * | 100 0001 | A | 101 0001 | Q | 110 0111 | g | 111 0111 | w |
| 010 1011 | + | 100 0010 | B | 101 0010 | R | 110 1000 | h | 111 1000 | x |
| 010 1100 | , | 100 0011 | C | 101 0011 | S | 110 1001 | i | 111 1001 | y |
| 010 1101 | - | 100 0100 | D | 101 0100 | T | 110 1010 | j | 111 1010 | z |
| 010 1110 | . | 100 0101 | E | 101 0101 | U | 110 1011 | k | | |
| 010 1111 | / | 100 0110 | F | 101 0110 | V | 110 1100 | l | | |

| EBCDIC | Symbol | EBCDIC | Symbol | EBCDIC | Symbol | EBCDIC | Symbol | EBCDIC | Symbol |
|---|---|---|---|---|---|---|---|---|---|
| 0100 0000 | Space | 1000 0001 | a | 1001 1000 | q | 1100 0111 | G | 1110 0110 | W |
| 0100 1011 | . | 1000 0010 | b | 1001 1001 | r | 1100 1000 | H | 1110 0111 | X |
| 0100 1100 | < | 1000 0011 | c | 1010 0010 | s | 1100 1001 | I | 1110 1000 | Y |
| 0100 1101 | ( | 1000 0100 | d | 1010 0011 | t | 1101 0001 | J | 1110 1001 | Z |
| 0100 1110 | + | 1000 0101 | e | 1010 0100 | u | 1101 0010 | K | 1111 0000 | 0 |
| 0101 0000 | & | 1000 0110 | f | 1010 0101 | v | 1101 0011 | L | 1111 0001 | 1 |
| 0101 1100 | * | 1000 0111 | g | 1010 0110 | w | 1101 0100 | M | 1111 0010 | 2 |
| 0101 1101 | ) | 1000 1000 | h | 1010 0111 | x | 1101 0101 | N | 1111 0011 | 3 |
| 0101 1110 | ; | 1000 1001 | i | 1010 1000 | y | 1101 0110 | O | 1111 0100 | 4 |
| 0110 0000 | - | 1001 0001 | j | 1010 1001 | z | 1101 0111 | P | 1111 0101 | 5 |
| 0110 1011 | , | 1001 0010 | k | 1100 0001 | A | 1101 1000 | Q | 1111 0110 | 6 |
| 0110 1101 | _ | 1001 0011 | l | 1100 0010 | B | 1101 1001 | R | 1111 0111 | 7 |
| 0110 1110 | > | 1001 0100 | m | 1100 0011 | C | 1110 0010 | S | 1111 1000 | 8 |
| 0111 1100 | @ | 1001 0101 | n | 1100 0100 | D | 1110 0011 | T | 1111 1001 | 9 |
| 0111 1101 | ‘ | 1001 0110 | o | 1100 0101 | E | 1110 0100 | U | | |
| 0111 1110 | = | 1001 0111 | p | 1100 0110 | F | 1110 0101 | V | | |

The string – ‘WELCOME’ is represented using ASCII and EBCDIC codes as shown below.

| | W | E | L | C | O | M | E |
|---|---|---|---|---|---|---|---|
| **ASCII** | 101 0111 | 100 0101 | 100 1100 | 100 0011 | 100 1111 | 100 1101 | 100 0101 |
| **EBCDIC** | 1110 0110 | 1100 0101 | 1101 0011 | 1100 0011 | 1101 0110 | 1101 0100 | 1100 0101 |

# UNIT-2

## BOOLEAN ALGEBRA

Boolean algebra is the branch of algebra that is used to analyze and simplify digital (logic) circuits. The values of the variables are the truth values TRUE and FALSE, usually denoted as 1 and 0, respectively. The basic operations of Boolean algebra are the 'AND' operation indicated by a dot (.) between variables, the 'OR' operation indicated by a plus symbol (+) between variables and the 'NOT' operation indicated by a bar (-) over a variable.

## BOOLEAN LAWS AND THEOREMS

The Boolean laws and theorems are used to reduce and simplify complex Boolean expressions in an attempt to reduce the number logic gates required to construct logic circuits. Some of the important Boolean laws and theorems are discussed below:

*Commutative law:* Commutative law states that changing the sequence of the variables does not have any effect on the output. The laws are:

(i) $A.B = B.A$ (ii) $A + B = B + A$

*Associative law:* This law states that the order in which the logic operations are performed is irrelevant as their effect is the same. The laws are:

(i) $(A.B).C = A.(B.C)$ (ii) $(A + B) + C = A + (B + C)$

*Distributive law:* Distributive law states the following condition.

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

*Identity Law:* A variable ORed with a "0" or ANDed with a "1" will always be equal to that variable.

(i) $A + 0 = A$ (ii) $A.1 = A$

*Annulment Law*: A variable ANDed with a "0" equals 0 or ORed with a "1" will be equal to 1.

(i) $A.0 = 0$ (ii) $A + 1 = 1$

*Idempotent Law*: A variable that is ANDed or ORed with itself is equal to that variable.

(i) $A + A = A$ (ii) $A.A = A$

*Complement Law:* A variable ANDed with its complement equals "0" and a variable ORed with its complement equals "1"

(i) $A.\bar{A} = 0$ (ii) $A + \bar{A} = 1$

*Inversion law:* This law uses the NOT operation. The inversion law states that double inversion of variable results in the original variable itself.

$\bar{\bar{A}} = A$

*Absorption Law*: This law enables a reduction in a complicated expression to a simpler one by absorbing like terms.

(i) $A + A.B = A$ (ii) $A.(A + B) = A$

*Other Laws:* (i) $A + \bar{A}B = A + B$ (i) $A.(\bar{A} + B) = A.B$

**De Morgan's Theorems:**

There are two De Morgan's theorems in Boolean Algebra. They are:

1. First theorem : $\overline{A+B} = \bar{A}.\bar{B}$
2. Second theorem: $\overline{A.B} = \bar{A}+\bar{B}$

**First theorem : $\overline{\boldsymbol{A}+\boldsymbol{B}} = \bar{\boldsymbol{A}}.\bar{\boldsymbol{B}}$**

This theorem states that the complement of a logical sum of two variables is equivalent to the logical product of the complements of individual variables.

*Proof using Truth Tables:*

| $\boldsymbol{A}$ | $\boldsymbol{B}$ | $\overline{\boldsymbol{A}+\boldsymbol{B}}$ | $\bar{\boldsymbol{A}}.\bar{\boldsymbol{B}}$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |

From the above truth table we find that $\overline{A+B} = \bar{A}.\bar{B}$. According to this theorem, a NOR gate is equivalent to a bubbled-AND gate.

**Second theorem : $\overline{\boldsymbol{A}.\boldsymbol{B}} = \bar{\boldsymbol{A}}+\bar{\boldsymbol{B}}$**

This theorem states that the complement of a logical product of two variables is equivalent to the logical sum of the complements of individual variables.

*Proof using Truth Tables:*

| $\boldsymbol{A}$ | $\boldsymbol{B}$ | $\overline{\boldsymbol{A}.\boldsymbol{B}}$ | $\bar{\boldsymbol{A}}+\bar{\boldsymbol{B}}$ |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |

From the above truth table we find that $\overline{A.B} = \bar{A}+\bar{B}$. According to this theorem, a NAND gate is equivalent to a bubbled-OR gate.

**Complement of a Boolean Function:**

The complement of a Boolean function is obtained using De Morgan's theorems as follows:

- Complement all the variables in the given function
- Change all OR (+) operations into AND (.) operations and vice versa.

Example:

Given Boolean function, $F = A\bar{B} + B\bar{C} + \bar{A}C$

The Complement of the above Boolean function is $F = (\bar{A}+B).(\bar{B}+C).(A+\bar{C})$

## BOOLEAN EXPRESSIONS

There are two forms of Boolean expressions in Boolean algebra. They are:

1. Sum of Products expression (SOP)
2. Product of Sums expression (POS)

***Sum of Products expression (SOP):*** A Boolean expression consisting of logical products (minterms) separated by OR ('+') operators is referred to as a sum of products expression.

Examples: $F = AB + BC + AC$

$F = \bar{A}BC + A\bar{B}C + AB\bar{C}$

***Product of Sums expression (POS):*** A Boolean expression consisting of logical sums (maxterms) separated by AND ('.') operators is referred to as a product of sums expression.

Examples: $F = (A + B).(B + C).(A + C)$

$F = (\bar{A} + B + C).(A + \bar{B} + C).(A + B + \bar{C})$

## LOGIC GATES

A **logic gate** is a basic building block of a digital circuit that works on the principles of Boolean algebra. Logic gates have one or more inputs and only one output. The relationship between the input and the output is based on certain logic.

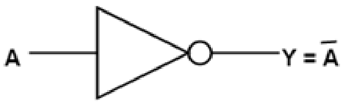
## BASIC LOGIC GATES

There are three basic logic gates. They are:

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

### NOT gate:

The logic symbol of a NOT gate is shown in the figure below.

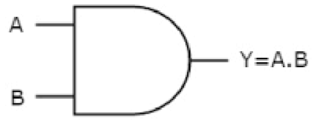



A NOT gate has one input signal and one output signal. The output of a NOT gate is the complement of its input. If the input of a NOT gate is low (0), the output is high (1) and if the input is high (1), the output is low (0). A NOT gate is also referred to as an inverter or a complementer. The working of a NOT gate is illustrated in the truth table given below:

| **Input A** | **Output $\bar{A}$** |
|---|---|
| 0 | 1 |
| 1 | 0 |

**AND gate:**

An AND gate has two or more input signals and one output signal. The logic symbol of 2-input AND gate is shown in the figure below. A and B are inputs and Y (= A.B) is the output.

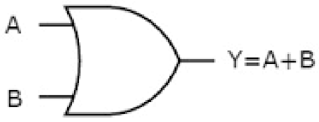



The output of an AND gate is high (1) when all the inputs are high. The output is low when any one of the inputs in low. The truth table of a 2-input AND gate is given in the table below.

| **Inputs** | | **Output** |
|---|---|---|
| **A** | **B** | **Y=A.B** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**OR gate:**

An OR gate has two or more input signals and one output signal. The logic symbol of a 2-input OR gate is shown below.



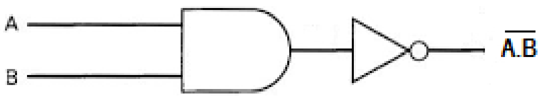

The output of an OR gate is high (1) when any one of the inputs is high (1). The output is low (0) only when all the inputs are low (0). The truth table of a 2-input OR gate is given below.

| **Inputs** | | **Output** |
|---|---|---|
| **A** | **B** | **Y=A+B** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**OTHER LOGIC GATES**

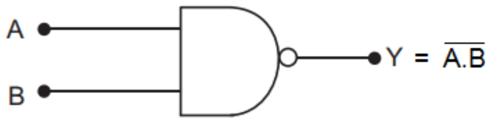
**NAND gate:**

A NAND gate has two or more input signals and one output signal. A NAND gate is constructed using an AND gate and a NOT gate as shown below:




AND + NOT = NAND

The logic symbol of a 2-input NAND gate is shown below:



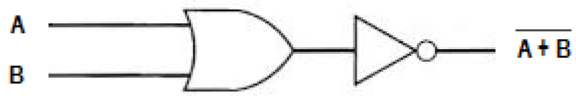

The output of a NAND gate is high when any one of the inputs is low. The output is low when all the inputs are high. The truth table of a NAND gate is given below:

| Inputs | | Output |
|---|---|---|
| **A** | **B** | $\overline{A.B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The output of a NAND gate is the complement of an AND gate.

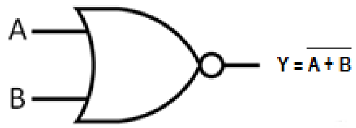
**NOR gate:**

A NOR gate has two or more input signals and one output signal. A NOR gate is constructed using an OR gate and a NOT gate as shown below:




OR + NOT = NOR

The logic symbol of a 2-input NOR gate is shown below:



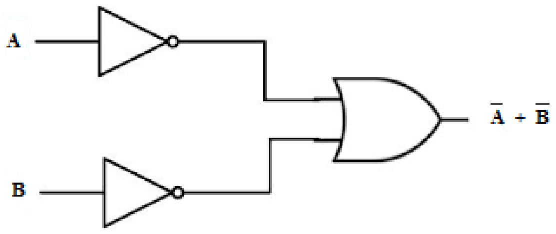

The output of a NOR gate is high when all the inputs are low. The output is low when any one of the inputs is high. The truth table of a NOR gate is given below:

| Inputs | | Output |
|---|---|---|
| **A** | **B** | $\overline{A + B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

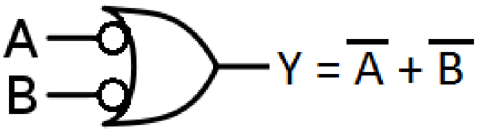
The output of a NOR gate is the complement of an OR gate.

**Bubbled-OR gate: (Invert OR)**

A bubbled-OR gate is constructed using two NOT gates and one OR gate as shown in the following diagram.




The logic symbol of a bubbled-OR gate is shown below.



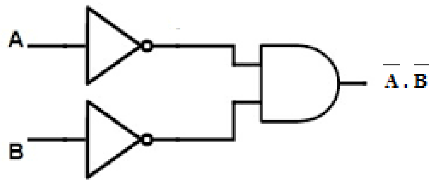

The truth table of a bubbled-OR gate is given below.

| **Inputs** | | **Output** |
|---|---|---|
| **A** | **B** | $\bar{A} + \bar{B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

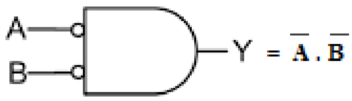
The output of a bubbled-OR gate is high when any one of the inputs is low. The output is low when all the inputs are high. A bubbled-OR gate is equivalent to a NAND gate.

**Bubbled-AND gate:**

A bubbled-AND gate is constructed using two NOT gates and one AND gate as shown in the following diagram.




The logic symbol of a bubbled-AND gate is shown below.



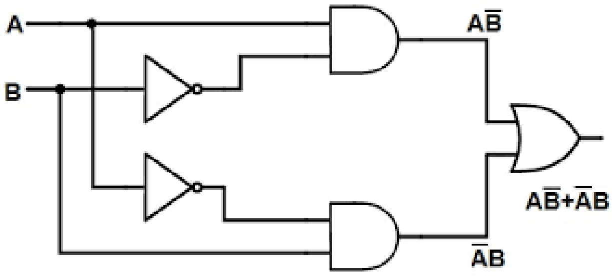

The truth table of a bubbled-AND gate is given below.

| **Inputs** | | **Output** |
|---|---|---|
| **A** | **B** | $\bar{A} . \bar{B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

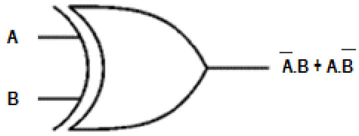
The output of a bubbled-AND gate is high when any one of the inputs is low. The output is low when all the inputs are high. A bubbled-AND gate is equivalent to a NOR gate.

**Exclusive-OR gate (EX-OR or XOR gate):**

An EX-OR gate (sometimes referred to as Exclusive-OR gate) is a digital logic gate with two or more inputs and one output. The output of an exclusive-OR gate is expressed as a Boolean function, $Y = A\bar{B} + \bar{A}B$. The circuit diagram of an EX-OR gate is shown following figure.




The logic symbol of an EX-OR gate is given below:



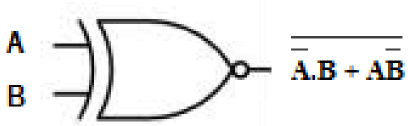

The output of the EX-OR gate is expressed in the truth table given below:

| Inputs | | Output |
|---|---|---|
| **A** | **B** | $\bar{A}B + A\bar{B}$ |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The output of an EX-OR gate is high when the inputs are different. The output is low when the inputs are same. In general, the output of an EX-OR gate is high when the number of high inputs is odd. The output is low otherwise.

**Exclusive-NOR gate (EX-NOR or XNOR gate):**

The logic symbol of an EX-OR gate is given below:




The output of the EX-NOR gate is expressed in the truth table given below:

| Inputs | | Output |
|---|---|---|
| **A** | **B** | $\overline{\bar{A}B + A\bar{B}}$ |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

The output of an EX-NOR gate is low when the inputs are different and the output is high when the inputs are same. In general, the output of an EX-NOR gate is low when the number of high inputs is odd. The output is high otherwise.

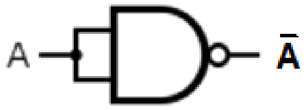
## UNIVERSAL LOGIC GATES

NAND and NOR gates are referred to as universal gates. They are called so because all the other gates can be constructed using these gates.

**NAND as a Universal Gate:**

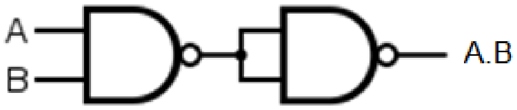
**NAND as a NOT gate:**

A NOT gate is constructed using a NAND gate by combining both of its inputs into a single input as shown below. The output of this circuit is the complement of its input.




**NAND as AND gate:**

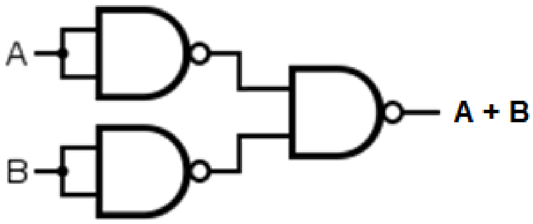
An AND gate is constructed using two NAND gates as shown below:




The output of the first NAND gate is $\overline{A.B}$. This output is given as input to the second NAND gate to obtain A.B as the final output.

