UNIT – I

Number System and Codes

- **Binary number system**
- **Binary to Decimal**
- **Decimal to Binary**
- **Hexadecimal**
- **Ascii code**
- **Excess-3 Code**
- **Gray Code**

Digital Logic

- **Basic Gates- AND, OR, NOT**
- **Universal Logic Gates- NOR, NAND**

NUMBER SYSTEM AND CODES

In a digital system, the system can understand only the optional number system. In these systems, digits symbols are used to represent different values, depending on the index from which it settled in the number system.

In simple terms, for representing the information, we use the number system in the digital system.

The digit value in the number system is calculated using:

1. The digit
2. The index, where the digit is present in the number.
3. Finally, the base numbers, the total number of digits available in the number system.

Types of Number System

In the digital computer, there are various types of number systems used for representing information.

1. Binary Number System
2. Decimal Number System
3. Hexadecimal Number System
4. Octal Number System

Binary Number System

Generally, a binary number system is used in the digital computers. In this number system, it carries only two digits, either 0 or 1. There are two types of electronic pulses present in a binary number system. The first one is the absence of an electronic pulse representing '0'and second one is the presence of electronic pulse representing '1'. Each digit is known as a bit. A four-bit collection (1101) is known as a nibble, and a collection of eight bits (11001010) is known as a byte. The location of a digit in a binary number represents a specific power of the base (2) of the number system.

Characteristics:

1. It holds only two values, i.e., either 0 or 1.
2. It is also known as the base 2 number system.
3. The position of a digit represents the 0 power of the base(2). Example: 2^0
4. The position of the last digit represents the x power of the base(2). Example: 2^x, where x represents the last position, i.e., 1

Examples:

$(10100)_2$, $(11011)_2$, $(11001)_2$, $(000101)_2$, $(011010)_2$.

Decimal Number System

The decimal numbers are used in our day to day life. The decimal number system contains ten digits from 0 to 9(base 10). Here, the successive place value or position, left to the decimal point holds units, tens, hundreds, thousands, and so on.

The position in the decimal number system specifies the power of the base (10). The 0 is the minimum value of the digit, and 9 is the maximum value of the digit. For example, the decimal number 2541 consist of the digit 1 in the unit position, 4 in the tens position, 5 in the hundreds position, and 2 in the thousand positions and the value will be written as:

$$(2\times1000) + (5\times100) + (4\times10) + (1\times1)$$
$$(2\times10^3) + (5\times10^2) + (4\times10^1) + (1\times10^0)$$
$$2000 + 500 + 40 + 1$$
$$2541$$

Octal Number System

The octal number system has base 8(means it has only eight digits from 0 to 7). There are only eight possible digit values to represent a number. With the help of only three bits, an octal number is represented. Each set of bits has a distinct value between 0 and 7.

Below, we have described certain characteristics of the octal number system:

Characteristics:

1. An octal number system carries eight digits starting from 0, 1, 2, 3, 4, 5, 6, and 7.
2. It is also known as the base 8 number system.
3. The position of a digit represents the 0 power of the base(8). Example: 8^0
4. The position of the last digit represents the x power of the base(8). Example: 8^x, where x represents the last position, i.e., 1

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

Examples:

$(273)_8$, $(5644)_8$, $(0.5365)_8$, $(1123)_8$, $(1223)_8$.

Hexadecimal Number System

It is another technique to represent the number in the digital system called the **hexadecimal number system**. The number system has a base of 16 means there are total 16 symbols(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) used for representing a number. The single-bit representation of decimal values10, 11, 12, 13, 14, and 15 are represented by A, B, C, D, E, and F. Only 4 bits are required for representing a number in a hexadecimal number. Each set of bits has a distinct value between 0 and 15. There are the following characteristics of the octal number system:

Characteristics:

1. It has ten digits from 0 to 9 and 6 letters from A to F.
2. The letters from A to F defines numbers from 10 to 15.
3. It is also known as the base 16number system.
4. In hexadecimal number, the position of a digit represents the 0 power of the base(16). Example: 16^0
5. In hexadecimal number, the position of the last digit represents the x power of the base(16). Example: 16^x, where x represents the last position, i.e., 1

Binary Number	Hexadecimal Number
0000	0
0001	1
0010	2

0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Examples:

$(FAC2)_{16}$, $(564)_{16}$, $(0ABD5)_{16}$, $(1123)_{16}$, $(11F3)_{16}$.

Number Base Conversion

Types of number systems are binary, decimal, octal, and hexadecimal. In this part, we will learn how we can change a number from one number system to another number system.

As, we have four types of number systems so each one can be converted into the remaining three systems. There are the following conversions possible in Number System

1. Binary to other Number Systems.
2. Decimal to other Number Systems.
3. Octal to other Number Systems.
4. Hexadecimal to other Number Systems.

Binary to other Number Systems

There are three conversions possible for binary number, i.e., binary to decimal, binary to octal, and binary to hexadecimal. The conversion process of a binary number to decimal differs from the remaining others. Let's take a detailed discussion on Binary Number System conversion.

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.

Let's take an example to understand how the conversion is done from binary to decimal.

Example: $(10110.001)_2$

We multiplied each bit of $(10110.001)_2$ with its respective positional weight, and last we add the products of all the bits with its weight.

$(10110.001)_2=(1\times24)+(0\times23)+(1\times22)+(1\times21)+(0\times20)+$
$(0\times2-1)+(0\times2-2)+(1\times2-3)$
$(10110.001)_2=(1\times16)+(0\times8)+(1\times4)+(1\times2)+(0\times1)+$
$(0\times1/2)+(0\times1/4)+(1\times1/8)$
$(10110.001)_2=16+0+4+2+0+0+0+0.125$
$(10110.001)_2=(22.125\ )_{10}$

Binary to Octal Conversion

The base numbers of binary and octal are 2 and 8, respectively. In a binary number, the pair of three bits is equal to one octal digit. There are only two steps to convert a binary number into an octal number which are as follows:

1. In the first step, we have to make the pairs of three bits on both sides of the binary point. If there will be one or two bits left in a pair of three bits pair, we add the required number of zeros on extreme sides.
2. In the second step, we write the octal digits corresponding to each pair.

Example: $(111110101011.0011)_2$

1. Firstly, we make pairs of three 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 pair of three bits, we added two zeros on the extreme side.

111 110 101 011.001 100

2. Then, we wrote the octal digits, which correspond to each pair.

$(111110101011.0011)_2=(7653.14)_8$

Binary to Hexadecimal Conversion

The base numbers of binary and hexadecimal are 2 and 16, respectively. In a binary number, the pair of four bits is equal to one hexadecimal digit. There are also only two steps to convert a binary number into a hexadecimal number which are as follows:

1. In the first step, we have to make the pairs of four bits on both sides of the binary point. If there will be one, two, or three bits left in a pair of four bits pair, we add the required number of zeros on extreme sides.
2. In the second step, we write the hexadecimal digits corresponding to each pair.

Example: $(10110101011.0011)_2$

1. Firstly, we make pairs of four bits on both sides of the binary point.

111 1010 1011.0011

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

0111 1010 1011.0011

2. Then, we write the hexadecimal digits, which correspond to each pair.

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

<u>Decimal to other Number System</u>

The decimal number can be an integer or floating-point integer. When the decimal number is a floating-point integer, then we convert both part (integer and fractional) of the decimal number in the isolated form(individually). There are the following steps that are used to convert the decimal number into a similar number of any base **'r'**.

1. In the first step, we perform the division operation on integer and successive part with base **'r'**. We will list down all the remainders till the quotient is zero. Then we find out the remainders in reverse order for getting the integer part of the equivalent number of base **'r'**. In this, the least and most significant digits are denoted by the first and the last remainders.
2. In the next step, the multiplication operation is done with base **'r'** of the fractional and successive fraction. The carries are noted until the result is zero or when the required number of the equivalent digit is obtained. For getting the fractional part of the equivalent number of base **'r'**, the normal sequence of carrying is considered.

<u>Decimal to Binary Conversion</u>

For converting decimal to binary, there are two steps required to perform, which are as follows:

1. In the first step, we perform the division operation on the integer and the successive quotient with the base of binary(2).

2. Next, we perform the multiplication on the integer and the successive quotient with the base of binary(2).

Example: $(152.25)_{10}$

Step 1:

Divide the number 152 and its successive quotients with 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
½	0	1(MSB)

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

Step 2:

Now, perform the multiplication of 0.27 and successive fraction with base 2.

Operation	**Result**	**carry**
0.25×2	0.50	0
0.50×2	0	1

$(0.25)_{10}$=$(.01)_2$

<u>Decimal to Octal Conversion</u>

For converting decimal to octal, there are two steps required to perform, which are as follows:

1. In the first step, we perform the division operation on the integer and the successive quotient with the base of octal(8).
2. Next, we perform the multiplication on the integer and the successive quotient with the base of octal(8).

Example: $(152.25)_{10}$

Step 1:

Divide the number 152 and its successive quotients with base 8.

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

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

Step 2:

Now perform the multiplication of 0.25 and successive fraction with base 8.