**NAND as OR gate:**

An OR gate is constructed using three NAND gates as shown in the circuit below.

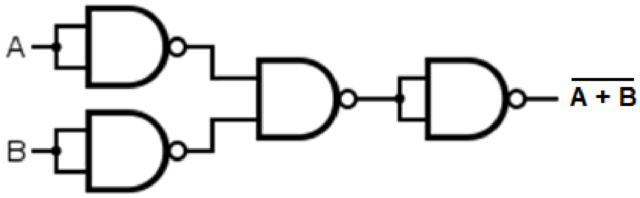



The output of the first NAND gate is $\bar{A}$ and that of the second NAND gate is $\bar{B}$. These two outputs are fed into the third NAND gate as inputs to obtain A+B as the final output. This is shown in the expression below.

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

**NAND as NOR gate:**

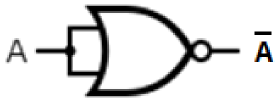
A NOR gate is constructed using four NAND gates. The output of the OR gate (see previous diagram) is connected to the input of a NOT gate as shown below.




**NOR as a Universal Gate:**

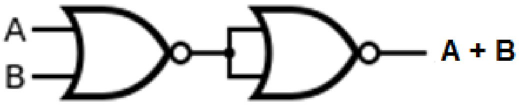
**NOR as NOT gate:**

A NOT gate is constructed using a NOR gate by combining both of its inputs into a single input as shown below. The output of this circuit is the complement of its input.



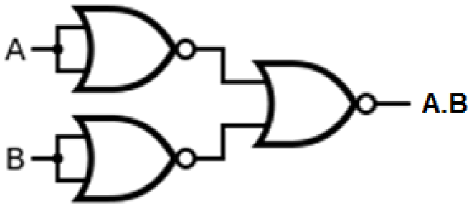

**NOR as OR gate:**

An OR gate is constructed using two NOR gates as shown below. The output of the first NAND gate is $\overline{A + B}$. This output is given as input to the second NAND gate to obtain A+B as the final output.




**NOR as AND gate:**

An AND gate is constructed using three NOR gates as shown in the circuit below.

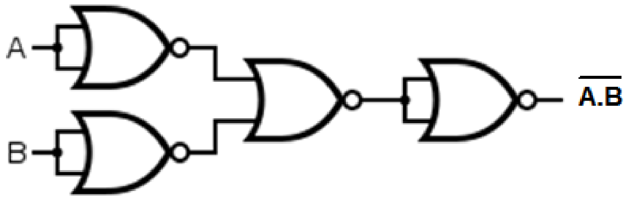



The output of the first NOR gate is $\bar{A}$ and that of the second NOR gate is $\bar{B}$. These two outputs are fed into the third NOR gate as inputs to obtain A.B as the final output. This is shown in the expression below.

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

**NOR as NAND gate:**

A NAND gate is constructed using four NOR gates. The output of the AND gate (see previous diagram) is connected to the input of a NOT gate as shown below.

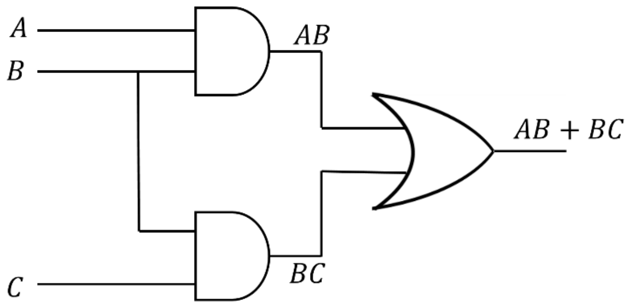



## IMPLEMENTATION OF BOOLEAN EXPRESSIONS USING GATE NETWORKS

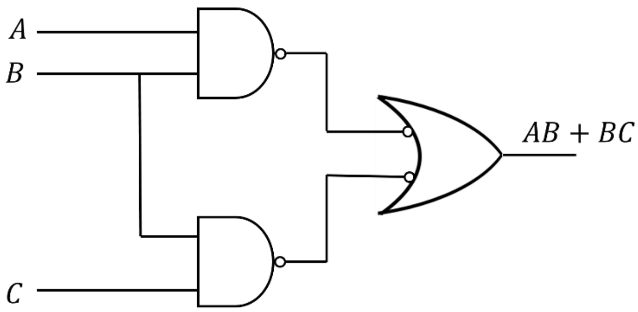
*SOP Expression:* An SOP form of expression can be implemented using AND gates in the first level and OR gates in the second level. The resultant circuit is referred to as a AND-OR circuit.

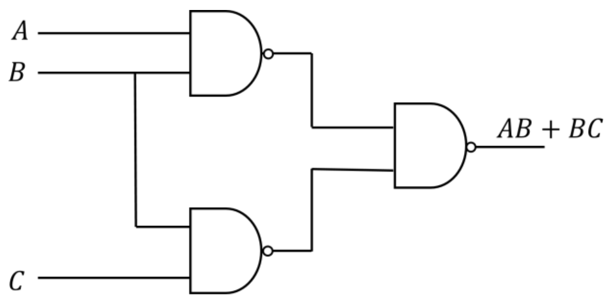
Consider the Boolean expression, $F = AB + BC$.

This expression can be implemented using AND-OR circuit as shown in the following figure.




Any AND-OR circuit can be converted into a NAND-NAND circuit by replacing all the gates in the AND-OR circuit with NAND gates. This is illustrated below.

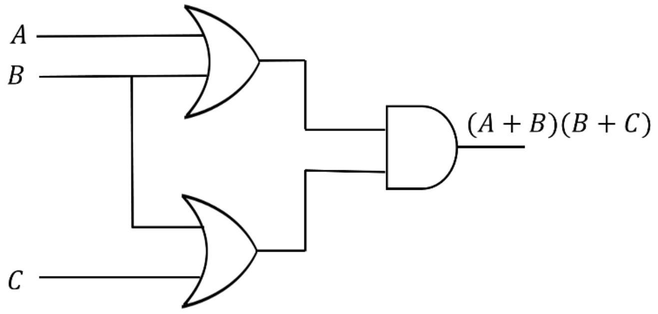


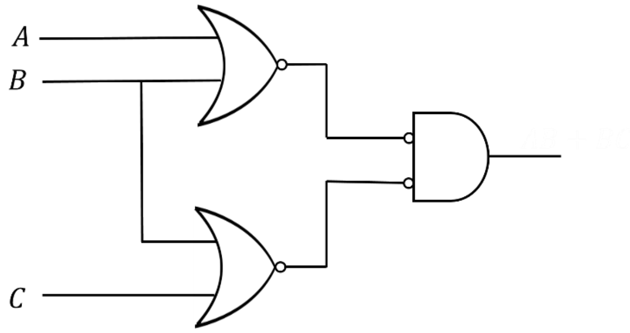
*POS Expression:* A POS form of expression can be implemented using OR gates in the first level and AND gates in the second level. The resultant circuit is referred to as a OR-AND circuit.

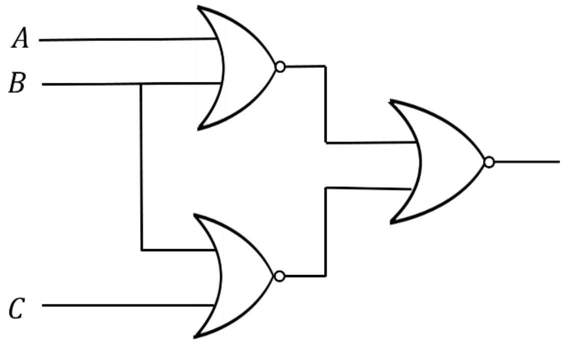
Consider the Boolean expression, $F = (A + B)(B + C)$.

This expression can be implemented using OR-AND circuit as shown in the following figure.




Any OR-AND circuit can be converted into a NOR-NOR circuit by replacing all the gates in the OR-AND circuit with NOR gates. This is illustrated below.




**SIMPLIFICATION OF BOOLEAN EXPRESSIONS**

The most practical use of Boolean algebra is to simplify logic circuits. A Boolean expression can be implemented directly in a logic circuit. The number of terms and operations in a Boolean expression is directly related to the number of logic components. Through Boolean algebra simplification, a Boolean expression is translated to another form with less number of terms and operations. A logic circuit for the simplified Boolean expression performs the identical function with fewer logic components as compared to its original form. Additionally, the simplified Boolean expression when implemented to a logic circuit is reliable with a reduced cost.

**Simplification of Boolean Expressions using laws of Boolean algebra:**

1. F = $\bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$

   $= \bar{A}BC + A\bar{B}C + AB(\bar{C} + C)$

   $= \bar{A}BC + A\bar{B}C + AB$ [Since $\bar{C} + C = 1$]

   $= \bar{A}BC + A(\bar{B}C + B)$

   $= \bar{A}BC + A(B + \bar{B}C)$

   $= \bar{A}BC + A(B + C)$ [Since $B + \bar{B}C = B + C$]

   $= \bar{A}BC + AB + AC$

   $= B(\bar{A}C + A) + AC$

   $= B(A + \bar{A}C) + AC$

   $= B(A + C) + AC$ [Since $A + \bar{A}C = A + C$]

   $= AB + BC + AC$

2. F = $\bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + ABC$

   $= \bar{A}\bar{B}C + \bar{A}B\bar{C} + BC(\bar{A} + A)$

   $= \bar{A}\bar{B}C + \bar{A}B\bar{C} + BC$ [Since $\bar{A} + A = 1$]

   $= \bar{A}\bar{B}C + B(\bar{A}\bar{C} + C)$

   $= \bar{A}\bar{B}C + B(\bar{A} + C)$ [Since $\bar{A}\bar{C} + C = \bar{A} + C$]

   $= \bar{A}\bar{B}C + B\bar{A} + BC$

   $= \bar{A}\bar{B}C + \bar{A}B + BC$

   $= \bar{A}(\bar{B}C + B) + BC$

   $= \bar{A}(B + \bar{B}C) + BC$

   $= \bar{A}(B + C) + BC$ [Since $B + \bar{B}C = B + C$]

   $= \bar{A}B + \bar{A}C + BC$

3. F = $\bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC$

   $= \bar{A}\bar{B}(\bar{C} + C) + \bar{A}BC + AC(\bar{B} + B)$

   $= \bar{A}\bar{B} + \bar{A}BC + AC$

   $= \bar{A}(\bar{B} + BC) + AC$

   $= \bar{A}(\bar{B} + C) + AC$ [Since $\bar{B} + BC = \bar{B} + C$]

   $= \bar{A}\bar{B} + \bar{A}C + AC$

   $= \bar{A}\bar{B} + C(\bar{A} + A)$

   $= \bar{A}\,\bar{B} + C$ [Since $\bar{A} + A = 1$]

4. F = $\bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C}$

$= \bar{A}\bar{B}(\bar{C} + C) + B\bar{C}(\bar{A} + A) + A\bar{B}(\bar{C} + C)$

$= \bar{A}\bar{B} + B\bar{C} + A\bar{B}$

$= \bar{B}(\bar{A} + A) + B\bar{C}$

$= \bar{B} + B\bar{C}$ [Since $\bar{B} + B\bar{C} = \bar{B} + \bar{C}$]

$= \bar{B} + \bar{C}$

**Simplification of Boolean Expressions – Karnaugh Map Method**

The Karnaugh Map is a method of simplifying Boolean expressions without using Boolean laws, so that they can be implemented using a minimum number of logic gates.

**Structure of the Karnaugh Map:**

The structure of Karnaugh Map depends on the number of variables in a given Boolean expression. Depending on the number of variables present in the Boolean expression, the Karnaugh Maps are classified as:

1. Two-variable Karnaugh Map
2. Three-variable Karnaugh Map
3. Four-variable Karnaugh Map

*Two-variable Karnaugh Map:*

A two-variable Karnaugh Map is used for simplifying Boolean expressions containing two different variables. The structure of a two-variable Karnaugh map is shown below.

| | $\bar{B}$ | $B$ |
|---|---|---|
| $\bar{A}$ | | |
| $A$ | | |

*Three-variable Karnaugh Map:*

A three-variable Karnaugh Map is used for simplifying Boolean expressions containing three different variables. The structure of a three-variable Karnaugh map is shown below.

| | $\bar{B}\bar{C}$ | $\bar{B}C$ | $BC$ | $B\bar{C}$ |
|---|---|---|---|---|
| $\bar{A}$ | | | | |
| $A$ | | | | |

*Four-variable Karnaugh Map:*

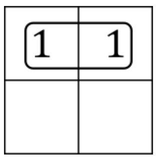
A four-variable Karnaugh Map is used for simplifying Boolean expressions containing four different variables. The structure of a four-variable Karnaugh map is shown below.

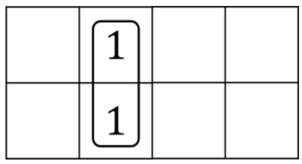
| | $\bar{C}\bar{D}$ | $\bar{C}D$ | $CD$ | $C\bar{D}$ |
|---|---|---|---|---|
| $\bar{A}\bar{B}$ | | | | |
| $\bar{A}B$ | | | | |
| $AB$ | | | | |
| $A\bar{B}$ | | | | |

**Grouping of Minterms (Pair, Quad, Octet):**

*Pair* : A pair is a group of two adjacent 1's or 0's.  A pair will eliminate one variable.

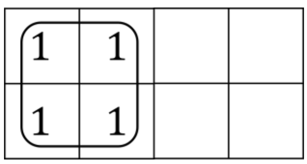
Examples:

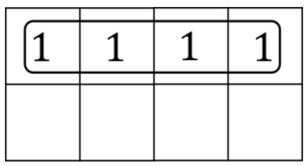




*Quad* : A quad is a group of four adjacent 1's or 0's.  A quad will eliminate two variables.

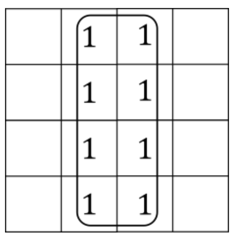
Examples:

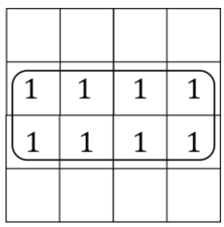




*Octet* : An octet is a group of four adjacent 1's or 0's.  An octet will eliminate three variables.

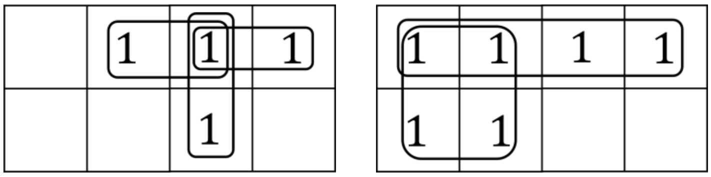
Examples:





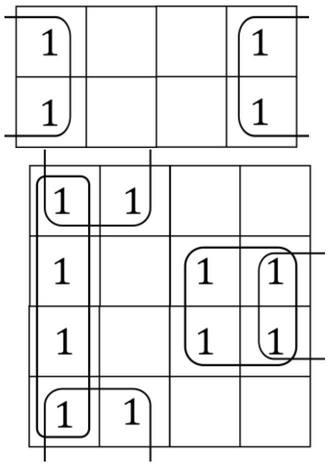
*Overlapping groups:*

Examples



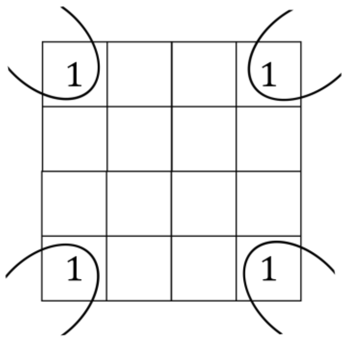
*Folding the map:*

Examples



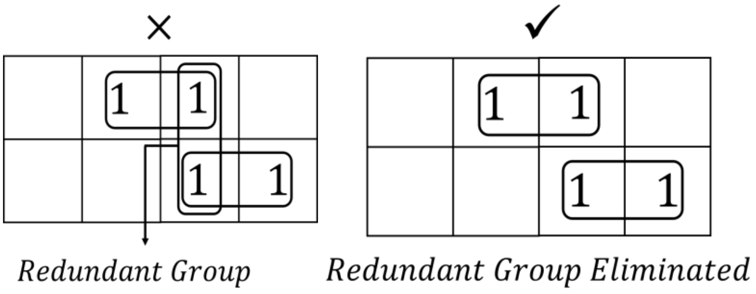
*Double folding:*

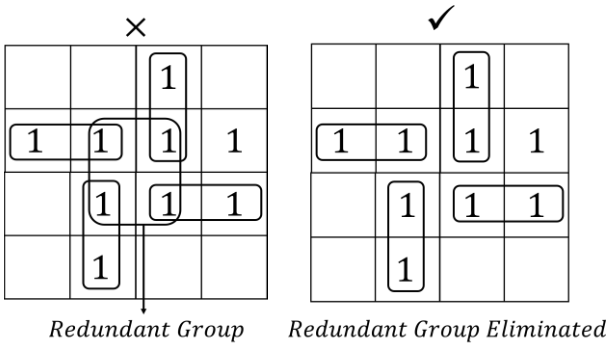
Example:



*Redundant groups:* A group is said to be redundant if all its 1's or 0's are used by other groups. Redundant groups, if any, should be eliminated.

Examples




**Simplification Problems using Karnaugh Map: (Refer class notes)**

Karnaugh maps can be used to simplify Boolean expressions in both SOP and POS forms.

*Simplification of Boolean expressions in SOP form:*

1. Fill up the Karnaugh map with 1's and 0's according to the given expression,
2. Consider 1's and group them into all possible groups.
3. Write the simplified SOP form of expression from the map.
4. Draw AND-OR circuit for the simplified expression.
5. Convert the above circuit into a NAND-NAND circuit by replacing all the gates in the circuit with NAND gates.

*Simplification of Boolean expressions in POS form:*

1. Fill up the Karnaugh map with 1's and 0's according to the given expression,
2. Consider 0's and group them into all possible groups.
3. Write the expression from the map.
4. Complement the above expression to obtain the simplified POS form of expression.
5. Draw OR-AND circuit for the simplified expression.
6. Convert the above circuit into a NOR-NOR circuit by replacing all the gates in the circuit with NOR gates.

# UNIT-3

## COMBINATIONAL CIRCUITS

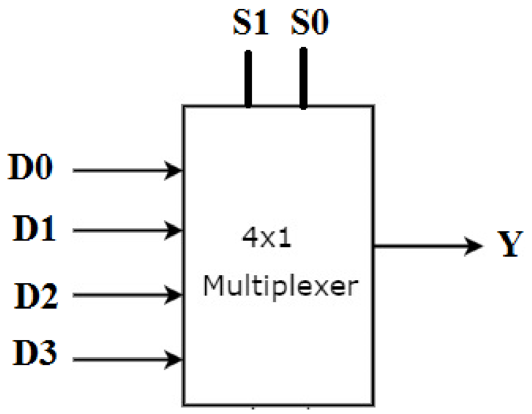
## MULTIPLEXER

Multiplex means many-to-one. A multiplexer is a combinational circuit that has maximum of $2^n$ data inputs, ‘n’ selection (control) lines and one output line. One of these data inputs will be connected to the output based on the values of selection lines.

Since there are ‘n’ selection lines, there will be $2^n$ possible combinations of 0s and 1s. So, each combination will select only one data input. Multiplexer is also referred to as MUX.

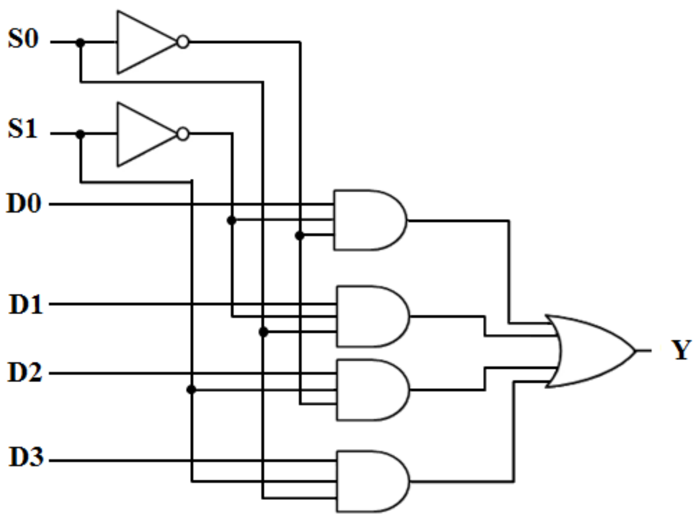
### 4x1 Multiplexer

4x1 Multiplexer has four data inputs D0, D1, D2, D3, two selection lines S1 and S0 and one output Y. The block diagram of a 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.

**Internal Diagram of 4x1 Multiplexer:**



The truth table of 4x1 Multiplexer is shown below.

| Selection Inputs | | Output |
|---|---|---|
| **S1** | **S0** | **Y** |
| 0 | 0 | $D_0$ |
| 0 | 1 | $D_1$ |
| 1 | 0 | $D_2$ |
| 1 | 1 | $D_3$ |

## DEMULTIPLEXER

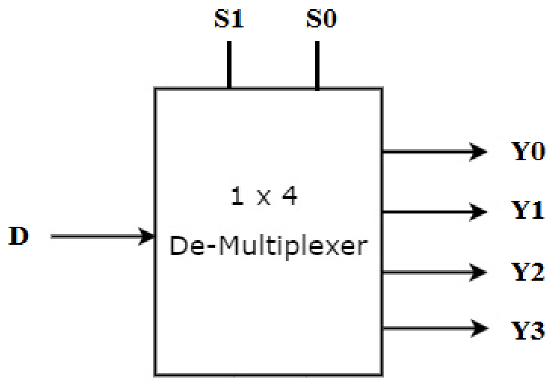
Demultiplex means one-to-many. A De-Multiplexer is a combinational circuit that performs the reverse operation of a Multiplexer. It has single input, 'n' selection lines and maximum of $2^n$ outputs. The input will be connected to one of these outputs based on the values of selection lines.

Since there are 'n' selection lines, there will be $2^n$ possible combinations of 0s and 1s. So, each combination can select only one output. De-Multiplexer is also referred to as DEMUX.

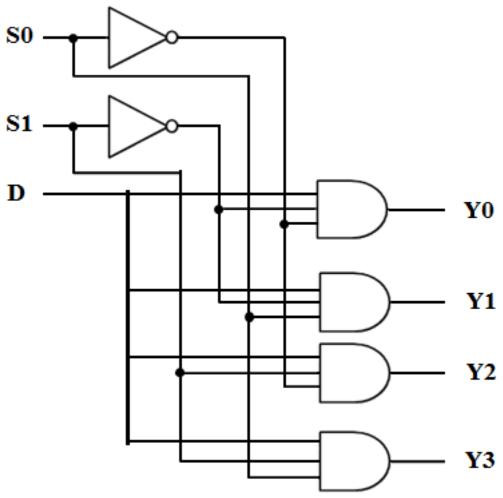
### 1x4 De-Multiplexer

1x4 De-Multiplexer has one input I, two selection lines, S1 and S0 and four outputs $Y_3$, $Y_2$, $Y_1$ & $Y_0$.

The block diagram of 1x4 De-Multiplexer is shown in the following figure.




The single input **'D'** will be connected to one of the four outputs, **$Y_3$ to $Y_0$** based on the values of selection lines **S1** and **S0**. The **circuit diagram** of 1x4 De-Multiplexer is shown in the following figure.




The Truth table of 1x4 De-Multiplexer is shown below.

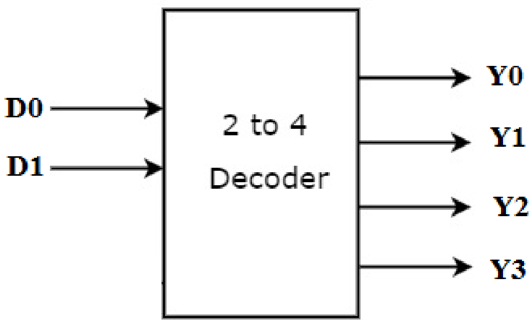
| Selection Inputs | | Outputs | | | |
|---|---|---|---|---|---|
| $S_1$ | $S_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
| 0 | 0 | 0 | 0 | 0 | **D** |
| 0 | 1 | 0 | 0 | **D** | 0 |
| 1 | 0 | 0 | **D** | 0 | 0 |
| 1 | 1 | **D** | 0 | 0 | 0 |

**DECODER**

A Decoder is a combinational circuit that has 'n' input lines and $2^n$ output lines. One of these outputs will be active High (enabled) based on the combination of inputs present.

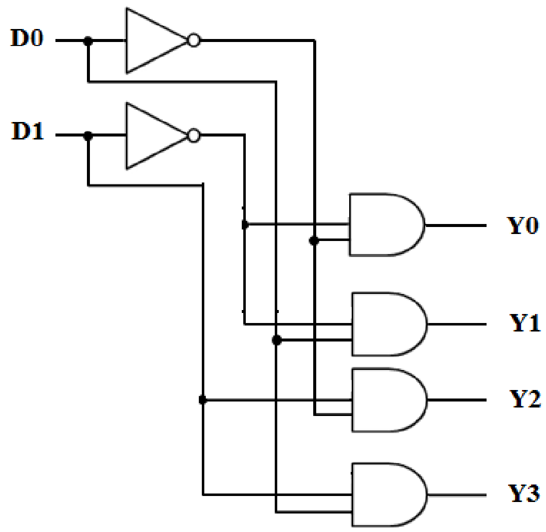
**2-to-4 Decoder:**

A 2-to-4 Decoder has two inputs $D_1$ & $D_0$ and four outputs $Y_3$, $Y_2$, $Y_1$ & $Y_0$. The block diagram of a 2-to-4 decoder is shown in the following figure.



One of these four outputs will be '1' for each combination of inputs.

The circuit diagram of 2-to-4 decoder is shown in the following figure.




The Truth table of 2 to 4 decoder is shown below.

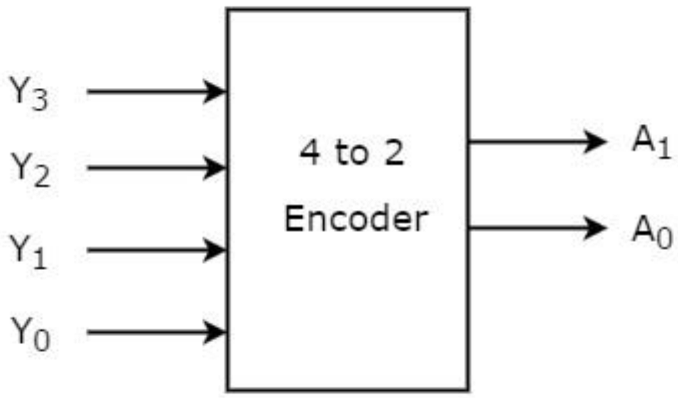
| Inputs | | Outputs | | | |
|---|---|---|---|---|---|
| $D_1$ | $D_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
| 0 | 0 | 0 | 0 | 0 | **1** |
| 0 | 1 | 0 | 0 | **1** | 0 |
| 1 | 0 | 0 | **1** | 0 | 0 |
| 1 | 1 | **1** | 0 | 0 | 0 |

## ENCODER

An Encoder is a combinational circuit that performs the reverse operation of Decoder. It has maximum of $2^n$ input lines and 'n' output lines. 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. It is optional to represent the enable signal in encoders.

### 4 to 2 Encoder:

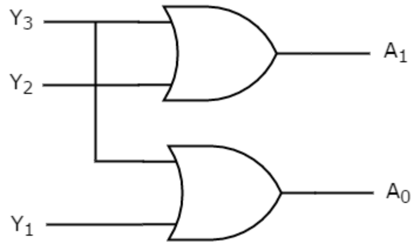
Let 4 to 2 Encoder has four inputs $Y_3$, $Y_2$, $Y_1$ & $Y_0$ and two outputs $A_1$ & $A_0$. The block diagram of 4 to 2 Encoder is shown in the following figure.




At any time, only one of these 4 inputs can be '1' in order to get the respective binary code at the output. The Truth table of 4 to 2 encoder is shown below.

| **Inputs** | | | | **Outputs** | |
|---|---|---|---|---|---|
| **$Y_3$** | **$Y_2$** | **$Y_1$** | **$Y_0$** | **$A_1$** | **$A_0$** |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |

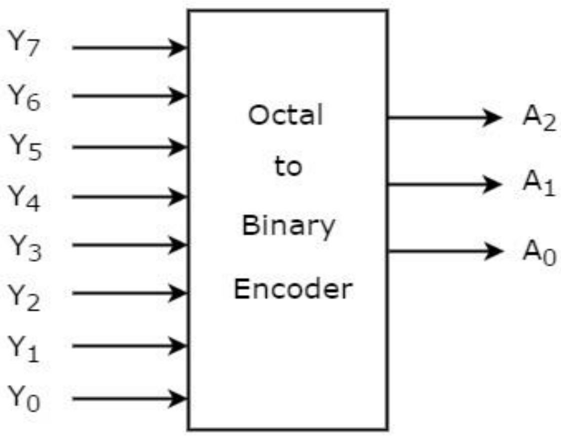
We can implement the above two Boolean functions by using two input OR gates. The circuit diagram of 4 to 2 encoder is shown in the following figure.




The above circuit diagram contains two OR gates. These OR gates encode the four inputs with two bits

**Octal to Binary Encoder (Octal encoder):**

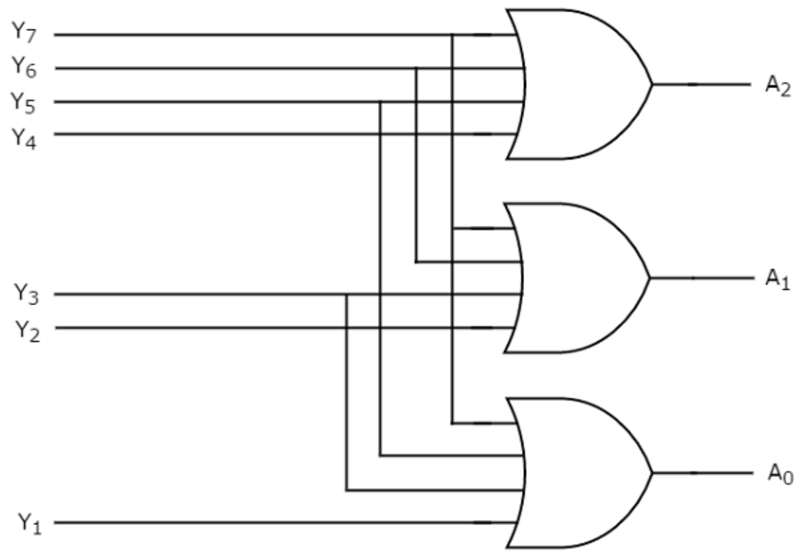
Octal to binary Encoder has eight inputs, $Y_7$ to $Y_0$ and three outputs $A_2$, $A_1$ & $A_0$. Octal to binary encoder is nothing but 8 to 3 encoder. The block diagram of octal to binary Encoder is shown in the following figure.



At any time, only one of these eight inputs can be '1' in order to get the respective binary code. The Truth table of octal to binary encoder is shown below.

| **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 |

We can implement the above Boolean functions by using four input OR gates. The circuit diagram of octal to binary encoder is shown in the following figure.




The above circuit diagram contains three 4-input OR gates. These OR gates encode the eight inputs with three bits.

## ARITHMETIC CIRCUITS

### HALF ADDER

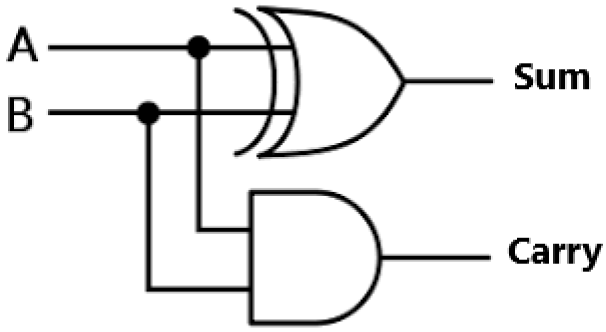
A half adder is a combinational circuit that generates the arithmetic sum of two bits at time. The circuit has two inputs and two outputs (Sum and Carry). The **truth table** of a half adder is given below:

| Inputs | | Outputs | |
|---|---|---|---|
| **A** | **B** | **Sum** | **Carry** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

The Carry output is 0 unless both inputs are 1. The simplified Boolean expressions for the two outputs Sum and Carry can be obtained directly from the truth table. The expressions are:

Sum = A'B + AB' (XOR gate)
Carry = AB (AND gate)

From the above expressions we find that the Sum output represents the output of a 2-input EXOR gate and the Carry output represents the output of an AND gate. The circuit of a half adder can be constructed using a 2-input EXOR gate and an AND gate as shown below.




**FULL ADDER**

A full adder is a combinational circuit that generates the arithmetic sum of three bits at time. The circuit has three inputs and two outputs (Sum and Carry). The truth table of a full adder is given below:

| 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 |

The simplified Boolean expressions for the two outputs Sum and Carry can be obtained directly from the truth table. The expressions are:

Sum = A’B’C + A’BC’ + AB’C’ + ABC
Carry = A’BC + AB’C + ABC’ + ABC

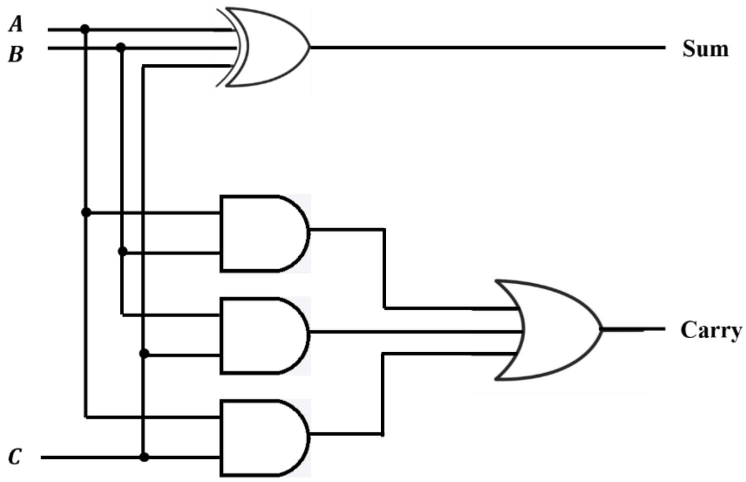
From the above expressions we find that the Sum output represents the output of a 3-input EXOR gate. The simplified expression for Carry output can be derived using a Karnaugh map as shown below:

| | B’C’ | B’C | BC | BC’ |
|---|---|---|---|---|
| A’ | 0 | 0 | 1 | 0 |
| A | 0 | 1 | 1 | 1 |

The simplified expression for Carry output is obtained as:

Carry = AB + BC + AC

The circuit diagram of a full adder can now be constructed using a 3-input EXOR gate (for Sum output) and a combinational circuit for the expression AB + BC + AC (for Carry output) as shown below:




**HALF SUBTRACTOR**

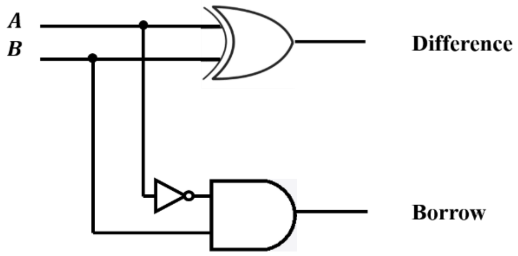
A half subtractor is a combinational circuit that finds the arithmetic difference between two bits at a time. The circuit has two inputs and two outputs (Difference and Borrow). The truth table of a half subtractor is given below:

| **Inputs** | | **Outputs** | |
|---|---|---|---|
| **A** | **B** | **Difference** | **Borrow** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

For 0 – 1, a borrow of 1 is required to get a difference of 1. The simplified Boolean expressions for the two outputs Difference and Borrow can be obtained directly from the truth table. The expressions are:

Difference = A’B + AB’
Borrow = A’B

From the above expressions we find that the Difference output represents the output of a 2-input EXOR gate and Borrow output represents A’B. The circuit of a half subtractor can be constructed using a 2-input EXOR gate and a combinational circuit for A’B, as shown below.