Operation	**Result**	**carry**
0.25×8	**0**	**2**

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

So, the octal number of the decimal number 152.25 is **230.2**

<u>Decimal to hexadecimal conversion</u>

For converting decimal to hexadecimal, there are two steps required to perform, which are as follows:

1. In the first step, we perform the division operation on the integer and the successive quotient with the base of hexadecimal (16).
2. Next, we perform the multiplication on the integer and the successive quotient with the base of hexadecimal (16).

Example: $(152.25)_{10}$

Step 1:

Divide the number 152 and its successive quotients with base 8.

Operation	**Quotient**	**Remainder**
152/16	9	8

9/16	0	9

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

Step 2:

Now perform the multiplication of 0.25 and successive fraction with base 16.

Operation	**Result**	**carry**
0.25×16	0	4

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

So, the hexadecimal number of the decimal number 152.25 is **230.4.**

<u>Octal to other Number System</u>

Like binary and decimal, the octal number can also be converted into other number systems. The process of converting octal to decimal differs from the remaining one. Let's start understanding how conversion is done.

<u>Octal to Decimal Conversion</u>

The process of converting octal to decimal is the same as binary to decimal. The process starts from multiplying the digits of octal numbers with its corresponding positional weights. And lastly, we add all those products.

Let's take an example to understand how the conversion is done from octal to decimal.

Example: $(152.25)_8$

Step 1:

We multiply each digit of **152.25** with its respective positional weight, and last we add the products of all the bits with its weight.

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

So, the decimal number of the octal number 152.25 is **106.328125**

<u>Octal to Binary Conversion</u>

The process of converting octal to binary is the reverse process of binary to octal. We write the three bits binary code of each octal number digit.

Example : $(152.25)_8$

We write the three-bit binary digit for 1, 5, 2, and 5.

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

So, the binary number of the octal number 152.25 is **$(001101010.010101)_2$**

Octal to hexadecimal conversion

For converting octal to hexadecimal, there are two steps required to perform, which are as follows:

1. In the first step, we will find the binary equivalent of number **25**.
2. Next, we have to make the pairs of four bits on both sides of the binary point. If there will be one, two, or three bits left in a pair of four bits pair, we add the required number of zeros on extreme sides and write the hexadecimal digits corresponding to each pair.

Example: $(152.25)_8$

Step 1:

We write the three-bit binary digit for 1, 5, 2, and 5.

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

So, the binary number of the octal number 152.25 is **$(001101010.010101)_2$**

Step 2:

1. Now, we make pairs of four bits on both sides of the binary point.

0 0110 1010.0101 01

On the left side of the binary point, the first pair has only one digit, and on the right side, the last pair has only two-digit. To make them complete pairs of four bits, add zeros on extreme sides.

0000 0110 1010.0101 0100

2. Now, we write the hexadecimal digits, which correspond to each pair.

$(0000\ 0110\ 1010.0101\ 0100)_2=(6A.54)_{16}$

Hexa-decimal to other Number System

Like binary, decimal, and octal, hexadecimal numbers can also be converted into other number systems. The process of converting hexadecimal to decimal differs from the remaining one. Let's start understanding how conversion is done.

Hexa-decimal to Decimal Conversion

The process of converting hexadecimal to decimal is the same as binary to decimal. The process starts from multiplying the digits of hexadecimal numbers with its corresponding positional weights. And lastly, we add all those products.

Let's take an example to understand how the conversion is done from hexadecimal to decimal.

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

Step 1:

We multiply each digit of **152A.25** with its respective positional weight, and last we add the products of all the bits with its weight.

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

So, the decimal number of the hexadecimal number 152A.25 is **5418.14453125**

Hexadecimal to Binary Conversion

The process of converting hexadecimal to binary is the reverse process of binary to hexadecimal. We write the four bits binary code of each hexadecimal number digit.

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

We write the four-bit binary digit for 1, 5, A, 2, and 5.

$\mathbf{(152A.25)_{16}=(0001\ 0101\ 0010\ 1010.0010\ 0101)_2}$

So, the binary number of the hexadecimal number 152.25 is $\mathbf{(1010100101010.00100101)_2}$

Hexadecimal to Octal Conversion

For converting hexadecimal to octal, there are two steps required to perform, which are as follows:

1. In the first step, we will find the binary equivalent of the hexadecimal number.

2. Next, we have to make the pairs of three bits on both sides of the binary point. If there will be one or two bits left in a pair of three bits pair, we add the required number of zeros on extreme sides and write the octal digits corresponding to each pair.

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

Step 1:

We write the four-bit binary digit for 1, 5, 2, A, and 5.

$(152A.25)_{16}$=$(0001\ 0101\ 0010\ 1010.0010\ 0101)_2$

So, the binary number of hexadecimal number 152A.25 is **$(0011010101010.010101)_2$**

Step 2:

3. Then, we make pairs of three bits on both sides of the binary point.

001 010 100 101 010.001 001 010

4. Then, we write the octal digit, which corresponds to each pair.

$(001010100101010.001001010)_2$=$(12452.112)_8$

So, the octal number of the hexadecimal number 152A.25 is **12452.112**

Gray Code

The **Gray Code** is a sequence of binary number systems, which is also known as **reflected binary code**. The reason for calling this code as reflected binary code is the first N/2 values compared with those of the last N/2 values in reverse order. In this code, two consecutive values are differed by one bit of binary digits. Gray codes are used in the general sequence of hardware-generated binary numbers. These numbers cause ambiguities or errors when the transition from one number to its successive is done. This code simply solves this problem by changing only one bit when the transition is between numbers is done.

The gray code is a very light weighted code because it doesn't depend on the value of the digit specified by the position. This code is also called a cyclic variable code as the transition of one value to its successive value carries a change of one bit only.

How to generate Gray code?

The prefix and reflect method are recursively used to generate the Gray code of a number. For generating gray code:

1. We find the number of bits required to represent a number.
2. Next, we find the code for 0, i.e., 0000, which is the same as binary.
3. Now, we take the previous code, i.e., 0000, and change the most significant bit of it.
4. We perform this process reclusively until all the codes are not uniquely identified.

5. If by changing the most significant bit, we find the same code obtained previously, then the second most significant bit will be changed, and so on.

Gray Code Table

Decimal Number	Binary Number	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

Excess-3 Code

The excess-3 code is also treated as **XS-3 code**. The excess-3 code is a non-weighted and self-complementary BCD code used to represent the decimal numbers. This code has a biased representation. This code plays an important role in arithmetic operations because it resolves deficiencies encountered when we use the 8421 BCD code for adding two decimal digits whose sum is greater than 9. The Excess-3 code uses a special type of algorithm, which differs from the binary positional number system or normal non-biased BCD.

We can easily get an excess-3 code of a decimal number by simply adding 3 to each decimal digit. And then we write the 4-bit binary number for each digit of the decimal number. We can find the excess-3 code of the given binary number by using the following steps:

1. We find the decimal number of the given binary number.
2. Then we add 3 in each digit of the decimal number.
3. Now, we find the binary code of each digit of the newly generated decimal number.

We can also add 0011 in each 4-bit BCD code of the decimal number for getting excess-3 code.

The Excess-3 code for the decimal number is as follows:

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

In excess-3 code, the codes 1111 and 0000 are never used for any decimal digit. Let's take some examples of Excess-3 code.

Example: Decimal number 31

1. We find the BCD code of each digit of the decimal number.

Digit	BCD
3	0011
1	0001

2) Then, we add 0011 in both of the BCD code.

Decimal	BCD	Excess-3
3	0011+0011	0110

1	0001+0011	0100

3. So, the excess-3 code of the decimal number 31 is **0110 0100**

Example: Decimal number 81.61

1. We find the BCD code of each digit of the decimal number.

Digit	BCD
8	1000
1	0001
6	0110
1	0001

2) Then, we add 0011 in both of the BCD code.

Decimal	BCD	Excess-3
8	1000+0011	1011
1	0001+0011	0100
6	0110+0011	1001

3) So, the excess-3 code of the decimal number 81.61 is **1011 0100.1001 0100**

<u>ASCII CODE</u>

The ASCII stands for American Standard Code for Information Interchange. The ASCII code is an alphanumeric code used for data communication in digital computers. The ASCII is a 7-bit code capable of representing 2^7 or 128 number of different characters. The ASCII code is made up of a three-bit group, which is followed by a four-bit code.

- The ASCII Code is a 7 or 8-bit alphanumeric code.
- This code can represent 127 unique characters.
- The ASCII code starts from 00h to 7Fh. In this, the code from 00h to 1Fh is used for control characters, and the code from 20h to 7Fh is used for graphic symbols.
- The 8-bit code holds ASCII, which supports 256 symbols where math and graphic symbols are added.
- The range of the extended ASCII is 80h to FFh.

<u>Basic Gates</u>

The basic gates are AND, OR & NOT gates.

AND gate

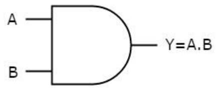
An AND gate is a digital circuit that has two or more inputs and produces an output, which is the **logical AND** of all those inputs. It is optional to represent the **Logical AND** with the symbol '.'.

The following table shows the **truth table** of 2-input AND gate.

A	B	Y = A.B
0	0	0
0	1	0
1	0	0
1	1	1

Here A, B are the inputs and Y is the output of two input AND gate. If both inputs are '1', then only the output, Y is '1'. For remaining combinations of inputs, the output, Y is '0'.