**FULL SUBTRACTOR**

A full subtractor is a combinational circuit that finds the arithmetic difference between three bits at time. The circuit has three inputs and two outputs (Difference and Borrow). The truth table of a full subtractor is given below:

| **Inputs** | | | **Outputs** | |
|---|---|---|---|---|
| **A** | **B** | **C** | **Difference** | **Borrow** |
| 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 |

The simplified Boolean expressions for the two outputs Difference and Borrow can be obtained directly from the truth table. The expressions are:

Difference = A’B’C + A’BC’ + AB’C’ + ABC
Borrow = A’B’C + A’BC’ + A’BC + ABC

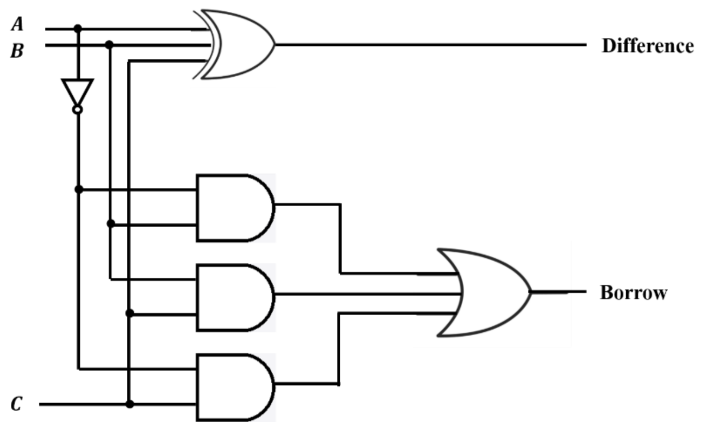
From the above expressions we find that the Difference output represents the output of a 3-input EXOR gate. The simplified expression for Borrow output can be derived using a Karnaugh map as shown below:

| | B’C’ | B’C | BC | BC’ |
|---|---|---|---|---|
| A’ | 0 | 1 | 1 | 1 |
| A | 0 | 0 | 1 | 0 |

The simplified expression for Borrow output is obtained as:

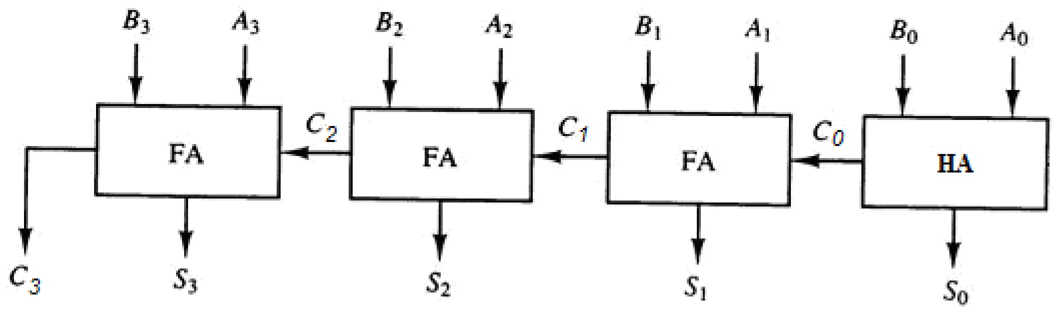
Borrow = A’B + BC + A’C

The circuit diagram of a full subtractor can now be constructed using a 3-input EXOR gate (for Difference output) and a combinational circuit for the expression A’B + BC + A’C (for Borrow output) as shown below:




**PARALLEL BINARY ADDER**

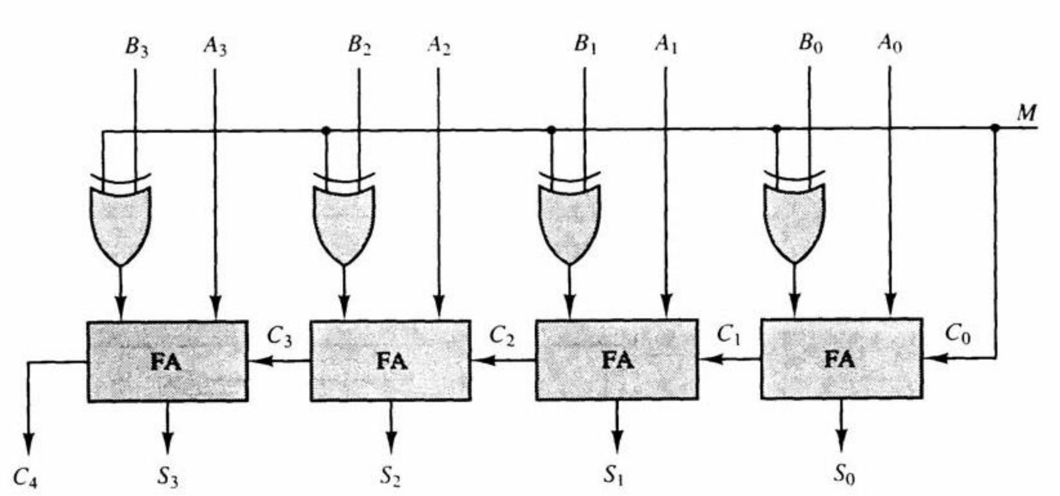
The digital circuit that generates the arithmetic sum of two binary numbers is called a parallel binary adder. The circuit diagram of a 4-bit parallel binary adder is shown in the figure below.




The two binary numbers are designated as $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$. The sum bits generated by the circuit are designated as $S_3S_2S_1S_0$. For the addition of the bits $A_0$ and $B_0$ we require a half-adder. This addition generates a carry bit ($C_0$) to the next position thereby requiring a full-adder to carry out this addition of three bits. Hence for all subsequent additions full-adders are required. The final carry generated by the parallel binary adder is $C_3$. An *n*-bit parallel binary adder requires 1 half-adder and *n-1* full-adders.

**2’S COMPLEMENT ADDER-SUBTRACTOR**

A 2’s complement adder-subtractor is a combinational circuit that performs both addition and subtraction of two binary words. The subtraction can also be performed by means of 2’s complement addition. The subtraction A – B can be carried out by taking the 2’s complement of B and adding it to A. The 2’s complement can be obtained by taking the 1’s complement and adding a 1 to the least significant pair of bits. The following figure shows the circuit diagram of a 4-bit adder-subtractor.




The circuit includes an exclusive-OR gate with each full adder. The exclusive-OR gate acts as a controlled inverter that helps in obtaining the 1's complement of a binary number for subtraction. The mode input M controls the operation. When M = 0 the circuit performs as an adder and when M = 1 the circuit performs as a subtractor. Each exclusive-OR gate receives input M an one of the inputs of B. When M = 0, we have $B \oplus 0 = B$. The full-adders receive the value of B, the input carry is 0, and the circuit performs A plus B (Addition). when M = 1, we have $B \oplus 0 = B'$. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's Complement of B (Subtraction).

## UNIT-4

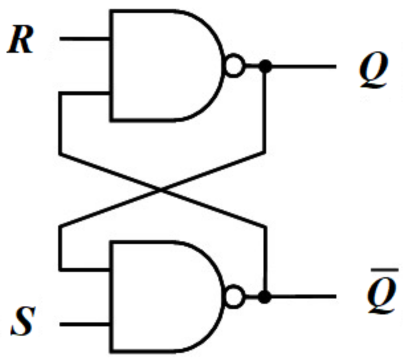
# SEQUENTIAL CIRCUITS

## FLIP-FLOPS

1. Flipflop is a sequential circuit used to store a binary digit (BIT)
2. One flipflop is required to store one bit (0 or 1)
3. A flipflop may have one or two inputs and two outputs. One output is complement of the other.
4. A group of flipflops used to store a binary word is called a register.
5. A flipflop is a bistable memory element that stores 0 or 1.

### R-S Flip-flop Element (Latch):

The R-S flip-flop can be realized using NAND gates. The circuit diagram and the truth table of R-S flip-flop are shown below.

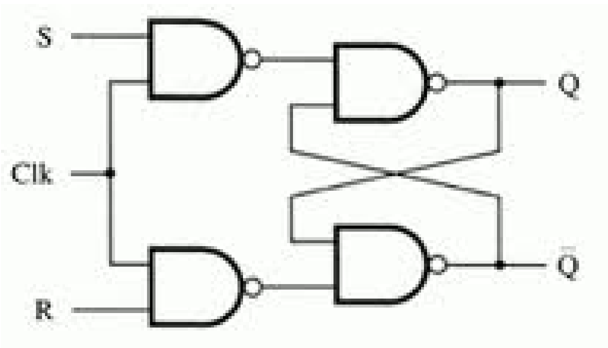



| R | S | Q | |
|---|---|---|---|
| 0 | 0 | * | *Forbidden State* |
| 0 | 1 | 1 | *Set State* |
| 1 | 0 | 0 | *Reset State* |
| 1 | 1 | NC | *No Change State* |

A 0 on any input to NAND gate will make its output high. Therefore, when R=0 and S=0, the output of the flip-flop becomes unpredictable. This state is sometimes referred to as a *forbidden state* or *race condition*. When R=0 and S=1, the output Q of the flip-flop is set to 1. This state is called *set* state. When R=1 and S=0, the output Q of the flip-flop is reset to 0. This state is called *reset* state. When R=1 and S=1, the output Q of the flip-flop will remain in its previous state. In other words, the output of the flip-flop will not change.

### Clocked R-S or S-R flip-flop:

The circuit diagram and the truth table of a S-R flip-flop are shown below.



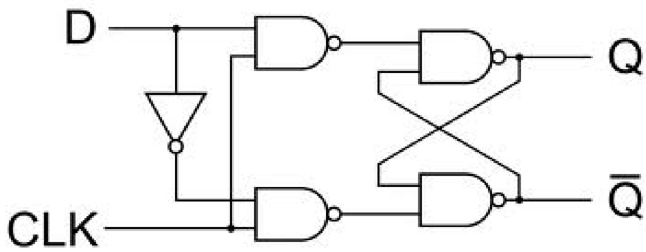
| CLK | S | R | Q | |
|---|---|---|---|---|
| 0 | X | X | NC | *No Change State* |
| 1 | 0 | 0 | NC | *No Change State* |
| 1 | 0 | 1 | 0 | *Reset State* |
| 1 | 1 | 0 | 1 | *Set State* |
| 1 | 1 | 1 | * | *Forbidden State* |

When the CLK input is 0, the output of the flip-flop will not change irrespective of the inputs at S and R. The flip-flop is disabled when the clock is 0. When the CLK input is 1, the output of the flip-flop will change according to the S and R inputs. The flip-flop is enabled when the clock is 1. The response of the flip-flop for various S and R inputs, when the CLK is 1, is as follows.

When R=0 and S=0, the output of the flip-flop will not change. When R=0 and S=1, the output Q of the flip-flop is set to 1. When R=1 and S=0, the output Q of the flip-flop is reset to 0. When R=1 and S=1, the output Q of the flip-flop becomes unpredictable.

**D flip-flop:**

The circuit diagram and the truth table of a D flip-flop are shown below.

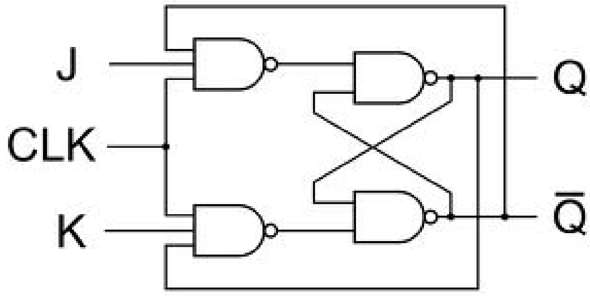



| CLK | D | Q | |
|---|---|---|---|
| 0 | X | NC | *No Change State* |
| 1 | 0 | 0 | *Reset State* |
| 1 | 1 | 1 | *Set State* |

When the CLK input is 0, the output of the flip-flop will not change irrespective of the input D. The flip-flop is disabled when the clock is 0. When the CLK input is 1, the output of the flip-flop will change according to the D input. The flip-flop is enabled when the clock is 1. The response of the flip-flop for various D input, when the CLK is 1, is as follows. When D=0, the output Q of the flip-flop become 0 (reset). When D=1, the output Q of the flip-flop become 1 (set). There is no forbidden or unpredictable state in a D flip-flop. A D flip-flop is sometimes referred to as a *data* flip-flop.

**J-K flip-flop:**

The circuit diagram and the truth table of a J-K flip-flop are shown below.



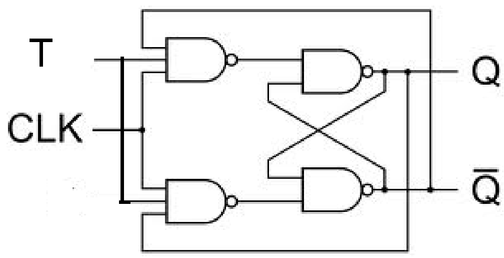

| CLK | J | K | Q | |
|---|---|---|---|---|
| 0 | X | X | NC | *No Change State* |
| 1 | 0 | 0 | NC | *No Change State* |
| 1 | 0 | 1 | 0 | *Reset State* |
| 1 | 1 | 0 | 1 | *Set State* |
| 1 | 1 | 1 | Q’ | *Toggle State* |

When the CLK input is 0, the output of the flip-flop will not change irrespective of the inputs at J and K. The flip-flop is disabled when the clock is 0. When the CLK input is 1, the output of the flip-flop will change according to the J and K inputs. The flip-flop is enabled when the clock is 1. The response of the flip-flop for various J and K inputs, when the CLK is 1, is as follows.

When J=0 and K=0, the output of the flip-flop will not change. When J=0 and K=1, the output Q of the flip-flop is set to 1. When J=1 and K=0, the output Q of the flip-flop is reset to 0. When J=1 and K=1, the clock pulse switches the output of the flip-flop to its *complement* state. This state is also referred to as *toggle* state. There is no forbidden or unpredictable state in a D flip-flop.

**T flip-flop:**

The circuit diagram and the truth table of a T flip-flop are shown below.



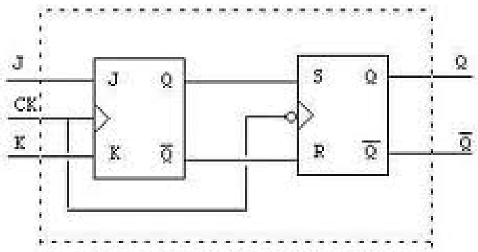
| CLK | T | Q | |
|---|---|---|---|
| 0 | X | NC | *No Change State* |
| 1 | 0 | NC | *No Change State* |
| 1 | 1 | Q' | *Toggle State* |

A T flip-flop can be realized from a J-K flip-flop when inputs J and K are combined to provide a single input designated as T. When the CLK input is 0, the output of the flip-flop will not change irrespective of the input T. The flip-flop is disabled when the clock is 0. When the CLK input is 1, the output of the flip-flop will change according to the T input. The flip-flop is enabled when the clock is 1.

The response of the flip-flop for various T input, when the CLK is 1, is as follows. When T=0, the output Q of the flip-flop will remain unchanged When T=1, the clock pulse switches the output of the flip-flop to its *complement* (toggle) state. A T flip-flop is sometimes referred to as a *toggle* flip-flop.

**J-K Master-Slave Flip-flop:**

The circuit diagram of a J-K Master-Slave flip-flop is shown below.




The circuit consists of two J-K flip-flops of which one is called the master and other the slave. The master flip-flop is positive-edge-triggered and the slave is negative-edge triggered. Therefore, the master responds to its J and K inputs before the slave. The master flip-flop changes state during the positive clock pulse and the slave changes state during the successive negative clock pulse.

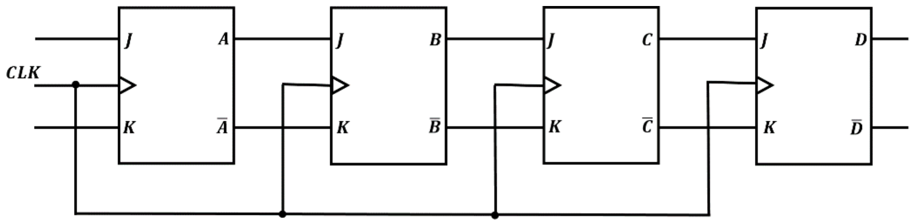
If J = 1 and K = 0, the master sets on the positive clock edge. The high Q output of the master drives the J input of the slave. So, when the negative clock edge hits, the slave sets, copying the action of the master. If J = 0 and K = 1, the master resets on the positive clock edge. The high Q' output of the master drives the K input of the slave. Hence, the arrival of the negative clock edge forces the slave to reset. If the J and K inputs of the master are both high, it toggles on the positive clock edge and the slave then toggles on the negative clock edge. Regardless of what the master does, therefore, the slave copies it; if the master sets, the slave sets; if the master resets, the slave resets.

**SHIFT REGISTER**

A register is a group of flip-flops that can be used to store a binary word. There must be one flip-flop for each bit in the binary word. For instance, a register used to store an 8-bit word must have eight flip-flops. A register capable of shifting its binary information in one or both directions is called a shift register. The logical organization of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive common clock pulses that initiate the shift from one stage to the next. The shift register that shifts bits towards right is called a shift right register and that which shifts towards left is called a shift left register.

**Shift Right Register:**

The circuit diagram of a 4-bit shift right register, constructed using J-K flip-flops, is shown in the figure below.




For a J-K flip-flop, the data bit to be shifted into the flip-flop must be present at the J and K inputs when the clock strikes. To shift a 0 into the flip-flop, J = 0 and K = 1. To shift a 1 into the flip-flop, J = 1 and K = 0. The circuit consists of four J-K flip-flops, whose outputs are designated as A, B, C, and D. The data bits to be shifted are given at the J and K inputs of flip-flop A. The outputs of flip-flop A is connected to the inputs of flip-flop B. The outputs of flip-flop B is connected to the inputs of flip-flop C and so on. The clock pulse is connected to all the flip-flops as shown in the figure.

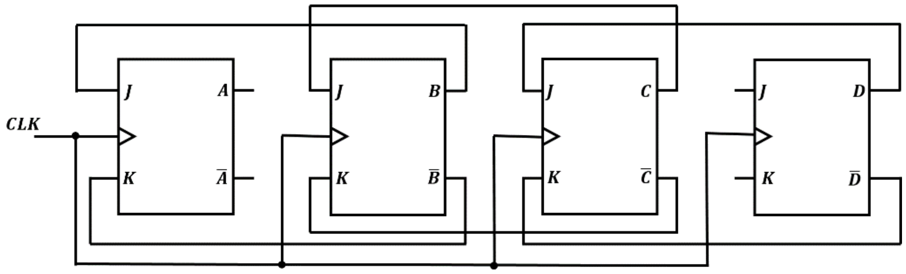
Consider the data bits at the inputs of flip-flop A as J = 1, and K = 0. As the clock pulses are applied to all the flip-flops directly, each flip-flop changes state depending on its J and K inputs at the time of arrival of the clock pulse. When the first clock pulse strikes, the 0 in C is shifted into D, the 0 in B is shifted into C, the 0 in A is shifted into B and the data input 1 is shifted into A. At this point of time the output ABCD = 1000. When the second clock pulse strikes, the 0 in C is shifted into D, the 0 in B is shifted into C, the 1 in A is shifted into B and the data input 1 is shifted into A. The output at this stage is ABCD = 1100. The shifting continues in this order for every strike of the clock pulse and at the end of the fourth clock pulse the outputs of the flip-flops would be ABCD = 1111.

Similarly, consider the data bits at the inputs of flip-flop A as J = 0, and K = 1. When the first clock pulse strikes, the 1 in C is shifted into D, the 1 in B is shifted into C, the 1 in A is shifted into B and the data input 0 is shifted into A. At this point of time the output ABCD = 0111. When the second clock pulse strikes, the 1 in C is shifted into D, the 1 in B is shifted into C, the 0 in A is shifted into B and the data input 0 is shifted into A. The output at this stage is ABCD = 0011. The shifting continues in this order for every strike of the clock pulse and at the end of the fourth clock pulse the outputs of the flip-flops would be ABCD = 0000. This is illustrated in the table given below.

| | **CLK** | **A** | **B** | **C** | **D** |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| J=1, K=0 | 1 | 1 | 0 | 0 | 0 |
| | 2 | 1 | 1 | 0 | 0 |
| | 3 | 1 | 1 | 1 | 0 |
| | 4 | 1 | 1 | 1 | 1 |
| J=0, K=1 | 5 | 0 | 1 | 1 | 1 |
| | 6 | 0 | 0 | 1 | 1 |
| | 7 | 0 | 0 | 0 | 1 |
| | 8 | 0 | 0 | 0 | 0 |

**Shift Left Register:**

The circuit diagram of a 4-bit shift left register, constructed using J-K flip-flops, is shown in the figure below:




The circuit consists of four J-K flip-flops, whose outputs are designated as A, B, C, and D. The data bits to be shifted are given at the J and K inputs of flip-flop D. The outputs of flip-flop D is connected to the inputs of flip-flop C. The outputs of flip-flop C is connected to the inputs of flip-flop B and so on. The clock pulse is connected to all the flip-flops as shown in the figure.

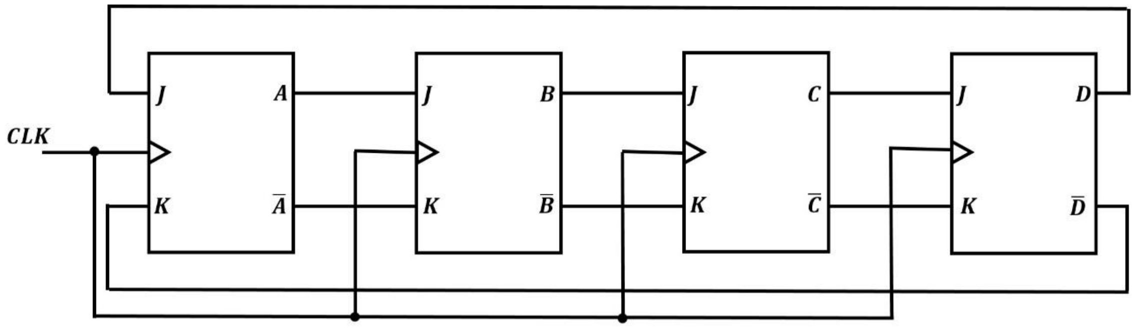
Consider the data bits at the inputs of flip-flop D as J = 1, and K = 0. As the clock pulses are applied to all the flip-flops directly, each flip-flop changes state depending on its J and K inputs at the time of arrival of the clock pulse. When the first clock pulse strikes, the data input 1 is shifted into D, the 0 in D is shifted into C, the 0 in C is shifted into B and the 0 in B is shifted into A. At this point of time the output ABCD = 0001. When the second clock pulse strikes, the 1 in D is shifted into C, the 0 in C is shifted into B, and the 0 in B is shifted into A. The output at this stage is ABCD = 0011. The shifting continues in this order for every strike of the clock pulse and at the end of the fourth clock pulse the outputs of the flip-flops would be ABCD = 1111.

Similarly consider the data bits at the inputs of the flip-flop D as J = 0 and K = 1. When the first clock pulse strikes, the data input 0 is shifted into D, the 1 in D is shifted into C, the 1 in C is shifted into B and the 1 in B is shifted into A. At this point of time the output ABCD = 1110. When the second clock pulse strikes, the 0 in D is shifted into C, the 1 in C is shifted into B, and the 1 in B is shifted into A. The output at this stage is ABCD = 1100. The shifting continues in this order for every strike of the clock pulse and at the end of the fourth clock pulse the outputs of the flip-flops would be ABCD = 0000. This is illustrated in the table given below.

| | **CLK** | **A** | **B** | **C** | **D** |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 |
| J=1, K=0 | 1 | 0 | 0 | 0 | 1 |
| | 2 | 0 | 0 | 1 | 1 |
| | 3 | 0 | 1 | 1 | 1 |
| | 4 | 1 | 1 | 1 | 1 |
| J=0, K=1 | 5 | 1 | 1 | 1 | 0 |
| | 6 | 1 | 1 | 0 | 0 |
| | 7 | 1 | 0 | 0 | 0 |
| | 8 | 0 | 0 | 0 | 0 |

## RING COUNTER

A ring counter is constructed by connecting the outputs of the last flip-flop of the shift register to the inputs of the first flip-flop. The circuit diagram of a 4-bit ring counter is shown in the figure below.




The outputs of the flip-flop A is connected to the inputs of flip-flop B. The outputs of the flip-flop B is connected to the inputs of flip-flop C. The outputs of the flip-flop C is connected to the inputs of flip-flop D. The outputs of the flip-flop D is fed back and connected to the inputs of flip-flop A. The clock pulse is connected to all the flip-flops directly. When the clock pulse is allowed to run, the outputs of all the flip-flops will never change, as long as the low output of flip-flop D is connected to the input of A.

Suppose that, by some external means, flip-flop A is made high, and all other flip-flops are low, and then the clock is allowed to run. During the first clock pulse, the 1 in flip-flop A will shift into flip-flop B and flip-flop A will be reset (0), since the 0 in flip-flop D will shift into flip-flop A. All other flip-flops will still contain 0s. The second clock pulse will shift the 1 in flip-flop B into flip-flop C, while B resets. The third clock pulse will shift the 1 in flip-flop C into flip-flop D, and so on. Thus the single 1 will shift down the register, traveling from one flip-flop to the next flip-flop for each clock pulse. When it reaches flip-flop D, the next clock will shift it into flip-flop A by means of the feedback connection. This configuration is also referred to as a circulating register. The following table shows the state of the ring counter for each clock pulse.

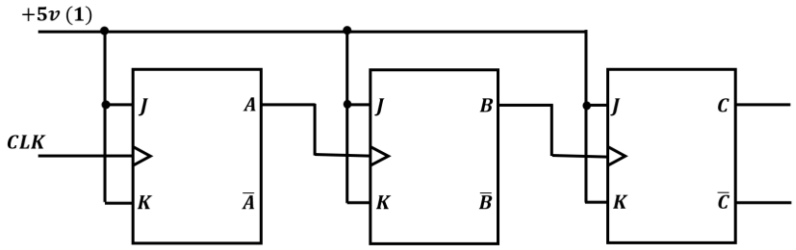
| CLK | A | B | C | D |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 |

## COUNTERS

A register that goes through a prescribed sequence of states upon the application of clock pulses is called a counter. The sequence of states may follow the binary number sequence or any other sequence of states. A counter that follows the binary number sequence is called a binary counter. An n-bit counter consists of n flip-flops and can count in binary from 0 through $2^n - 1$. There are basically two types of counters – asynchronous and synchronous counters.

**Asynchronous Counter:**

In an asynchronous counter, each flip-flop is triggered by the previous flip-flop. In other words, the output of a flip-flop is used as the clock input for the next flip-flop. Such a counter is also called a ripple counter or a serial counter. A 3-bit binary ripple counter can be constructed using three JK flip-flops as shown in the figure below:




The system clock drives flip-flop A. The output of flip-flop A drives flip-flop B, and the output of flip-flop B drives flip-flop C. All the J and K inputs are connected to $+V_{cc}$. This means each flip-flop will change state with a negative transition of the clock input.

| CLK | C | B | A |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 |
| 8 | 0 | 0 | 0 |

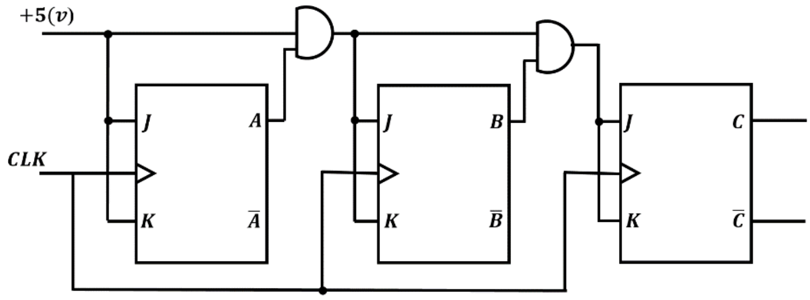
To understand the operation of this counter, refer to the functional table given above. The count starts with 000 and increments by one with each clock pulse input. After the count 111, the counter goes back to 000 to repeat the count. The least significant bit A is complemented with each clock pulse input. Every time A goes from 1 to 0, it complements B. Every time B goes from 1 to 0, it complements C, and so on for any other higher order bits of a counter. During the first clock pulse, flip-flop A changes from 0 to 1. This is a positive change and hence flip-flops B and C will not change state. The contents of the counter at this stage is CBA=001. During the second clock pulse, flip-flop A changes from 1 to 0. This negative transition drives flip-flop B and hence it changes from 0 to 1. This positive transition in B does not affect flip-flop C and hence the contents of the counter is CBA= 010. During the third clock pulse, flip-flop A changes from 0 to 1. This positive transition does not affect the other flip-flops. Hence the state of the counter changes to CBA = 011. The counter advances one count for each clock pulse until it reaches the count 111. At this point it resets back to 000 and begins the count cycle all over again.

A counter with three flip-flops is often referred to as a modulus-8 (or mod-8) counter since it counts 8 states. Similarly, a counter with four flip-flops is called a mod-16 counter, and so on. The modulus of a counter is the total number of states through which the counter can progress.

**Synchronous Counter:**

Synchronous counters are different from ripple counters in that clock pulses are applied to all the flip-flops directly. A common clock triggers all the flip-flops simultaneously. The decision whether a flip-flop is to be complemented or not is determined from the values of the data inputs J and K at the time of the clock pulse. If J = K = 0, the flip-flop does not change state. It J = K = 1, the flip-flop toggles.

In a synchronous binary counter, the flip-flop in the least significant position is complemented with every clock pulse.  A flip-flop in any other position is complemented when all the bits in the lower significant positions are equal to 1.  The circuit diagram and the functional table of a 3-bit synchronous counter is shown in the figure below.




| CLK | C | B | A |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 |
| 8 | 0 | 0 | 0 |

The clock inputs of all the flip-flops are connected to a common clock. The counter is enabled with the count enable input ($+V_{cc}$). The first stage flip-flop A has its J and K inputs equal to 1. The J and K inputs of other flip-flops are equal to 1 if all previous least significant stages are equal to 1 and the count is enabled.  The chain of AND gates generates the required logic for the J and K inputs in each stage. For example, if the present state of the counter is CBA = 0011, the next count is 100. A is always complemented. B is complemented because the present state of A = 1.  C is complemented because the present state of BA = 11. The counter can be extended to any number of stages, with each stage having an additional flip-flop and an AND gate that gives and output of 1 if all previous flip-flop outputs are 1.

# UNIT-5

## REGISTER TRANSFER AND MICROOPERATIONS

### Register Transfer Language

The operations executed on data stored in registers are called *microoperations*. A microoperation is an elementary operation performed on the information stored in one or more registers. The result of the operation may replace the previous binary information of a register or may be transferred to another register. Examples of microoperations are shift, count, clear, and load. The internal hardware organization of a digital computer is best defined by specifying:

- *The set of registers it contains and their function.*
- *The sequence of microoperations performed on the binary information stored in the registers.*
- *The control that initiates the sequence of microoperations*.

The symbolic notation used to describe the microoperation transfers among registers is called a *register transfer language*. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated microoperation and transfer the result of the operation to the same or another register. A register transfer language is a system for expressing in symbolic form the microoperation sequences among the registers of a digital module.

### Register Transfer

Computer registers are designated by capital letters (sometimes followed by numerals) to denote the function of the register. For example, the register that holds an address for the memory unit is usually called a *memory address register* and is designated by the name *MAR*. Other designations for registers are *PC* (for *program counter*), *IR* (for *instruction register*), and *R1* (for *processor register*). The individual flip-flops in an *n-bit* register are numbers in sequence from *0* through *n-1*, starting from *0* in the rightmost position and increasing the numbers toward the left. Fig.1 shows the representation of registers in block diagram form. The most common way to represent a register is by a rectangular box with the name of the register inside, as in Fig.1 (a). The individual bits can be distinguished as in (b). The numbering of bits in a 16-bit register can be marked on top of the box as shown in (c). A 16-bit register is partitioned into two parts in (d). Bits 0 through 7 are assigned the symbol *L* (for low byte) and bits 8 through 15 are assigned the symbol *H* (for high byte). The name of the 16-bit register is PC. The symbol *PC (0-7)* or *PC (L)* refers to the low-order byte and *PC (8-15)* or *PC (H)* to the high-order byte.

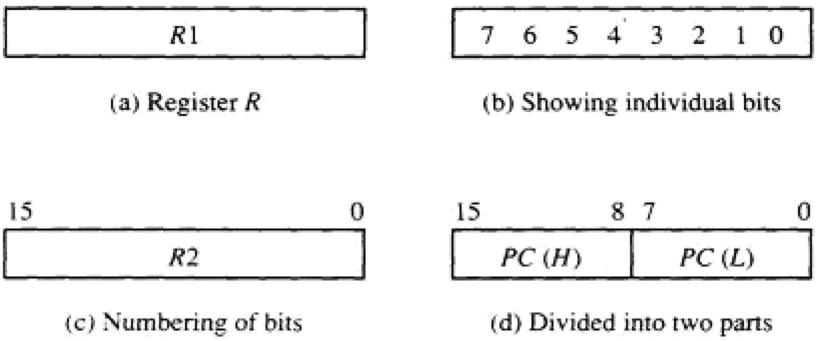



Fig. 1: Block diagram of register

Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement R1 ← R2 denotes a transfer of the content of register R1 into register R2. It designates a replacement of the content of R2 by the content of R1. By definition, the content of the source register R1 does not change after the transfer.

The basic symbols of the register transfer notation are listed in Table-1. Registers are denoted by capital letters, and numerals may follow the letters. Parentheses are used to denote a part of a register by specifying the range of bits or by giving a symbol name to a portion of a register. The arrow denotes a transfer of information and the direction of transfer. A comma is used to separate two or more operations that are executed at the same time. The statement

$$T: R2 \leftarrow R1, R1 \leftarrow R2$$

denotes an operation that exchanges the contents of two registers during one common clock pulse provided that T = 1. This simultaneous operation is possible with registers that have edge-triggered flip-flops.

| Symbol | Description | Examples |
|---|---|---|
| Letters (and numerals) | Denotes a register | $MAR$, $R2$ |
| Parentheses ( ) | Denotes a part of a register | $R2(0\text{–}7)$, $R2(L)$ |
| Arrow ← | Denotes transfer of information | $R2 \leftarrow R1$ |
| Comma , | Separates two microoperations | $R2 \leftarrow R1$, $R1 \leftarrow R2$ |

Table-1: Basic Symbols for Register Transfers

**Bus and Memory Transfers**

A typical digital computer has many registers, and paths must be provided to transfer information from one register to another. A more efficient scheme for transferring information between registers in a multiple-register configuration is a *common bus system*. A bus structure consists of a set of common lines, one for each bit of a register, through which binary information is transferred one at a time. Control signals determine which register is selected by the bus during each particular register transfer.