The following figure shows the **symbol** of an AND gate, which is having two inputs A, B and one output, Y.



This AND gate produces an output (Y), which is the **logical AND** of two inputs A, B. Similarly, if there are 'n' inputs, then the AND gate produces an output, which is the logical AND of all those inputs. That means, the output of AND gate will be '1', when all the inputs are '1'.

OR gate

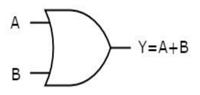
An OR gate is a digital circuit that has two or more inputs and produces an output, which is the logical OR of all those inputs. This **logical OR** is represented with the symbol '+'.

The following table shows the **truth table** of 2-input OR gate.

A	B	Y = A + B
0	0	0
0	1	1
1	0	1
1	1	1

Here A, B are the inputs and Y is the output of two input OR gate. If both inputs are '0', then only the output, Y is '0'. For remaining combinations of inputs, the output, Y is '1'.

The following figure shows the **symbol** of an OR gate, which is having two inputs A, B and one output, Y.



This OR gate produces an output (Y), which is the **logical OR** of two inputs A, B. Similarly, if there are 'n' inputs, then the OR gate produces an output, which is the logical OR of all those inputs. That means, the output of an OR gate will be '1', when at least one of those inputs is '1'.

NOT gate

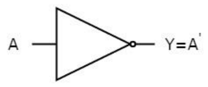
A NOT gate is a digital circuit that has single input and single output. The output of NOT gate is the **logical inversion** of input. Hence, the NOT gate is also called as inverter.

The following table shows the **truth table** of NOT gate.

A	Y = A'
0	1
1	0

Here A and Y are the input and output of NOT gate respectively. If the input, A is '0', then the output, Y is '1'. Similarly, if the input, A is '1', then the output, Y is '0'.

The following figure shows the **symbol** of NOT gate, which is having one input, A and one output, Y.



This NOT gate produces an output (Y), which is the **complement** of input, A.

Universal gates

NAND & NOR gates are called as **universal gates**. Because we can implement any Boolean function, which is in sum of products form by using NAND gates alone. Similarly, we can implement any Boolean function, which is in product of sums form by using NOR gates alone.

NAND gate

NAND gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical AND** of all those inputs.

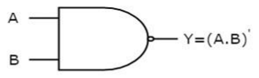
The following table shows the **truth table** of 2-input NAND gate.

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

0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input NAND gate. When both inputs are '1', the output, Y is '0'. If at least one of the input is zero, then the output, Y is '1'. This is just opposite to that of two input AND gate operation.

The following image shows the **symbol** of NAND gate, which is having two inputs A, B and one output, Y.



NAND gate operation is same as that of AND gate followed by an inverter. That's why the NAND gate symbol is represented like that.

NOR gate

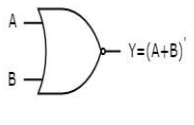
NOR gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical OR** of all those inputs.

The following table shows the **truth table** of 2-input NOR gate

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

Here A, B are the inputs and Y is the output. If both inputs are '0', then the output, Y is '1'. If at least one of the input is '1', then the output, Y is '0'. This is just opposite to that of two input OR gate operation.

The following figure shows the **symbol** of NOR gate, which is having two inputs A, B and one output, Y.



NOR gate operation is same as that of OR gate followed by an inverter. That's why the NOR gate symbol is represented like that.

REVIEW QUESTIONS

1) Define Ascii code.

2) Define gray code.
3) Explain binary to decimal conversion.
4) Define hexadecimal.
5) Explain about universal logic gates.
6) Define excess-3 code.
7) Define AND gate.
8) Explain binary to octal conversion.

UNIT – II

Combinational Logic Circuits

- **Boolean Laws and Theorems**
- **Sum of Product Method**
- **Karnaugh Map**
- **Pair, Quad and Octet**
- **Don't Care Condition**
- **Product of Sum Method**
- **Product of Sum Simplification**

Data Processing Circuit

- **Multiplexer**
- **De-Multiplexer**
- **1-of-16 Decoder**
- **BCD to Decimal Decoder**
- **Seven Segment Decoders**
- **Encoders**
- **Exclusive OR Gates**
- **Parity Generator and checker**

COMBINATIONAL LOGIC CIRCUIT

Boolean Laws and Theorems

Boolean Algebra is an algebra, which deals with binary numbers & binary variables. Hence, it is also called as Binary Algebra or logical Algebra. A mathematician, named George Boole had developed this algebra in 1854. The variables used in this algebra are also called as Boolean variables.

The range of voltages corresponding to Logic 'High' is represented with '1' and the range of voltages corresponding to logic 'Low' is represented with '0'.

Postulates and Basic Laws of Boolean Algebra

In this section, let us discuss about the Boolean postulates and basic laws that are used in Boolean algebra. These are useful in minimizing Boolean functions.

Boolean Postulates

Consider the binary numbers 0 and 1, Boolean variable (x) and its complement (x’). Either the Boolean variable or complement of it is known as **literal**. The four possible **logical OR** operations among these literals and binary numbers are shown below.

x + 0 = x

x + 1 = 1

x + x = x

x + x’ = 1

Similarly, the four possible **logical AND** operations among those literals and binary numbers are shown below.

x.1 = x

x.0 = 0

x.x = x

x.x’ = 0

These are the simple Boolean postulates. We can verify these postulates easily, by substituting the Boolean variable with ‘0’ or ‘1’.

Note− The complement of complement of any Boolean variable is equal to the variable itself. i.e., (x’)’=x.

<u>Basic Laws of Boolean Algebra</u>

Following are the three basic laws of Boolean Algebra.

- Commutative law
- Associative law
- Distributive law

<u>Commutative Law</u>

If any logical operation of two Boolean variables give the same result irrespective of the order of those two variables, then that logical operation is said to be **Commutative**. The logical OR & logical AND operations of two Boolean variables x & y are shown below

x + y = y + x

x.y = y.x

The symbol ‘+’ indicates logical OR operation. Similarly, the symbol ‘.’ indicates logical AND operation and it is optional to represent. Commutative law obeys for logical OR & logical AND operations.

<u>Associative Law</u>

If a logical operation of any two Boolean variables is performed first and then the same operation is performed with the remaining variable gives the same result, then that logical operation is said to be **Associative**. The logical OR & logical AND operations of three Boolean variables x, y & z are shown below.

x + (y + z) = (x + y) + z

x.(y.z) = (x.y).z

Associative law obeys for logical OR & logical AND operations.

Distributive Law

If any logical operation can be distributed to all the terms present in the Boolean function, then that logical operation is said to be **Distributive**. The distribution of logical OR & logical AND operations of three Boolean variables x, y & z are shown below.

x.(y + z) = x.y + x.z

x + (y.z) = (x + y).(x + z)

Distributive law obeys for logical OR and logical AND operations.

These are the Basic laws of Boolean algebra. We can verify these laws easily, by substituting the Boolean variables with ‘0’ or ‘1’.

Theorems of Boolean Algebra

The following two theorems are used in Boolean algebra.

- Duality theorem
- DeMorgan’s theorem

Duality Theorem

This theorem states that the **dual** of the Boolean function is obtained by interchanging the logical AND operator with logical OR operator and zeros with ones. For every Boolean function, there will be a corresponding Dual function.

Let us make the Boolean equations (relations) that we discussed in the section of Boolean postulates and basic laws into two groups. The following table shows these two groups.

Group1	Group2
x + 0 = x	x.1 = x
x + 1 = 1	x.0 = 0
x + x = x	x.x = x
x + x’ = 1	x.x’ = 0

$x + y = y + x$	$x.y = y.x$
$x + (y + z) = (x + y) + z$	$x.(y.z) = (x.y).z$
$x.(y + z) = x.y + x.z$	$x + (y.z) = (x + y).(x + z)$

In each row, there are two Boolean equations and they are dual to each other. We can verify all these Boolean equations of Group1 and Group2 by using duality theorem.

DeMorgan's Theorem

This theorem is useful in finding the **complement of Boolean function**. It states that the complement of logical OR of at least two Boolean variables is equal to the logical AND of each complemented variable.

DeMorgan's theorem with 2 Boolean variables x and y can be represented as

$$\mathbf{(x + y)' = x'.y'}$$

The dual of the above Boolean function is

$$\mathbf{(x.y)' = x' + y'}$$

Therefore, the complement of logical AND of two Boolean variables is equal to the logical OR of each complemented variable. Similarly, we can apply DeMorgan's theorem for more than 2 Boolean variables also.

Simplification of Boolean Functions

Till now, we discussed the postulates, basic laws and theorems of Boolean algebra. Now, let us simplify some Boolean functions.

Eg

Let us **simplify** the Boolean function, f = p'qr + pq'r + pqr' + pqr

We can simplify this function in two methods.

Method 1

Given Boolean function, f = p'qr + pq'r + pqr' +pqr.

Step 1 − In first and second terms r is common and in third and fourth terms pq is common. So, take the common terms by using **Distributive law**.

$$\Rightarrow f = (p'q + pq')r + pq(r' + r)$$

Step 2 − The terms present in first parenthesis can be simplified to Ex-OR operation. The terms present in second parenthesis can be simplified to '1' using **Boolean postulate**

$$\Rightarrow f = (p \oplus q)r + pq(1)$$

Step 3 − The first term can't be simplified further. But, the second term can be simplified to pq using **Boolean postulate**.

$$\Rightarrow f = (p \oplus q)r + pq$$

Therefore, the simplified Boolean function is **f = (p⊕q)r + pq**

Method 2

Given Boolean function, f = p’qr + pq’r + pqr’ + pqr.

Step 1 − Use the **Boolean postulate**, x + x = x. That means, the Logical OR operation with any Boolean variable ‘n’ times will be equal to the same variable. So, we can write the last term pqr two more times.

$$\Rightarrow f = p'qr + pq'r + pqr' + pqr + pqr + pqr$$

Step 2 − Use **Distributive law** for 1st and 4th terms, 2nd and 5th terms, 3rd and 6th terms.

$$\Rightarrow f = qr(p' + p) + pr(q' + q) + pq(r' + r)$$

Step 3 − Use **Boolean postulate**, x + x’ = 1 for simplifying the terms present in each parenthesis.

$$\Rightarrow f = qr(1) + pr(1) + pq(1)$$

Step 4 − Use **Boolean postulate**, x.1 = x for simplifying the above three terms.

$$\Rightarrow f = qr + pr + pq$$

$$\Rightarrow f = pq + qr + pr$$

Therefore, the simplified Boolean function is **f = pq + qr + pr**.

So, we got two different Boolean functions after simplifying the given Boolean function in each method. Functionally, those two Boolean functions are same. So, based on the requirement, we can choose one of those two Boolean functions.

Eg

Let us find the **complement** of the Boolean function, f = p’q + pq’.

The complement of Boolean function is f’ = (p’q + pq’)’.

Step 1 − Use DeMorgan’s theorem, (x + y)’ = x’.y’.

$$\Rightarrow f' = (p'q)'.(pq')'$$

Step 2 − Use DeMorgan’s theorem, (x.y)’ = x’ + y’

$$\Rightarrow f' = \{(p')' + q'\}.\{p' + (q')'\}$$

Step3 − Use the Boolean postulate, (x’)’=x.

$$\Rightarrow f' = \{p + q'\}.\{p' + q\}$$

$$\Rightarrow f' = pp' + pq + p'q' + qq'$$

Step 4 − Use the Boolean postulate, xx’=0.

$$\Rightarrow f = 0 + pq + p'q' + 0$$

$$\Rightarrow f = pq + p'q'$$

Therefore, the **complement** of Boolean function, p’q + pq’ is **pq + p’q’**.

Sum of product Method(SOP)

A canonical sum of products is a boolean expression that entirely consists of minterms. The Boolean function F is defined on two variables X and Y. The X and Y are the inputs of the boolean function F whose output is true when any one of the inputs is set to true. The truth table for Boolean expression F is as follows:

Inputs		**Output**
X	**Y**	**F**
0	0	0
0	1	1
1	0	1
1	1	1

we can form the minterm from the variable's value. Now, a column will be added for the minterm in the above table. The complement of the variables is taken whose value is 0, and the variables whose value is 1 will remain the same.

Inputs		**Output**	**Minterm**
X	**Y**	**F**	**M**
0	0	0	X'Y'
0	1	1	X'Y
1	0	1	XY'
1	1	1	XY

Now, we will add all the minterms for which the output is true to find the desired canonical SOP(Sum of Product) expression.

F=X' Y+XY'+XY

Converting Sum of Products (SOP) to shorthand notation

The process of converting SOP form to shorthand notation is the same as the process of finding shorthand notation for minterms. There are the following steps to find the shorthand notation of the given SOP expression.

- Write the given SOP expression.
- Find the shorthand notation of all the minterms.
- Replace the minterms with their shorthand notations in the given expression.

Example: F = X'Y+XY'+XY

1. Firstly, we write the SOP expression:

F = X'Y+XY'+XY

2. Now, we find the shorthand notations of the minterms X'Y, XY', and XY.

X'Y = $(01)_2$ = m_1

XY' = $(10)_2$ = m_2

XY = $(11)_2$ = m_3

3. In the end, we replace all the minterms with their shorthand notations:

F=m1+m2+m3

Converting shorthand notation to SOP expression

The process of converting shorthand notation to SOP is the reverse process of converting SOP expression to shorthand notation. Let's see an example to understand this conversion.

Example:

Let us assume that we have a boolean function F, which defined on two variables X and Y. The minterms for the function F are expressed as shorthand notation is as follows:

F=∑(1,2,3)

Now, from this expression, we will find the SOP expression. The Boolean function F has two input variables X and y and the output of F=1 for m1, m2, and m3, i.e., 1st, 2nd, and 3rd combinations. So,

F=∑(1,2,3)

F= m1 + m2 + m3

F= 01 + 10 + 11

Now, we replace zeros with either X' or Y' and ones with either X or Y. Simply, the complement variable is used when the variable value is 1 otherwise the non-complement variable is used.

F = ∑(1,2,3)

F=01+10+11

F= A'B + AB' + AB

Karnaugh Map(K-Map) method

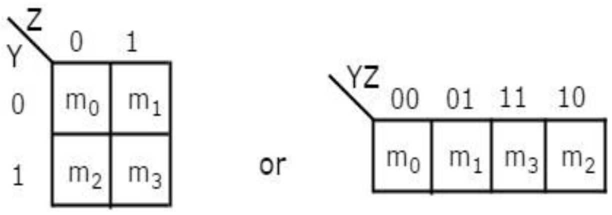
Karnaugh introduced a method for simplification of Boolean functions in an easy way. This method is known as Karnaugh map method or K-map method. It is a graphical method, which consists of 2^n cells for 'n' variables. The adjacent cells are differed only in single bit position.

K-Maps for 2 to 5 Variables

K-Map method is most suitable for minimizing Boolean functions of 2 variables to 5 variables. Now, let us discuss about the K-Maps for 2 to 5 variables one by one.

2 Variable K-Map

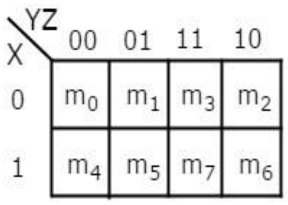
The number of cells in 2 variable K-map is four, since the number of variables is two. The following figure shows **2 variable K-Map**.



- There is only one possibility of grouping 4 adjacent min terms.
- The possible combinations of grouping 2 adjacent min terms are {(m_0, m_1), (m_2, m_3), (m_0, m_2) and (m_1, m_3)}.

3 Variable K-Map

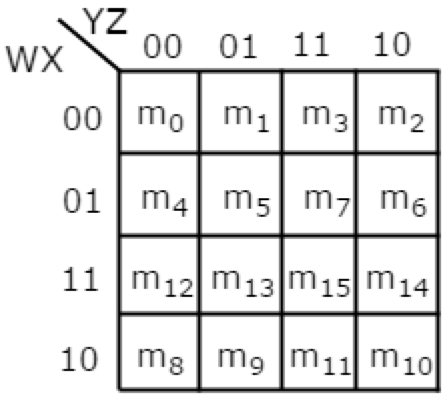
The number of cells in 3 variable K-map is eight, since the number of variables is three. The following figure shows **3 variable K-Map**.



- There is only one possibility of grouping 8 adjacent min terms.
- The possible combinations of grouping 4 adjacent min terms are {(m_0, m_1, m_3, m_2), (m_4, m_5, m_7, m_6), (m_0, m_1, m_4, m_5), (m_1, m_3, m_5, m_7), (m_3, m_2, m_7, m_6) and (m_2, m_0, m_6, m_4)}.
- The possible combinations of grouping 2 adjacent min terms are {(m_0, m_1), (m_1, m_3), (m_3, m_2), (m_2, m_0), (m_4, m_5), (m_5, m_7), (m_7, m_6), (m_6, m_4), (m_0, m_4), (m_1, m_5), (m_3, m_7) and (m_2, m_6)}.
- If x=0, then 3 variable K-map becomes 2 variable K-map.

4 Variable K-Map

The number of cells in 4 variable K-map is sixteen, since the number of variables is four. The following figure shows **4 variable K-Map**.

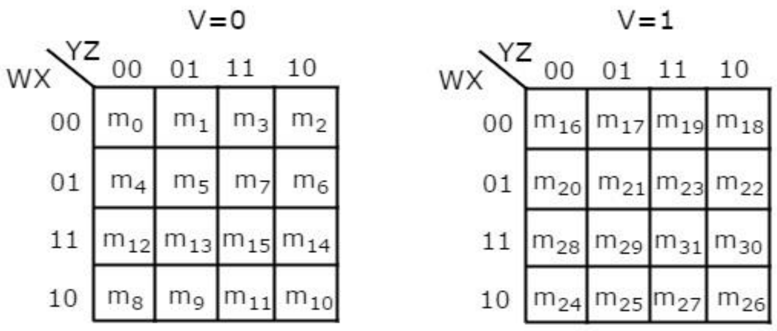


WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

- There is only one possibility of grouping 16 adjacent min terms.
- Let R_1, R_2, R_3 and R_4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C_1, C_2, C_3 and C_4 represents the min terms of first column, second column, third column and fourth column respectively. The possible combinations of grouping 8 adjacent min terms are $\{(R_1, R_2), (R_2, R_3), (R_3, R_4), (R_4, R_1), (C_1, C_2), (C_2, C_3), (C_3, C_4), (C_4, C_1)\}$.
- If w=0, then 4 variable K-map becomes 3 variable K-map.