One way of constructing a common bus system is with *multiplexers*. The multiplexers select the source register whose binary information is then placed on the bus. The construction of a bus system for four registers is shown in Fig.2. Each register has four bits, numbered 0 through 3. The bus consists of four 4 x 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, $S_1$ and $S_0$. In order not to complicate the diagram with 16 lines crossing each other, labels are used to show the connections from the outputs of the registers to the inputs of the multiplexers. For example, output 1 of register A is connected to input 0 of MUX 1 because this input is labeled $A_1$. The diagram shows that the bits in the same significant position in each register are connected to the data inputs of one multiplexer to form one line of the bus. Thus MUX 0 multiplexes the four 0 bits of the registers, MUX 1 multiplexes the four 1 bits of the registers, and similarly for the other two bits.

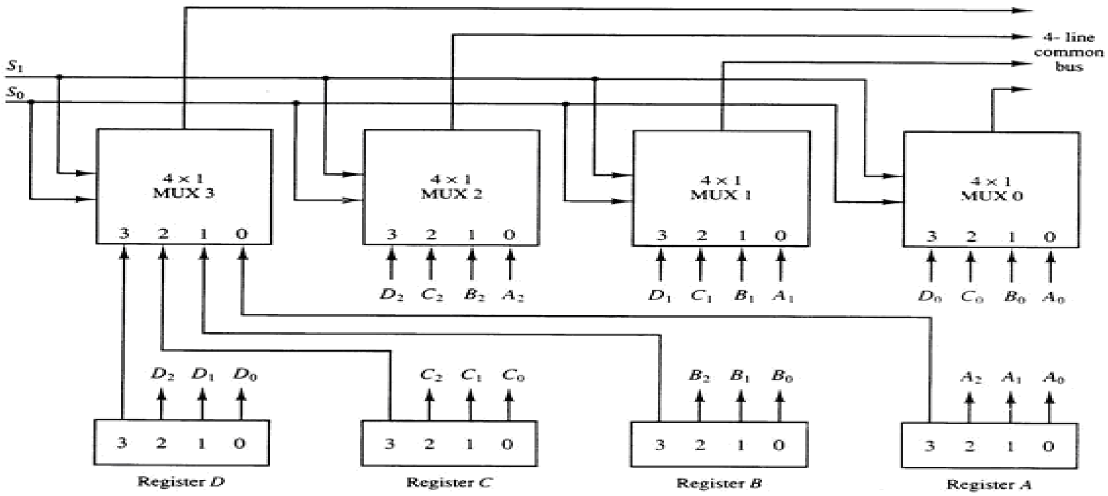



Fig. 2: Bus system for four registers

The two *selection lines* $S_1$ and $S_0$ are connected to the selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus. When $S_1S_0 = 00$, the 0 data inputs of all four multiplexers are selected and applied to the outputs that form the bus. This causes the bus lines to receive the content of register A since the outputs of this register are connected to the 0 data inputs of the multiplexers. Similarly, register B is selected if $S_1S_0 = 01$, and so on. Table-2 shows the register that is selected by the bus for each of the four possible binary values of the selection lines.

| $S_1$ | $S_0$ | *Register selected* |
|---|---|---|
| 0 | 0 | A |
| 0 | 1 | B |
| 1 | 0 | C |
| 1 | 1 | D |

Table-2: Function Table for Bus of Fig. 2

The transfer of information from a bus into one of many destination registers can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control of the particular destination register selected.

**Three-State Bus Buffers**

A bus system can be constructed with *three-state gates* instead of multiplexers. A three-state gate is a digital circuit that exhibits three states. Two of the states are signals equivalent to logic 1 and 0 as in a conventional gate. The third state is a *high-impedance state*. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logical significance.

The graphic symbol of a *three-state buffer* gate is shown in Fig.3. It is distinguished from a normal buffer by having both a normal input and a control input. The control input determines the output state. When the control input is equal to 1, the output is enabled and the gate behaves like any conventional buffer, with the output equal to the normal input. When the control input is 0, the output is disabled and the gate goes to a high-impedance state, regardless of the value in the normal input.

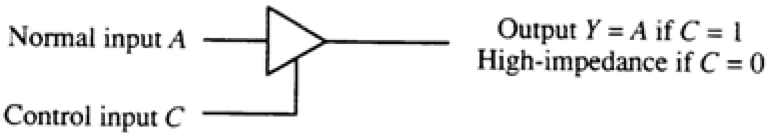



Fig. 3: Graphic symbols for three-state buffer

The construction of a *bus system* with three-state buffers is demonstrated in Fig.4. The outputs of four buffers are connected together to form a single bus line. (It must be realized that this type of connection cannot be done with gates that do not have three-state outputs). The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line.

One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the enable input of the decoder is 0, all of its four outputs are 0, and the bus line is in a high-impedance state because all four buffers are disabled. When the enable input is active, one of the three-state buffers will be active, depending on the binary value in the select inputs of the decoder. To construct a common bus for four registers of n bits each using three-state buffers, we need n circuits with four buffers in each as shown in Fig.4. Each group of four buffers receives one significant bit from the four registers. Each common output produces one of the lines for the common bus for a total of n lines. Only one decoder is necessary to select between the four registers.

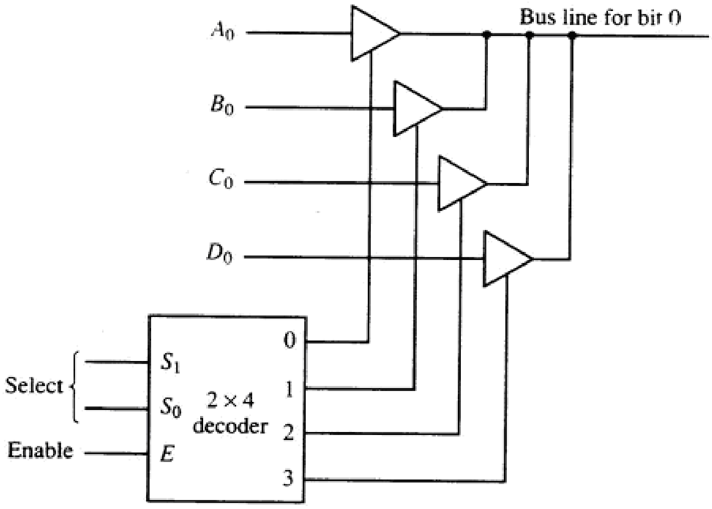



Fig. 4: Bus line with three state-buffers

**Memory Transfer**

The transfer of information from a memory word to the outside environment is called a *read* operation. The transfer of new information to be stored into the memory is called a *write* operation. A memory word will be symbolized by the letter M. The particular memory word among the many available is selected by the memory address during the transfer. It is necessary to specify the address of M when writing memory transfer operations. This will be done by enclosing the address in square brackets following the letter M.

Consider a memory unit that receives the address from a register, called the address register, symbolized by AR. The data are transferred to another register, called the data register, symbolized by DR. The read operation can be stated as follows:

$$\text{Read: } DR \leftarrow M[AR]$$

This causes a transfer of information into DR from the memory word M selected by the address in AR.

The write operation transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is in AR. The write operation can be stated symbolically as follows:

$$\text{Write: } M[AR] \leftarrow R1$$

This causes a transfer of information from R1 into the memory word M selected by the address in AR.

**Arithmetic Microoperations**

A microoperation is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computers are classified into four categories:

1. Register transfer microoperations transfer binary information from one register to another.
2. Arithmetic microoperations perform arithmetic operations on numeric data stored in registers.
3. Logic microoperations perform bit manipulation operations on non-numeric data stored in registers.
4. Shift microoperations perform shift operations on data stored in registers.

The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift. Arithmetic shifts are explained later in conjunction with the shift microoperations. The arithmetic microoperation defined by the statement

$$R3 \leftarrow R1 + R2$$

specifies an *add* microoperation. It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3. To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table-3. Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement:

$$R3 \leftarrow R1 + R2 + 1$$

Adding the contents of R1 to the 2's complement of R2 is equivalent to R1 – R2.

| Symbolic Designation | Description |
|---|---|
| $R3 \leftarrow R1 + R2$ | Contents of R1 plus R2 transferred to R3 |
| $R3 \leftarrow R1 - R2$ | Contents of R1 minus R2 transferred to R3 |
| $R2 \leftarrow R2$ | Complement the contents of R2 (1's complement) |
| $R2 \leftarrow R2 + 1$ | 2's complement the contents of R2 (negate) |
| $R3 \leftarrow R1 + R2 + 1$ | R1 plus the 2's complement of R2 (subtraction) |
| $R1 \leftarrow R1 + 1$ | Increment the contents of R1 by one |
| $R1 \leftarrow R1 - 1$ | Decrement the contents of R1 by one |

Table-3: Arithmetic Microoperations

The increment and decrement microoperations are symbolized by plus-one and minus-one operations, respectively.

**Binary Incrementer**

The increment microoperation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented. This microoperation is easily implemented with a binary counter. Every time the count enable is active, the clock pulse transition increments the content of the register by one. There may be occasions when the increment microoperation must be done with a combinational circuit independent of a particular register. This can be accomplished by means of half-adders connected in cascade.

The diagram of a 4-bit combinational circuit incrementer is shown in Fig.7. One of the inputs to the least significant half-adder (HA) is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one half-adder is connected to one of the inputs of the next-higher-order half-adder. The circuit receives the four bits from $A_0$ through $A_3$, adds one to it, and generates the incremented output in $S_0$ through $S_3$. The output carry $C_4$ will be 1 only after incrementing binary 1111. This also causes outputs $S_0$ through $S_3$ to go to 0.

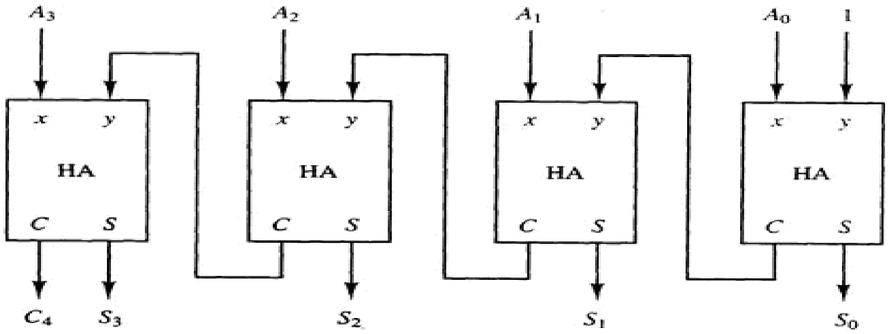



Fig. 7: 4-bit binary incrementer

The circuit of Fig.7 can be extended to an n-bit binary incrementer by extending the diagram to include n half-adders. The least significant bit must have one input connected to logic-1. The other inputs receive the number to be incremented or the carry from the previous stage.

**Arithmetic Circuit**

The arithmetic microoperations listed in Table 3 can be implemented in one composite arithmetic circuit. The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

The diagram of a 4-bit arithmetic circuit is shown in Fig.8. It has four full-adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs A and B and a 4-bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B is connected to the data inputs of the multiplexers. The multiplexers data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1. Logic-0 is a fixed voltage value (0 volts for TTL integrated circuits) and the logic-1 signal can be generated through an inverter whose input is 0. The four multiplexers are controlled by two selection inputs, $S_1$ and $S_0$. The input carry $C_{in}$ goes to the carry input of the FA in the least significant position. The other carries are connected from one stage to the next.

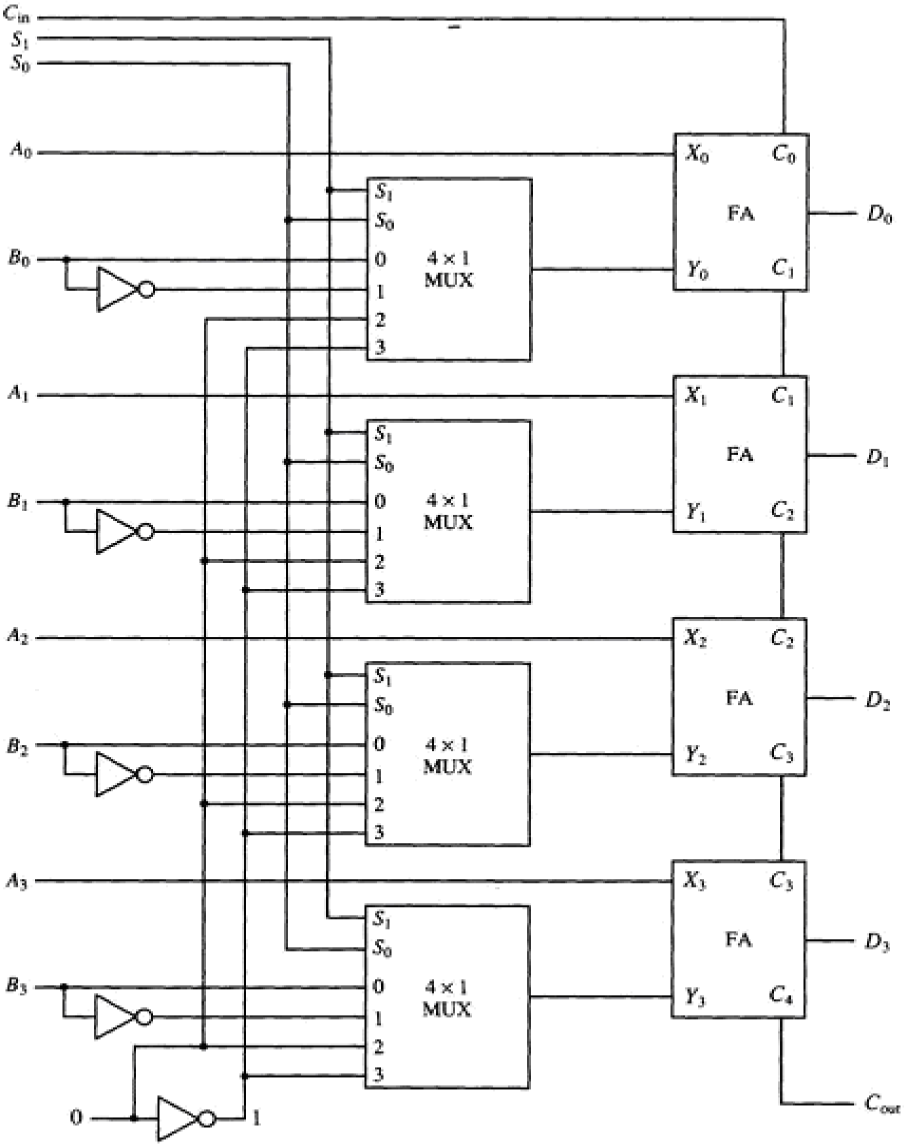



Fig. 8: 4-bit arithmetic circuit

The output of the binary adder is calculated from the following arithmetic sum:D = A + Y + $C_{in}$, where A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder. $C_{in}$ is the input carry, which can be equal to 0 or 1. Note that the symbol + in the equation above denotes an arithmetic plus. By controlling the value of Y with the two selection inputs $S_1$ and $S_0$ and making $C_{in}$ equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in Table-4.

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

Table-4: Arithmetic Circuit Function Table

When $S_1S_0$ = 00, the value of B is applied to the Y inputs of the adder.If $C_{in}$ = 0, the output D = A + B. If $C_{in}$ = 1, output D = A + B + 1. Both cases perform the add microoperation with or without adding the input carry.

When $S_1S_0$ = 01, the complement of B is applied to the Y inputs of the adder. If $C_{in}$ = 1, then D = A + B + 1. This produces A plus the 2's complement of B, which is equivalent to a subtraction of A – B. When $C_{in}$ = 0, then D = A + B. This is equivalent to a subtract with borrow, that is, A – B – 1.

When $S_1S_0$ = 10, the inputs from B are neglected, and instead, all 0's are inserted into the Y inputs. The output becomes D = A + 0 + $C_{in}$. This gives D = A when $C_{in}$ = 0 and D = A + 1 when $C_{in}$ = 1. In the first case we have a direct transfer from input A to output D. In the second case, the value of A is incremented by 1.

When $S_1S_0$ = 11, all 1's are inserted into the Y inputs of the adder to produce the decrement operation D = A – 1 when $C_{in}$ = 0. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces F = A + 2's complement of 1 = A – 1. When $C_{in}$ = 1, then D = A – 1 + 1 = A, which causes a direct transfer from input A to output D. Note that the microoperation D = A is generated twice, so there are only seven distinct microoperations in the arithmetic circuit.

## Logic Microoperations

Logic microoperations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example, the exclusive-OR microoperation with the contents of two registers R1 and R2 is symbolized by the statement

$$P: R1 \leftarrow R1 \oplus R2$$

It specifies a logic microoperation to be executed on the individual bits of the registers provided that the control variable P = 1. As a numerical example, assume that each register has four bits. Let the content of R1 be 1010 and the content of R2 be 1100. The exclusive-OR microoperation stated above symbolizes the following logic computation:

1010 Content of R1
1100 Content of R2
0110 Content of R1 after P = 1

The content of R1, after the execution of the microoperation, is equal to the bit-by-bit exclusive-OR operation on pairs of bits in R2 and previous values of R1. The logic microoperations are seldom used in scientific computations, but they are very useful for bit manipulation of binary data and for making logical decisions.