5 Variable K-Map

The number of cells in 5 variable K-map is thirty-two, since the number of variables is 5. The following figure shows **5 variable K-Map**.



V=0

WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

V=1

WX \ YZ	00	01	11	10
00	m_{16}	m_{17}	m_{19}	m_{18}
01	m_{20}	m_{21}	m_{23}	m_{22}
11	m_{28}	m_{29}	m_{31}	m_{30}
10	m_{24}	m_{25}	m_{27}	m_{26}

- There is only one possibility of grouping 32 adjacent min terms.
- There are two possibilities of grouping 16 adjacent min terms. i.e., grouping of min terms from m_0 to m_{15} and m_{16} to m_{31}.
- If v=0, then 5 variable K-map becomes 4 variable K-map.

In the above all K-maps, we used exclusively the min terms notation. Similarly, you can use exclusively the Max terms notation.

Minimization of Boolean Functions using K-Maps

If we consider the combination of inputs for which the Boolean function is ‘1’, then we will get the Boolean function, which is in **standard sum of products** form after simplifying the K-map.

Similarly, if we consider the combination of inputs for which the Boolean function is ‘0’, then we will get the Boolean function, which is in **standard product of sums** form after simplifying the K-map.

Follow these **rules for simplifying K-maps** in order to get standard sum of products form.

- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as sum of min terms form, then place the ones at respective min term cells in the K-map. If the Boolean function is given as sum of products form, then place the ones in all possible cells of K-map for which the given product terms are valid.
- Check for the possibilities of grouping maximum number of adjacent ones. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.
- Each grouping will give either a literal or one product term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single ‘1’ is not covered with any other groupings but only that grouping covers.
- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

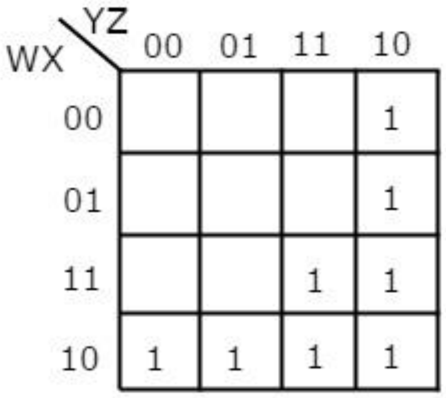
Note 1 − If outputs are not defined for some combination of inputs, then those output values will be represented with **don’t care symbol ‘x’**. That means, we can consider them as either ‘0’ or ‘1’.

Note 2 − If don’t care terms also present, then place don’t cares ‘x’ in the respective cells of K-map. Consider only the don’t cares ‘x’ that are helpful for grouping maximum number of adjacent ones. In those cases, treat the don’t care value as ‘1’.

<u>Eg</u>

Let us **simplify** the following Boolean function, **f(W, X, Y, Z)= WX’Y’ + WY + W’YZ’** using K-map.

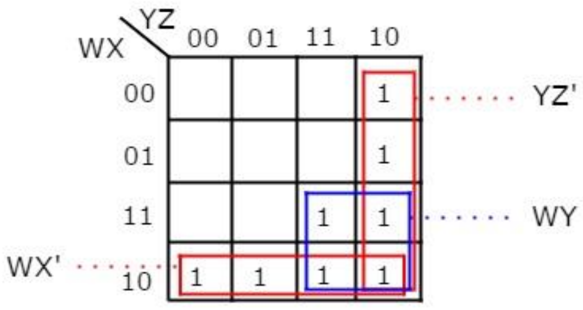
The given Boolean function is in sum of products form. It is having 4 variables W, X, Y & Z. So, we require **4 variable K-map**. The **4 variable K-map** with ones corresponding to the given product terms is shown in the following figure.



Here, 1s are placed in the following cells of K-map.

- The cells, which are common to the intersection of Row 4 and columns 1 & 2 are corresponding to the product term, **WX'Y'**.
- The cells, which are common to the intersection of Rows 3 & 4 and columns 3 & 4 are corresponding to the product term, **WY**.
- The cells, which are common to the intersection of Rows 1 & 2 and column 4 are corresponding to the product term, **W'YZ'**.

There are no possibilities of grouping either 16 adjacent ones or 8 adjacent ones. There are three possibilities of grouping 4 adjacent ones. After these three groupings, there is no single one left as ungrouped. So, we no need to check for grouping of 2 adjacent ones. The **4 variable K-map** with these three **groupings** is shown in the following figure.



Here, we got three prime implicants WX', WY & YZ'. All these prime implicants are **essential** because of following reasons.

- Two ones **(m_8 & m_9)** of fourth row grouping are not covered by any other groupings. Only fourth row grouping covers those two ones.
- Single one **(m_{15})** of square shape grouping is not covered by any other groupings. Only the square shape grouping covers that one.
- Two ones **(m_2 & m_6)** of fourth column grouping are not covered by any other groupings. Only fourth column grouping covers those two ones.

Therefore, the **simplified Boolean function** is

f = WX' + WY + YZ'

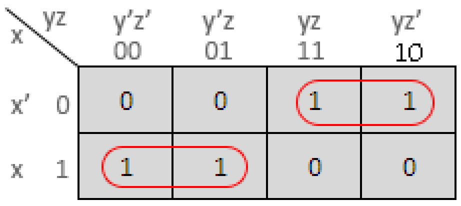
Follow these **rules for simplifying K-maps** in order to get standard product of sums form.

- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as product of Max terms form, then place the zeroes at respective Max term cells in the K-map. If the Boolean function is given as product of sums form, then place the zeroes in all possible cells of K-map for which the given sum terms are valid.
- Check for the possibilities of grouping maximum number of adjacent zeroes. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.
- Each grouping will give either a literal or one sum term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single '0' is not covered with any other groupings but only that grouping covers.
- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

Note − If don't care terms also present, then place don't cares 'x' in the respective cells of K-map. Consider only the don't cares 'x' that are helpful for grouping maximum number of adjacent zeroes. In those cases, treat the don't care value as '0'.

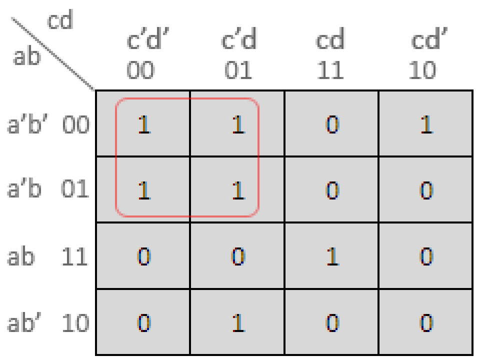
Pair, Quad and Octet

Pair Reduction Rule : Remove the variable which changes its state from complemented to uncomplemented or vice versa.Pair removes one variable only.



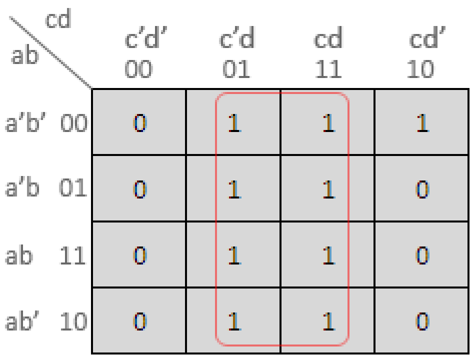
x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	0	0	1	1
x 1	1	1	0	0

Quad Reduction Rule : Remove the two variables which change their states.A quad removes two variables.



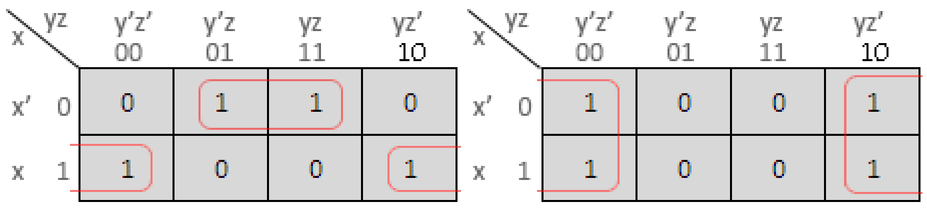
ab \ cd	c'd' 00	c'd 01	cd 11	cd' 10
a'b' 00	1	1	0	1
a'b 01	1	1	0	0
ab 11	0	0	1	0
ab' 10	0	1	0	0

Octet Reduction Rule : Remove the three variables which changes their state.Octet removes three variables.



ab \ cd	c'd' 00	c'd 01	cd 11	cd' 10
a'b' 00	0	1	1	1
a'b 01	0	1	1	0
ab 11	0	1	1	0
ab' 10	0	1	1	0

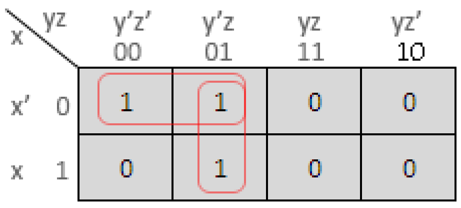
Map Rolling : Map rolling means roll the map considering the map as if its left edges are touching the right edges and top edges are touching bottom edges.While marking the pairs quads and octet, map must be rolled.



x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	0	1	1	0
x 1	1	0	0	1

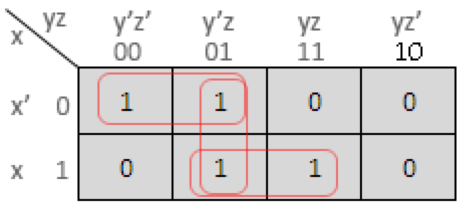
x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	1	0	0	1
x 1	1	0	0	1

Overlapping Groups : Overlapping means same 1 can be encircled more than once. Overlapping always leads to simpler expressions.



x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	1	1	0	0
x 1	0	1	0	0

Redundant Group : It is a group whose all 1's are overlapped by other groups. Redundant groups must be removed. Removal of redundant group leads to much simpler expression.