Special symbols will be adopted for the logic microoperations OR, AND, and complement, to distinguish them from the corresponding symbols used to express Boolean functions. The symbol $\vee$ will be used to denote an OR microoperation and the symbol $\wedge$ to denote an AND microoperation. The complement microoperation is the same as the 1's complement and uses a bar on top of the symbol that denotes the register name. By using different symbols, it will be possible to differentiate between a logic microoperation and a control (or Boolean) function. Another reason for adopting two sets of symbols is to be able to distinguish the symbol +, when used to symbolize an arithmetic plus, from a logic OR operation. Although the + symbol has two meanings, it will be possible to distinguish between them by noting where the symbol occurs. When the symbol + occurs in a microoperation, it will denote an arithmetic plus. When it occurs in a control (or Boolean) function, it will denote an OR operation. We will never use it to symbolize an OR microoperation.

### List of Logic Microoperations

There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in Table-5. In this table, each of the 16 columns $F_0$ through $F_{15}$ represents a truth table of one possible Boolean function for the two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

| $x$ | $y$ | $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ | $F_6$ | $F_7$ | $F_8$ | $F_9$ | $F_{10}$ | $F_{11}$ | $F_{12}$ | $F_{13}$ | $F_{14}$ | $F_{15}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

Table-5: Truth Table for 16 Functions of Two Variables

The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table-6. The 16 logic microoperations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B.

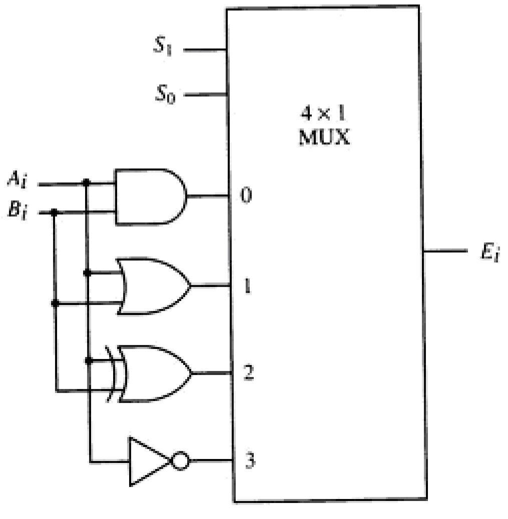
| Boolean function | Microoperation | Name |
|---|---|---|
| $F_0 = 0$ | $F \leftarrow 0$ | Clear |
| $F_1 = xy$ | $F \leftarrow A \wedge B$ | AND |
| $F_2 = xy'$ | $F \leftarrow A \wedge \overline{B}$ | |
| $F_3 = x$ | $F \leftarrow A$ | Transfer $A$ |
| $F_4 = x'y$ | $F \leftarrow \overline{A} \wedge B$ | |
| $F_5 = y$ | $F \leftarrow B$ | Transfer $B$ |
| $F_6 = x \oplus y$ | $F \leftarrow A \oplus B$ | Exclusive-OR |
| $F_7 = x + y$ | $F \leftarrow A \vee B$ | OR |
| $F_8 = (x + y)'$ | $F \leftarrow \overline{A \vee B}$ | NOR |
| $F_9 = (x \oplus y)'$ | $F \leftarrow \overline{A \oplus B}$ | Exclusive-NOR |
| $F_{10} = y'$ | $F \leftarrow \overline{B}$ | Complement $B$ |
| $F_{11} = x + y'$ | $F \leftarrow A \vee \overline{B}$ | |
| $F_{12} = x'$ | $F \leftarrow \overline{A}$ | Complement $A$ |
| $F_{13} = x' + y$ | $F \leftarrow \overline{A} \vee B$ | |
| $F_{14} = (xy)'$ | $F \leftarrow \overline{A \wedge B}$ | NAND |
| $F_{15} = 1$ | $F \leftarrow$ all 1's | Set to all 1's |

Table-6: Sixteen Logic Microoperations

**Hardware Implementation**

The hardware implementation of logic microoperations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function. Although there are 16 logic microoperations, most computers use only four – AND, OR, XOR (exclusive-OR), and complement –from which all others can be derived.

Fig. 9 (a) shows one stage of a circuit that generates the four basic logic microoperations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs $S_1$ and $S_0$ choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript *i*. For a logic circuit with *n* bits, the diagram must be repeated *n* times for $i = 0, 1, 2, ..., n-1$. The selection variables are applied to all stages. The function table in Fig.9 (b) lists the logic microoperations obtained for each combination of the selection variables.




(a) Logic diagram

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

(b) Function table

Fig. 9: One stage of logic circuit

**Some Applications**

Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into a register. The following examples show how the bits of one register (designated by A) are manipulated by logic microoperations as a function of the bits of another register (designated by B).

The *selective-set* operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B. The following numerical example clarifies this operation:

| | |
|---|---|
| 1010 | A before |
| 1100 | B (logic operand) |
| 1110 | A after |

We note that the bits of A after the operation are obtained from the logic-OR operation of bits in B and previous values of A. Therefore, the OR microoperation can be used to selectively set bits of a register.

The *selective-complement* operation complements bits in A where there are corresponding 1's in B. It does not affect bit positions that have 0's in B. For example:

| | |
|---|---|
| 1010 | A before |
| 1100 | B (logic operand) |
| 0110 | A after |

The selective-complement operation is just an exclusive-OR microoperation. Therefore, the exclusive-OR microoperation can be used to selectively complement bits of a register.

The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1's in B. For example:

| | |
|---|---|
| 1010 | A before |
| 1100 | B (logic operand) |
| 0010 | A after |

Again the two leftmost bits of B are 1's, so the corresponding bits of A are cleared to 0. One can deduce that the Boolean operation performed on the individual bits is AB'.

The *mask* operation is similar to the selective-clear operation except that the bits of A are cleared only where there are corresponding 0's in B. The mask operation is an AND micro operation as seen from the following numerical example:

| | |
|---|---|
| 1010 | A before |
| 1100 | B (logic operand) |
| 1000 | A after masking |

The *insert* operation inserts a new value into a group of bits. This is done by first masking the bits and then ORing them with the required value. For example, suppose that an A register contains eight bits, 0110, 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

| | |
|---|---|
| 0110 1010 | A before |
| 0000 1111 | B (mask) |
| 0000 1010 | A after masking |

and then insert the new value:

| | |
|---|---|
| 0000 1010 | A before |
| 1001 0000 | B (insert) |
| 1001 1010 | A after insertion |

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

The *clear* operation compares the words in A and B and produces an all 0s result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation as shown by the following example:

| | |
|---|---|
| 1010 | A |
| 1010 | B |
| 0000 | A (after clear) |

**Shift Microoperations**

Shift microoperations are used for serial transfer of data. The contents of a register can be shifted to the left or the right. During a shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. The information transferred through the serial input determines the type of shift. There are three types of shifts: logical, circular, and arithmetic. A logical shift is one that transfers 0 through the serial input. We will adopt the symbols *shl* and *shr* for logical shift-left and shift-right microoperations. For example:

$$R1 \leftarrow \mathit{shl}\ R1$$
$$R2 \leftarrow \mathit{shl}\ R2$$

are two microoperations that specify a 1-bit shift to the left of the content of register R1 and a 1-bit shift to the right of the content of register R2. The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

The circular shift (also known as a rotate operation) circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of the shift register to its serial input. We will use the symbols *cil* and *cir* for the circular shift left and right, respectively. The symbolic notation for the shift microoperations is shown in Table-7.

| Symbolic designation | Description |
|---|---|
| $R \leftarrow \mathrm{shl}\ R$ | Shift-left register $R$ |
| $R \leftarrow \mathrm{shr}\ R$ | Shift-right register $R$ |
| $R \leftarrow \mathrm{cil}\ R$ | Circular shift-left register $R$ |
| $R \leftarrow \mathrm{cir}\ R$ | Circular shift-right register $R$ |
| $R \leftarrow \mathrm{ashl}\ R$ | Arithmetic shift-left $R$ |
| $R \leftarrow \mathrm{ashr}\ R$ | Arithmetic shift-right $R$ |

Table-7: Shift Microoperations

An arithmetic shift is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The leftmost bit in a register holds the sign bit, and the remaining bits hold the number. The sign bit is 0 for positive and 1 for negative. Negative numbers are in 2’s complement form. Fig.10 shows a typical register of *n* bits. Bit $R_{n-1}$ in the leftmost position holds the sign bit.

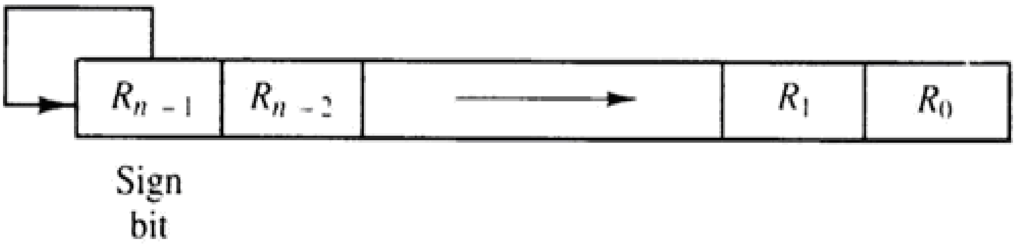
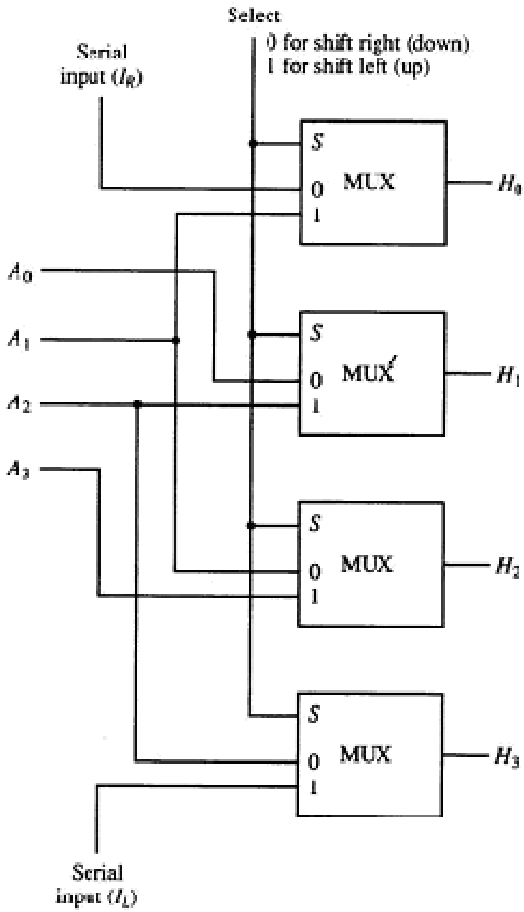



Fig. 10: Arithmetic shift right

The arithmetic shift-left inserts a 0 into $R_0$, and shifts all other bits to the left. The initial bit of $R_{n-1}$ is lost and replaced by the bit from $R_{n-2}$.

**Hardware Implementation**

A combinational circuit shifter can be constructed with multiplexers as shown in Fig.11. The 4-bit shifter has four data inputs. $A_0$ through $A_3$, and four data outputs, $H_0$ through $H_3$. There are two serial inputs, one for shift left ($I_L$) and the other for shift right ($I_R$). When the selection input S = 0, the input data are shifted right (down in the diagram). When S = 1, the input data are shifted left (up in the diagram). The function table in Fig.11 shows which input goes to each output after the shift. A shifter with *n* data inputs and outputs requires *n* multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.




Function table

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

Fig. 11: 4-bit combinational circuit shifter

## Arithmetic Logic Shift Unit

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

The arithmetic, logic, and shift circuits can be combined into one ALU with common selection variables. One stage of an arithmetic logic shift unit is shown in Fig.12. The subscript *i* designates a typical stage. Inputs $A_i$ and $B_i$ are applied to both the arithmetic and logic units. A particular microoperation is selected with inputs $S_1$ and $S_0$. A 4 x 1 multiplexer at the output chooses between an arithmetic output in $E_i$ and a logic output in $H_i$. The data in the multiplexer are selected with inputs $S_3$ and $S_2$. The other two data inputs to the multiplexer receive inputs $A_{i-1}$ for the shift-right operation and $A_{i+1}$ for the shift-left operation. Note that the diagram shows just one typical stage. The circuit of Fig.12 must be repeated *n* times for an *n*-bit ALU. The output-carry $C_{i+1}$ of a given arithmetic stage must be connected to the input carry $C_i$ of the next stage in sequence. The input carry to the first stage is the input carry $C_{in}$, which provides a selection variable for the arithmetic operations.

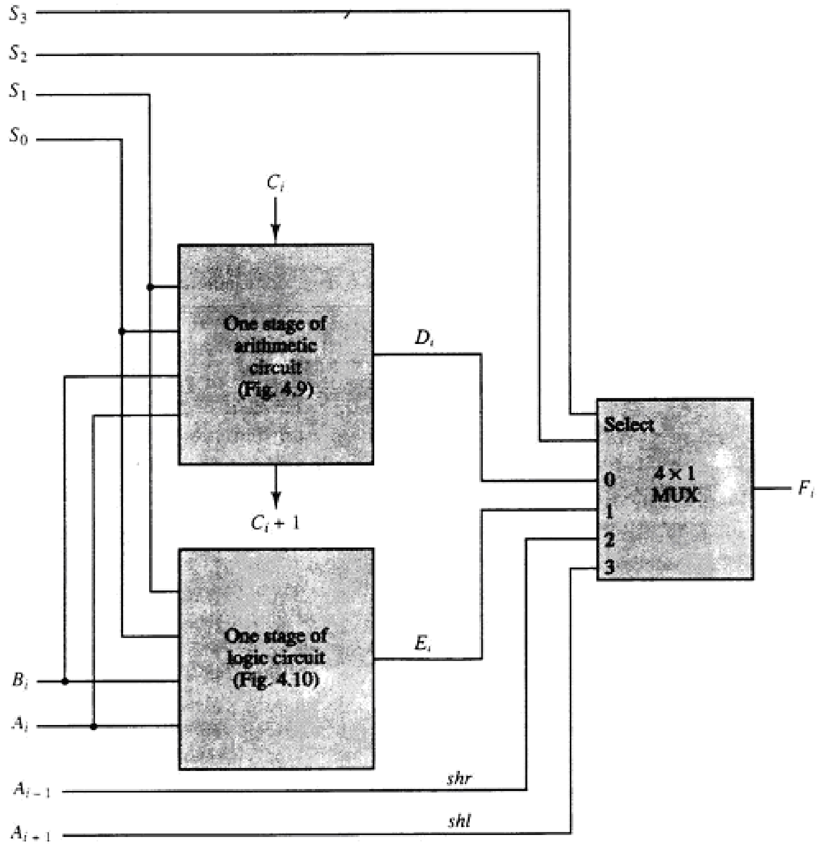



Fig. 12: One stage of arithmetic logic shift unit

The circuit whose one stage is specified in Fig.12 provides eight arithmetic operation, four logic operations, and two shift operations. Each operation is selected with the five variables $S_3$, $S_2$, $S_1$, $S_0$, and $C_{in}$. The input carry $C_{in}$ is used for selecting an arithmetic operation only.

Table-8 lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with $S_3S_2$ = 00. The next four are logic operations and are selected with $S_3S_2$ = 01. The input carry has no effect during the logic operations and is marked with don't-care x's. The last two operations are shift operations and are selected with $S_3S_2$ = 10 and 11. The other three selection inputs have no effect on the shift.