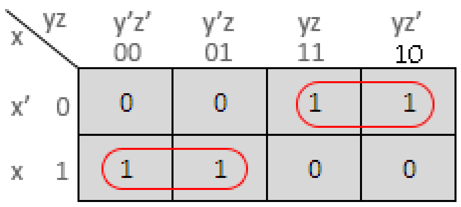
x \ yz	y'z' 00	y'z 01	yz 11	yz' 10
x' 0	1	1	0	0
x 1	0	1	1	0

Eg : Represent the following boolean expression in a K-map and simplify.

F = x'yz + x'yz' + xy'z' + xy'z

Solution :

The K-map is as follows :



Hence the simplified expression is

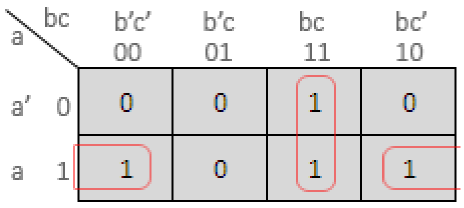
F = x'y + xy'

<u>Ex. 2</u> : Simplify the following boolean expression using K-map.

F = a'bc + ab'c' + abc + abc'

<u>Solution</u> :

The K-map is as follows :



Hence the simplified expression is

F = bc + ac'

<u>Don't Care Condition</u>

The "Don't Care" conditions allow us to replace the empty cell of a K-Map to form a grouping of the variables. While forming groups of cells, we can consider a "Don't Care" cell as either 1 or 0 or we can simply ignore that cell. Therefore, "Don't Care" condition can help us to form a larger group of cells.

A Don't Care cell can be represented by a cross(X) in K-Maps representing a invalid combination. For example, in Excess-3 code system, the states 0000, 0001, 0010, 1101, 1110 and 1111 are invalid or unspecified. These are called don't cares. Also, in design of 4-bit BCD-to-XS-3 code converter, the input combinations 1010, 1011, 1100, 1101, 1110, and 1111 are don't cares.

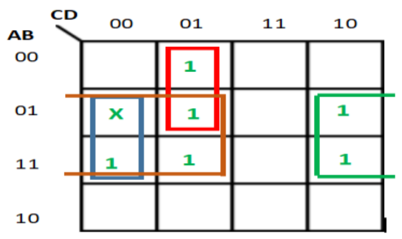
A standard SOP function having don't cares can be converted into a POS expression by keeping don't cares as they are, and writing the missing minterms of the SOP form as the maxterm of POS form. Similarly, a POS function having don't cares can be converted to SOP form keeping the don't cares as they are and write the missing maxterms of the POS expression as the minterms of SOP expression.

<u>Eg</u>

Minimise the following function in SOP minimal form using K-Maps:

f = m(1, 5, 6, 12, 13, 14) + d(4)

Explanation
The SOP K-map for the given expression is:



Therefore, SOP minimal is,

f = BC' + BD' + A'C'D

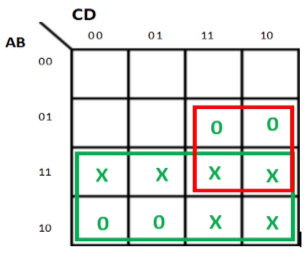
<u>Eg</u>
Minimise the following function in SOP minimal form using K-Maps:

F(A, B, C, D) = m(0, 1, 2, 3, 4, 5) + d(10, 11, 12, 13, 14, 15)

Explanation:
Writing the given expression in POS form:

F(A, B, C, D) = M(6, 7, 8, 9) + d(10, 11, 12, 13, 14, 15)

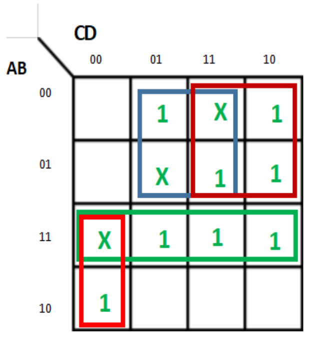
The POS K-map for the given expression is:



Therefore, POS minimal is,

F = A'(B' + C')

<u>Eg</u>
Minimise the following function in SOP minimal form using K-Maps: F(A, B, C, D) = m(1, 2, 6, 7, 8, 13, 14, 15) + d(3, 5, 12)
Explanation:
The SOP K-map for the given expression is:



Therefore,

f = AC'D' + A'D + A'C + AB

Product of Sum Method(POS)

A canonical product of sum is a boolean expression that entirely consists of maxterms. The Boolean function F is defined on two variables X and Y. The X and Y are the inputs of the boolean function F whose output is true when only one of the inputs is set to true. The truth table for Boolean expression F is as follows:

Inputs		**Output**
X	**Y**	**F**
0	0	0
0	1	1
1	0	1
1	1	0

In our minterm and maxterm section, we learned about how we can form the maxterm from the variable's value. A column will be added for the maxterm in the above table. The complement of the variables is taken whose value is 0, and the variables whose value is 1 will remain the same.

Inputs		**Output**	**Minterm**
X	**Y**	**F**	**M**
0	0	0	X'+Y'
0	1	1	X'+Y
1	0	1	X+Y'
1	1	1	X+Y

Now, we will multiply all the minterms for which the output is false to find the desired canonical POS(Product of sum) expression.

F=(X'+Y').(X+Y)

Converting Product of Sum (POS) to shorthand notation

The process of converting POS form to shorthand notation is the same as the process of finding shorthand notation for maxterms. There are the following steps used to find the shorthand notation of the given POS expression.

- Write the given POS expression.
- Find the shorthand notation of all the maxterms.
- Replace the minterms with their shorthand notations in the given expression.

Eg

F = (X'+Y').(X+Y)

1. Firstly, we will write the POS expression:

F = (X'+Y').(X+Y)

2. Now, we will find the shorthand notations of the maxterms X'+Y' and X+Y.

X'+Y' = $(00)_2$ = M_0

X+Y = $(11)_2$ = M_3

3. In the end, we will replace all the minterms with their shorthand notations:

F=M_0.M_3

Converting shorthand notation to POS expression

The process of converting shorthand notation to POS is the reverse process of converting POS expression to shorthand notation. Let's see an example to understand this conversion.

Eg

Let us assume that we have a boolean function F, defined on two variables X and Y. The maxterms for the function F are expressed as shorthand notation is as follows:

F=∏(1,2,3)

Now, from this expression, we find the POS expression. The Boolean function F has two input variables X and Y and the output of F=0 for M1, M2, and M3, i.e., 1st, 2nd, and 3rd combinations. So,

F=∏(1,2,3)

F= M1.M2.M3

F= 01.10.11

Next, we replace zeros with either X or Y and ones with either X' or Y'. Simply, if the value of the variable is 1, then we take the complement of that variable, and if the value of the variable is 0, then we take the variable "as is".

F = ∑(1,2,3)

F=01.10.11

F=(A+B').(A'+B).(A'+B')

Product of Sum Simplification

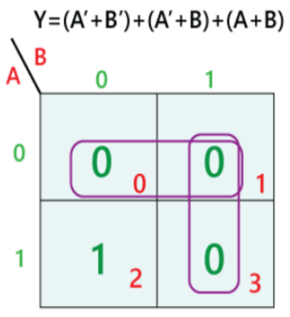
To find the simplified maxterm solution using K-map is the same as to find for the minterm solution. There are some minor changes in the maxterm solution, which are as follows:

1. We will populate the K-map by entering the value of 0 to each sum-term into the K-map cell and fill the remaining cells with one's.
2. We will make the groups of 'zeros' not for 'ones'.
3. Now, we will define the boolean expressions for each group as sum-terms.
4. At last, to find the simplified boolean expression in the POS form, we will combine the sum-terms of all individual groups.

Let's take some example of 2-variable, 3-variable, 4-variable and 5-variable K-map examples

Eg

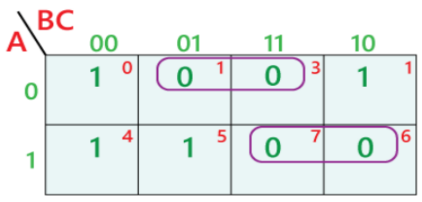
Y=(A'+B')+(A'+B)+(A+B)



Simplified expression: A'B

Eg

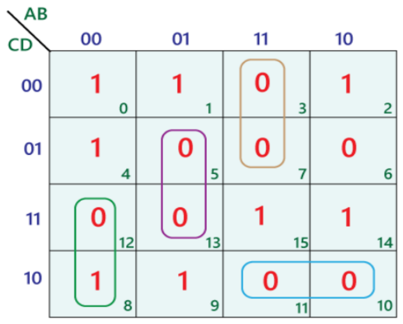
Y=(A + B + C') + (A + B' + C') + (A' + B' + C) + (A' + B' + C')



Simplified expression: Y=(A + C') .(A' + B')

Eg

F(A,B,C,D)=π(3,5,7,8,10,11,12,13)



Simplified expression: Y=(A + C') .(A' + B')

Data Processing Circuit

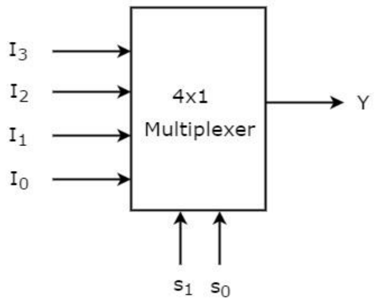
Multiplexer

Multiplexer is a combinational circuit that has maximum of 2^n data inputs, 'n' selection lines and single 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 zeros and ones. So, each combination will select only one data input. Multiplexer is also called as **Mux**.

4x1 Multiplexer

4x1 Multiplexer has four data inputs I_3, I_2, I_1 & I_0, two selection lines s_1 & s_0 and one output Y. The **block diagram** of 4x1 Multiplexer is shown in the following figure.



One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. **Truth table** of 4x1 Multiplexer is shown below.

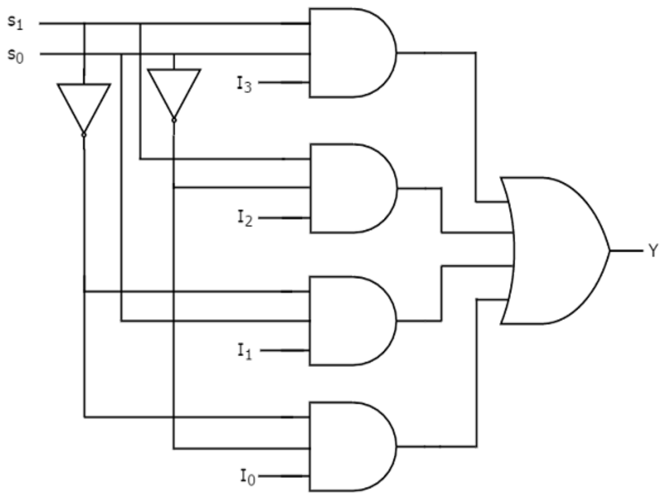
Selection Lines		**Output**
S_1	**S_0**	**Y**
0	0	I_0
0	1	I_1

1	0	I_2
1	1	I_3

From Truth table, we can directly write the **Boolean function** for output, Y as

Y={S{1}}'{S{0}}'I{0}+{S{1}}'S{0}I{1}+S{1}{S{0}}'I{2}+S{1}S{0}I{3}

We can implement this Boolean function using Inverters, AND gates & OR gate. The **circuit diagram** of 4x1 multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 8x1 Multiplexer and 16x1 multiplexer by following the same procedure.

Implementation of Higher-order Multiplexers.

Now, let us implement the following two higher-order Multiplexers using lower-order Multiplexers.

- 8x1 Multiplexer
- 16x1 Multiplexer

8x1 Multiplexer

In this section, let us implement 8x1 Multiplexer using 4x1 Multiplexers and 2x1 Multiplexer. We know that 4x1 Multiplexer has 4 data inputs, 2 selection lines and one output. Whereas, 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output.

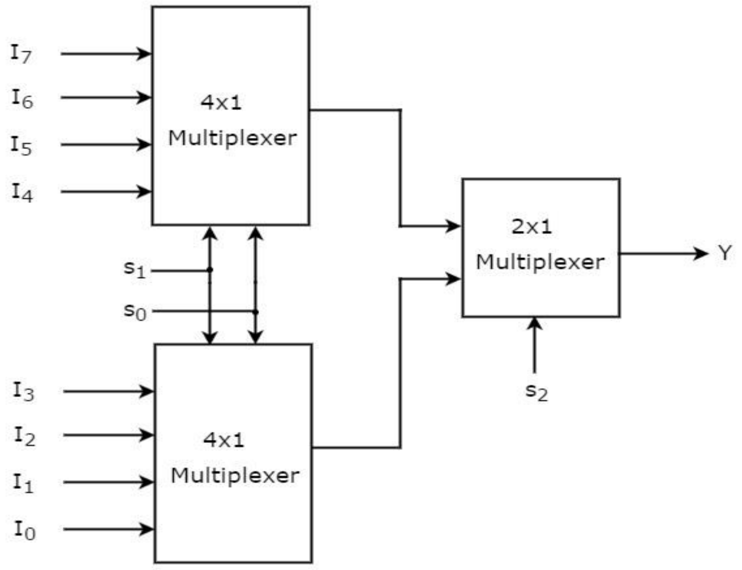
So, we require two **4x1 Multiplexers** in first stage in order to get the 8 data inputs. Since, each 4x1 Multiplexer produces one output, we require a **2x1 Multiplexer** in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 8x1 Multiplexer has eight data inputs I_7 to I_0, three selection lines s_2, s_1 & s0 and one output Y. The **Truth table** of 8x1 Multiplexer is shown below.

Selection Inputs			**Output**
S_2	**S_1**	**S_0**	**Y**

0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

We can implement 8x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 8x1 Multiplexer is shown in the following figure.



The same **selection lines, s_1 & s_0** are applied to both 4x1 Multiplexers. The data inputs of upper 4x1 Multiplexer are I_7 to I_4 and the data inputs of lower 4x1 Multiplexer are I_3 to I_0. Therefore, each 4x1 Multiplexer produces an output based on the values of selection lines, s_1 & s_0.

The outputs of first stage 4x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s_2** is applied to 2x1 Multiplexer.

- If s_2 is zero, then the output of 2x1 Multiplexer will be one of the 4 inputs I_3 to I_0 based on the values of selection lines s_1 & s_0.
- If s_2 is one, then the output of 2x1 Multiplexer will be one of the 4 inputs I_7 to I_4 based on the values of selection lines s_1 & s_0.

Therefore, the overall combination of two 4x1 Multiplexers and one 2x1 Multiplexer performs as one 8x1 Multiplexer.

16x1 Multiplexer

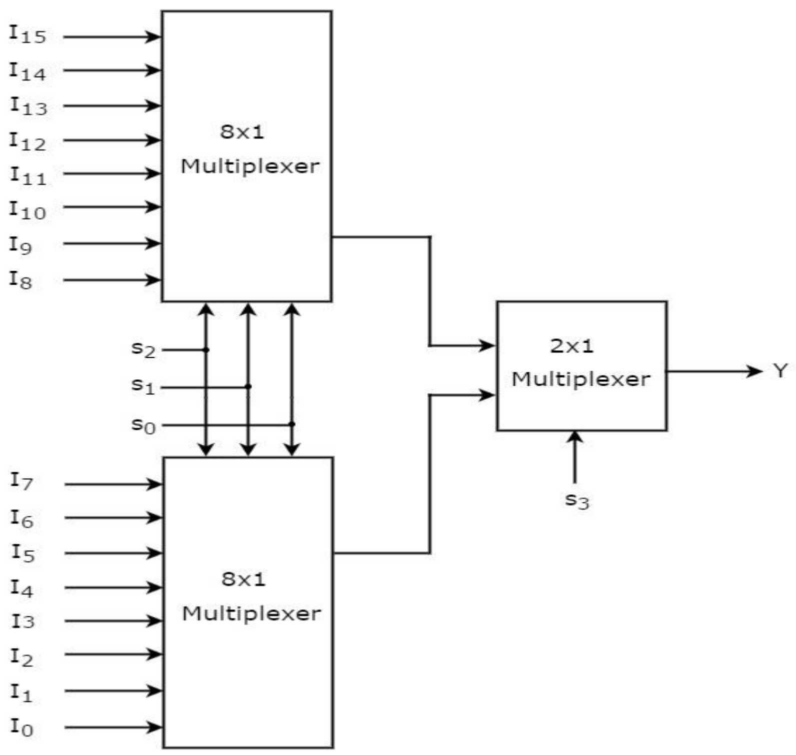
In this section, let us implement 16x1 Multiplexer using 8x1 Multiplexers and 2x1 Multiplexer. We know that 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output. Whereas, 16x1 Multiplexer has 16 data inputs, 4 selection lines and one output.

So, we require two **8x1 Multiplexers** in first stage in order to get the 16 data inputs. Since, each 8x1 Multiplexer produces one output, we require a 2x1 Multiplexer in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 16x1 Multiplexer has sixteen data inputs I_{15} to I_0, four selection lines s_3 to s_0 and one output Y. The **Truth table** of 16x1 Multiplexer is shown below.