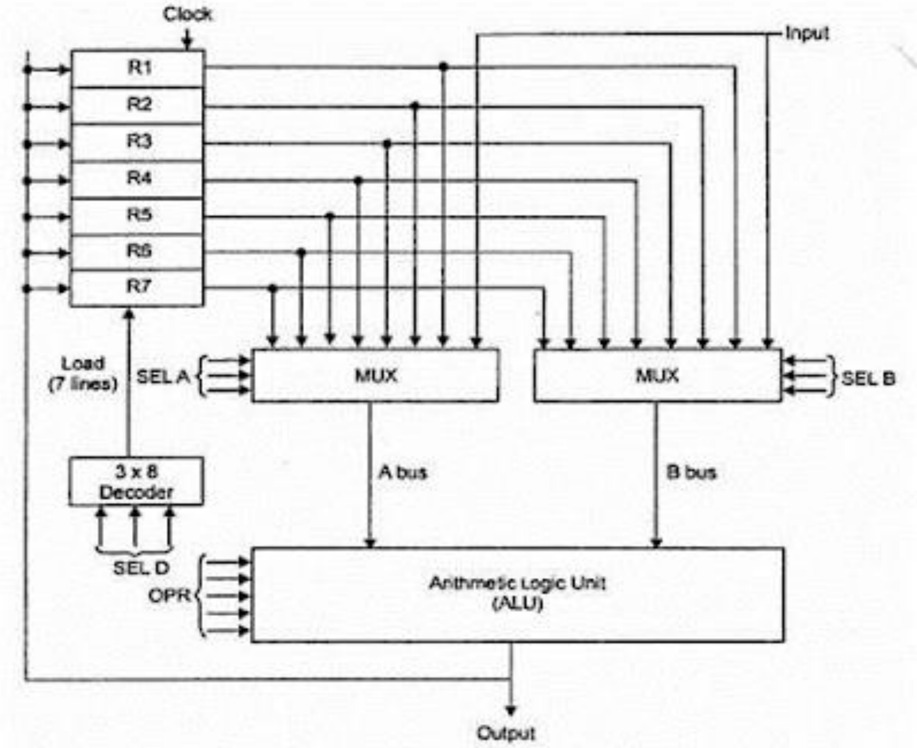
| Operation select | | | | | | |
|---|---|---|---|---|---|---|
| $S_3$ | $S_2$ | $S_1$ | $S_0$ | $C_{in}$ | Operation | Function |
| 0 | 0 | 0 | 0 | 0 | $F = A$ | Transfer $A$ |
| 0 | 0 | 0 | 0 | 1 | $F = A + 1$ | Increment $A$ |
| 0 | 0 | 0 | 1 | 0 | $F = A + B$ | Addition |
| 0 | 0 | 0 | 1 | 1 | $F = A + B + 1$ | Add with carry |
| 0 | 0 | 1 | 0 | 0 | $F = A + \overline{B}$ | Subtract with borrow |
| 0 | 0 | 1 | 0 | 1 | $F = A + \overline{B} + 1$ | Subtraction |
| 0 | 0 | 1 | 1 | 0 | $F = A - 1$ | Decrement $A$ |
| 0 | 0 | 1 | 1 | 1 | $F = A$ | Transfer $A$ |
| 0 | 1 | 0 | 0 | × | $F = A \wedge B$ | AND |
| 0 | 1 | 0 | 1 | × | $F = A \vee B$ | OR |
| 0 | 1 | 1 | 0 | × | $F = A \oplus B$ | XOR |
| 0 | 1 | 1 | 1 | × | $F = \overline{A}$ | Complement $A$ |
| 1 | 0 | × | × | × | $F = \text{shr } A$ | Shift right $A$ into $F$ |
| 1 | 1 | × | × | × | $F = \text{shl } A$ | Shift left $A$ into $F$ |

Table-8: Function Table for Arithmetic Logic Shift Unit

## CENTRAL PROCESSING UNIT

## GENERAL REGISTER ORGANIZATION:

When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system. The registers communicate with each other not only for direct data transfers, but also while performing various micro-operations. Hence, it is necessary to provide a common unit that can perform all the arithmetic, logic and shift micro-operation in the processor. The block diagram of a bus organization for seven CPU registers is shown in the figure below.



The output of each register is connected to two multiplexers (MUX) to form the two buses A & B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses forms the input to a common ALU. The operation selected in the ALU determines the arithmetic or logic micro-operation that is to be performed. The result of the micro-operation is available for output and also goes into the inputs of the registers. The register that receives the information from the output bus is selected by a decoder. The decoder activates one of the register load inputs, thus providing a transfer both between the data in the output bus and the inputs of the selected destination register.

The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the systems.

$$R_1 \leftarrow R_2 + R_3$$

1. MUX A selection (SEC A): to place the content of R2 into bus A
2. MUX B selection (sec B): to place the content of R3 into bus B
3. ALU operation selection (OPR): to provide the arithmetic addition (A + B)
4. Decoder destination selection (SEC D): to transfer the content of the output bus into $R_1$

The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle. The data from the two source registers propagate through the gates in the multiplexer and the ALU, to the output bus, and into the inputs of the destination register, all during the clock cycle interval.

**Control Word**:

There are 14 binary selection inputs in the units, and their combined value specifies a control word. It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA select a source register for the A input of the ALU. The three bits of SELB select a source register for the B input of the ALU. The three bits of SECD select a destination register using the decoder and its seven load outputs. The five bits of OPR select one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specify a particular micro-operation.

**Example of Micro-operation:**
The control word of the micro-operation $R_1 \leftarrow R_2 - R_3$ is given in the table below.

| Field | SELA | SELB | SELD | OPR |
|---|---|---|---|---|
| Symbol | $R_2$ | $R_3$ | $R_1$ | SUB |
| Control Word | 010 | 011 | 001 | 00101 |

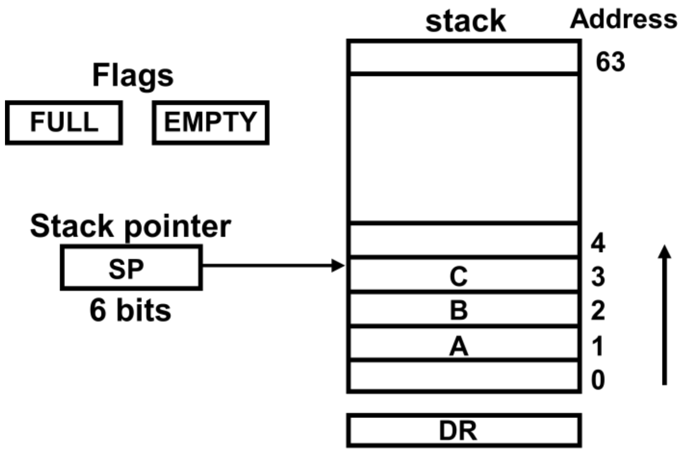
**STACK ORGANIZATION**

A useful feature that is included in the CPU of most computers is a stack or last-in first out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation a stack can be companied to a stack of trays.

The stack in Digital Computer is essentially a memory unit with an address register that can count only (after an initial value is loaded into it.) The register that holds the address for the stack is called a Stack Pointer (SP) because its values always points at the top item in the stack.

The two operations: PUSH (insert) and POP (delete)

**Register Stack**:- A stack can be placed in a portion of a large memory as it can be organized as a collection of a finite number of memory words as register.




In a 64- word stack (shown above), the stack pointer contains 6 bits because $2^6 = 64$.

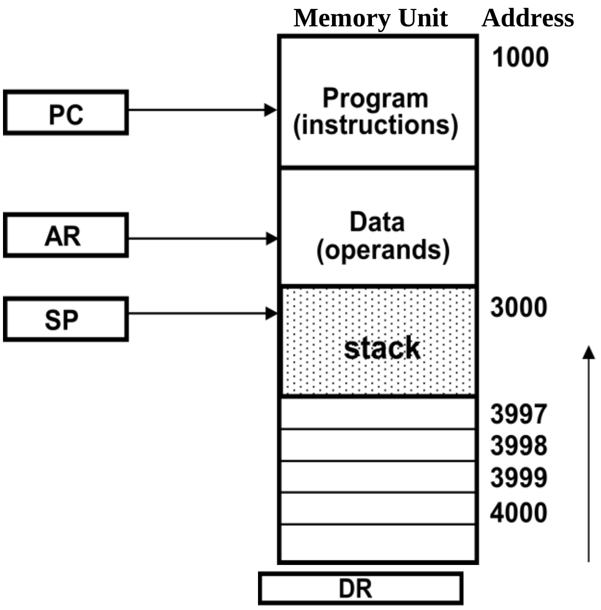
The one bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty. DR is the data register that holes the binary data to be written into on read out of the stack.

Initially, SP is decide to 0, EMTY is set to 1, FULL = 0, so that SP points to the word at address 0 and the stack is masked empty and not full.

| | | |
|---|---|---|
| **PUSH** | SP ← SP + 1 | increment stack pointer |
| | M [SP] ← DR | write item on top of the Stack |
| | If (SP = 0) then (FULL←1) | check if stack is full |
| | EMTY ← 0 | mask the stack not empty |
| **POP** | DR ← [SP] | read item from the top of stack |
| | SP ← SP –1 | decrement SP |
| | If (SP = 0) then (EMPTY ←1) | check if stack is empty |
| | FULL ← 0 | mark the stack not full. |

**Memory Stack:**

A stack can exist as a stand-alone unit or can be executed in a random-access memory attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory. A portion of memory is assigned to a stack operation and a processor register is used as a stack pointer to execute stack in the CPU. Figure below shows a portion of computer memory partitioned into three segments - program, data, and stack. The address of the next instruction in the program is located by the program counter PC while an array of data is pointed by address register AR. The top of the stack is located by the stack pointer SP. The three registers are connected to a common address bus, which connects the three registers and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.




**INSTRUCTION FORMATS**

The most common fields found in instruction format are:

1. An operation code field that specified the operation to be performed
2. An address field that designates a memory address or a processor registers.
3. A mode field that specifies the way the operand or the effective address is determined.

Computers may have instructions of several different lengths containing varying number of addresses. The number of address field in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organization.

| | | | |
|---|---|---|---|
| 1. | Single Accumulator organization | ADD X | $AC \leftarrow AC + M[X]$ |
| 2. | General Register Organization | ADD R1, R2, R3 | $R \leftarrow R2 + R3$ |
| 3. | Stack Organization | PUSH X | |

**Three address Instruction**

Computer with three addresses instruction format can use each address field to specify either processor register are memory operand.

| | |
|---|---|
| ADD $R_1$, A, B | $A_1 \leftarrow M[A] + M[B]$ |
| ADD $R_2$, C, D | $R_2 \leftarrow M[C] + M[B]$ |
| MUL X, $R_1$, $R_2$ | $M[X] \leftarrow R_1 * R_2$ |

The advantage of the three address formats is that it results in short program when evaluating arithmetic expression. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

**Two Address Instruction**

Each address field can specify either a processes register on a memory word.

| | | |
|---|---|---|
| MOV | $R_1$, A | $R_1 \leftarrow M[A]$ |
| ADD | $R_1$, B | $R_1 \leftarrow R_1 + M[B]$ |
| MOV | $R_2$, C | $R_2 \leftarrow M[C]$ |
| ADD | $R_2$, D | $R_2 \leftarrow R_2 + M[D]$ |
| MUL | $R_1$, $R_2$ | $R_1 \leftarrow R_1 * R_2$ |
| MOV | $X_1$ $R_1$ | $M[X] \leftarrow R_1$ |

**One Address instruction**

It used an implied accumulator (AC) register for all data manipulation. For multiplication/division, there is a need for a second register.

| | | |
|---|---|---|
| LOAD | A | AC ← M [A] |
| ADD | B | AC ← AC + M [B] |
| STORE | T | M [T] ← AC |
| LOAD | C | AC ← M (C) |
| ADD | D | AC ← AC + M (D) |
| MUL | T | AC ← AC + M (T) |
| STORE | X | M [X]← AC |

**Zero – Address Instruction**

A stack organized computer does not use an address field for the instruction ADD and MUL. The PUSH & POP instruction, however, need an address field to specify the operand that communicates with the stack (TOS ← top of the stack)

| | |
|---|---|
| PUSH A | TOS ← A |
| PUSH B | TOS ← B |
| ADD | TOS ← (A + B) |
| PUSH C | TOS ← C |
| PUSH D | TOS ← D |
| ADD | TOS ← (C + D) |
| MUL | TOS ← (C + D) * (A + B) |
| POP X | M [X] ← TOS |

**ADDRESSING MODES**

An addressing mode is a method of fetching operands for various operations. Addressing modes available in most of the processors are discussed below.

**Implied Mode:** This mode specifies the operands implicitly in the definition of the instruction. For example, the instruction ‘‘complement accumulator’’ is an implied mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register references instructions that use an accumulator are implied mode instructions. Zero-address introductions are implied mode instructions.

**Immediate Mode:** The operand is specified in the instruction itself in this mode i.e. the immediate mode instruction has an operand field rather than an address field. The actual operand to be used in conjunction with the operation specified in the instruction is contained in the operand field.

**Register Mode:** In this mode the operands are in registers that reside within the CPU. The register required is chosen from a register field in the instruction.

**Register Indirect Mode:** In this mode the instruction specifies a register in the CPU that contains the address of the operand and not the operand itself. Usage of register indirect mode instruction necessitates the placing of memory address of the operand in the processor register with a previous instruction.

**Autoincrement or Autodecrement Mode**: After execution of every instruction from the data in memory it is necessary to increment or decrement the register. This is done by using the increment or decrement instruction. Given upon its sheer necessity some computers use special mode that increments or decrements the content of the registers automatically.

**Direct Address Mode**: In this mode the operand resides in memory and its address is given directly by the address field of the instruction such that the affective address is equal to the address part of the instruction.

**Indirect Address Mode:** Unlike direct address mode, in this mode give the address field gives the address where the effective address is stored in memory. The instruction from memory is fetched through control to read is address part to access memory again to read the effective address. A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following equation:

Effective address = address part of instruction + content of CPU register

The CPU Register used in the computation may be the program counter, Index Register or a base Register.

**Relative Address Mode:** This mode is applied often with branch type instruction where the branch address position is relative to the address of the instruction word itself. As such in the mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address whose position in memory is relative to the address of the next instruction.

**Indexed Addressing Mode:** In this mode the effective address is obtained by adding the content of an index register to the address part of the instruction. The index register is a special CPU register that contains an index value and can be incremented after its value is used to access the memory.

**Base Register Addressing Mode:** In this mode the affective address is obtained by adding the content of a base register to the part of the instruction like that of the indexed addressing mode though the register here is a base register and not a index register.

**DATA TRANSFER AND MANIPULATION**

Computer provides an extensive set of instructions to give the user the flexibility to carryout various computational tasks. Most computer instructions can be classified into three categories.

1. Data transfer instructions
2. Data manipulation instructions
3. Program control instructions

Data transfer instruction cause transfer of data from one location to another without changing the binary information content. Data manipulation instructions are those that perform arithmetic logic, and shift operations. Program control instructions provide decision-making capabilities and change the path taken by the program when executed in the computer.

**Data Transfer Instructions:**

Data transfer instruction move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processes registers, between processor registers and input or output, and between processor register themselves.

The "load" instruction represents a transfer from memory to a processor register, usually an "accumulator" where as the store instruction designates a transfer from a processor register into memory. The move instruction is employed in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words. Swapping of information between to registers of a register and memory word is accomplished by using the exchange instruction. The input and output instructions cause transfer of data among processor registers and input or output terminals. The push and pop instructions take care of transfer of data between processor registers and a memory stack. The data transfer instructions are shown in the following table.

| Name | Mnemonic |
|---|---|
| Load | LD |
| Store | ST |
| Move | MOV |
| Exchange | XCH |
| Input | IN |
| Output | OUT |
| Push | PUSH |
| Pop | POP |

**Data Manipulation Instructions:**

These instructions perform operations on data and provide the computational capabilities for the computer. The data manipulation instructions in a typical computer are usually divided into three basic types.

1. Arithmetic Instructions
2. Logical bit manipulation Instructions
3. Shift Instructions

**Arithmetic Instructions:**

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most of the computers carry instructions for all four operations. The increment instruction adds 1 to the value stored in a register or memory word. The decrement instruction subtracts 1 from the value stored in a register or memory word. The instruction “add with carry” performs the addition on two operands plus the value of the carry from the previous computation. Similarly, the “subtract with borrow” instruction subtracts two words and a borrow which may have resulted from a previous subtract operation. The negate instruction forms the 2’s complement of a number, effectively reversing the sign of an integer when represented in the signed-2’s complement form. The arithmetic instructions are shown in the following table.

| **Name** | **Mnemonic** |
|---|---|
| Increment | INC |
| Decrement | DEC |
| Add | Add |
| Subtract | Sub |
| Multiply | MUL |
| Divide | DIV |
| Add with Carry | ADDC |
| Subtract with Borrow | SUBB |
| Negate (2’s Complement) | NEG |

**Logical and Bit Manipulation Instructions:**

Some typical logical and bit manipulation instructions are listed in the following table. The clear instruction causes the specified operand to be replaced by 0’s. The complement instruction produces the l’s complement by inverting all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on each bit of the operands separately. Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations. There are three bit manipulation operations possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented.

| **Name** | **Mnemonic** |
|---|---|
| Clear | CLR |
| Complement | COM |
| AND | AND |
| OR | OR |
| Exclusive-Or | XOR |
| Clear Carry | CLRC |
| Set Carry | SETC |
| Complement Carry | COMC |
| Enable Interrupt | EI |
| Disable Interrupt | DI |

**Shift Instructions:**

Instructions to shift the content of an operand are quite useful and are often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify either logical shifts, arithmetic shifts, or rotate-type shifts.

Four types of shift instructions are listed below in the table. The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left. Arithmetic shifts are used in conformity with the rules for signed-2's complement numbers. The arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unaltered. This is a shift-right operation wherein the end bit remains unchanged the same. The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction.

| Name | Mnemonic |
|---|---|
| Logical shift right | SHR |
| Logical shift left | SHL |
| Arithmetic shift right | SHRA |
| Arithmetic shift left | SHLA |
| Rotate right | ROR |
| Rotate left | ROL |
| Rotate right through carry | RORC |
| Rotate left through carry | ROLC |

A circular shift is produced by the rotate instructions. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated. Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry and at the same time, and shifts the entire register to the left.

**Program Control Instructions:**

The conditions for altering the content of the program counter, are specified by program control instruction, and the conditions for data-processing operations are specify by data transfer and manipulation instructions. As a result of execution of a program control instruction, a change in the value of program counter occurs, which causes a break in the sequence of instruction execution. Some typical program control instructions are listed in Table below.

| Name | Mnemonic |
|---|---|
| Branch | BR |
| Jump | JMP |
| Skip | SKP |
| Call | CALL |
| Return | RET |
| Compare | CMP |
| Test | TST |

The branch and jump instructions are identical in their use but sometimes they are used to denote different addressing modes. The branch is usually a one-address instruction. Branch and jump instructions may be conditional or unconditional. An unconditional branch instruction, as a name denotes, causes a branch to the specified address without any conditions. On the contrary the conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address. If the condition is not met, the program counter remains unaltered and the next instruction is taken from the next location in sequence.

The skip instruction does not require an address field and is, therefore, a zero-address instruction. A conditional skip instruction will skip the next instruction if the condition is met. The call and return instructions are used in conjunction with subroutines. The compare instruction performs a subtraction between two operands, but the result of the operation is not retained.

**There are three solutions for every problem to lead a happy life:**
**Accept it, change it, or leave it.**
**If you can't accept it, change it. If you can't change it, leave it.**
**You will certainly enjoy your life**