Selection Inputs				**Output**
S_3	**S_2**	**S_1**	**S_0**	**Y**
0	0	0	0	I_0
0	0	0	1	I_1
0	0	1	0	I_2
0	0	1	1	I_3
0	1	0	0	I_4
0	1	0	1	I_5
0	1	1	0	I_6
0	1	1	1	I_7
1	0	0	0	I_8
1	0	0	1	I_9
1	0	1	0	I_{10}
1	0	1	1	I_{11}
1	1	0	0	I_{12}
1	1	0	1	I_{13}
1	1	1	0	I_{14}
1	1	1	1	I_{15}

We can implement 16x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 16x1 Multiplexer is shown in the following figure.



The **same selection lines, s_2, s_1 & s_0** are applied to both 8x1 Multiplexers. The data inputs of upper 8x1 Multiplexer are I_{15} to I_8 and the data inputs of lower 8x1 Multiplexer are I_7 to I_0. Therefore, each 8x1 Multiplexer produces an output based on the values of selection lines, s_2, s_1 & s_0.

The outputs of first stage 8x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s_3** is applied to 2x1 Multiplexer.

- If s_3 is zero, then the output of 2x1 Multiplexer will be one of the 8 inputs Is_7 to I_0 based on the values of selection lines s_2, s_1 & s_0.
- If s_3 is one, then the output of 2x1 Multiplexer will be one of the 8 inputs I_{15} to I_8 based on the values of selection lines s_2, s_1 & s_0.

Therefore, the overall combination of two 8x1 Multiplexers and one 2x1 Multiplexer performs as one 16x1 Multiplexer.

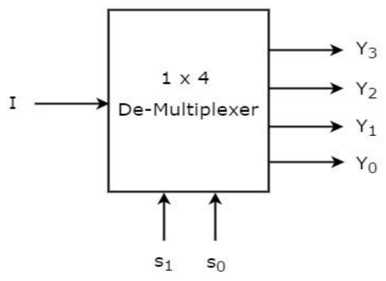
De-Multiplexer

De-Multiplexer is a combinational circuit that performs the reverse operation of 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 zeros and ones. So, each combination can select only one output. De-Multiplexer is also called as **De-Mux**.

1x4 De-Multiplexer

1x4 De-Multiplexer has one input I, two selection lines, s_1 & s_0 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 'I' will be connected to one of the four outputs, Y_3 to Y_0 based on the values of selection lines s_1 & s0. 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	**I**
0	1	0	0	**I**	0
1	0	0	**I**	0	0
1	1	**I**	0	0	0

From the above Truth table, we can directly write the **Boolean functions** for each output as

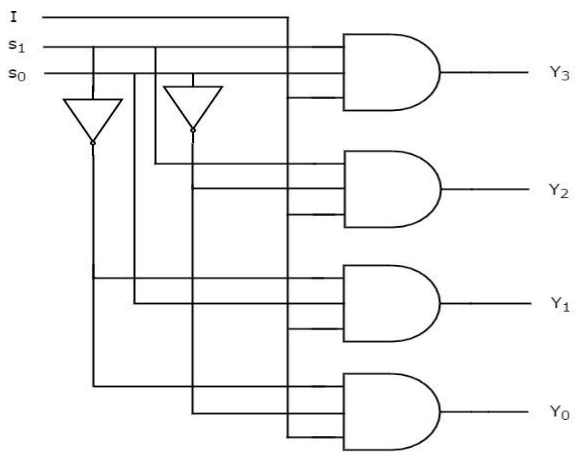
Y{3}=s{1}s{0}I

Y{2}=s{1}{s{0}}'I

Y{1}={s{1}}'s{0}I

Y{0}={s1}'{s{0}}'I

We can implement these Boolean functions using Inverters & 3-input AND gates. The **circuit diagram** of 1x4 De-Multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 1x8 De-Multiplexer and 1x16 De-Multiplexer by following the same procedure.

Implementation of Higher-order De-Multiplexers

Now, let us implement the following two higher-order De-Multiplexers using lower-order De-Multiplexers.

- 1x8 De-Multiplexer
- 1x16 De-Multiplexer

1x8 De-Multiplexer

In this section, let us implement 1x8 De-Multiplexer using 1x4 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x4 De-Multiplexer has single input, two selection lines and four outputs. Whereas, 1x8 De-Multiplexer has single input, three selection lines and eight outputs.

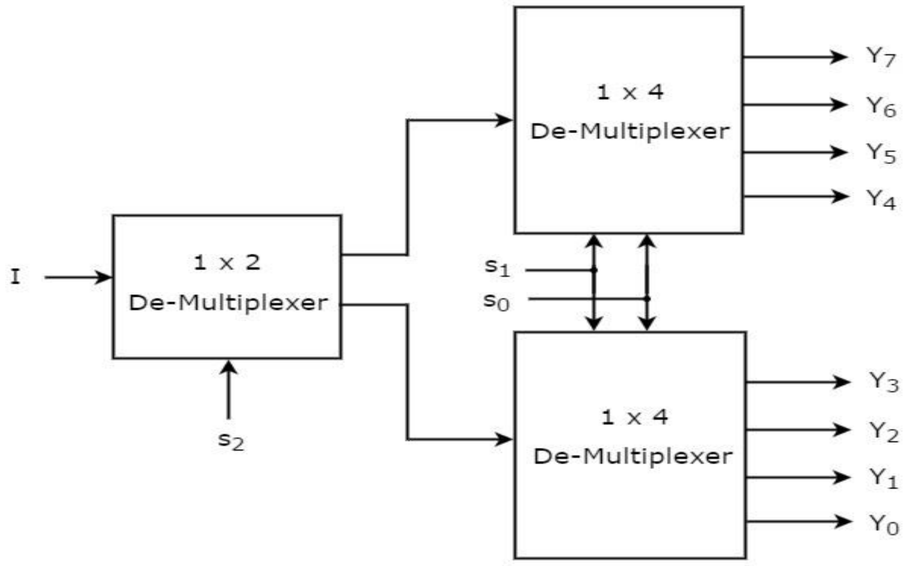
So, we require two **1x4 De-Multiplexers** in second stage in order to get the final eight outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x8 De-Multiplexer.

Let the 1x8 De-Multiplexer has one input I, three selection lines s_2, s_1 & s_0 and outputs Y_7 to Y_0. The **Truth table** of 1x8 De-Multiplexer is shown below.

Selection Inputs			**Outputs**							
s_2	s_1	s_0	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	**I**
0	0	1	0	0	0	0	0	0	**I**	0
0	1	0	0	0	0	0	0	**I**	0	0
0	1	1	0	0	0	0	**I**	0	0	0
1	0	0	0	0	0	**I**	0	0	0	0
1	0	1	0	0	**I**	0	0	0	0	0
1	1	0	0	**I**	0	0	0	0	0	0

1	1	1	**I**	0	0	0	0	0	0	0

We can implement 1x8 De-Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 1x8 De-Multiplexer is shown in the following figure.



The common **selection lines, s_1 & s_0** are applied to both 1x4 De-Multiplexers. The outputs of upper 1x4 De-Multiplexer are Y_7 to Y_4 and the outputs of lower 1x4 De-Multiplexer are Y_3 to Y_0.

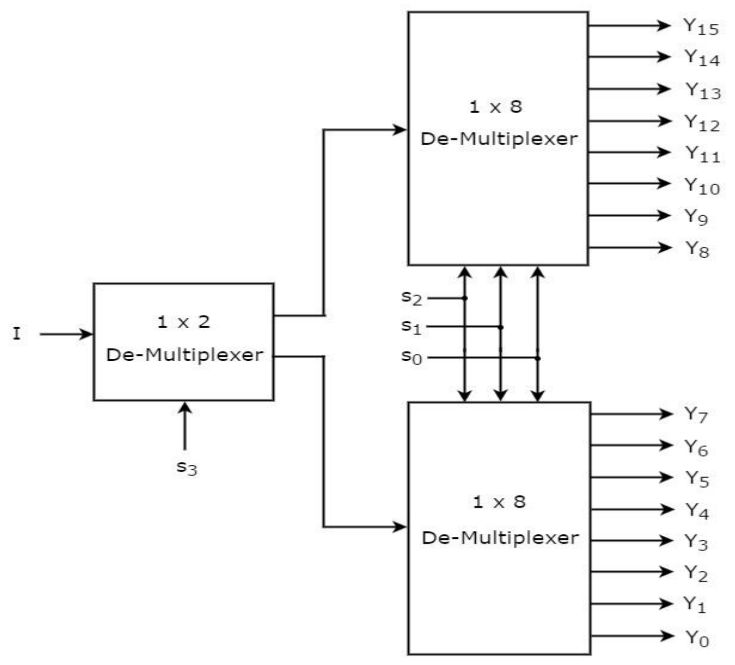
The other **selection line, s_2** is applied to 1x2 De-Multiplexer. If s_2 is zero, then one of the four outputs of lower 1x4 De-Multiplexer will be equal to input, I based on the values of selection lines s_1 & s_0. Similarly, if s_2 is one, then one of the four outputs of upper 1x4 DeMultiplexer will be equal to input, I based on the values of selection lines s_1 & s_0.

1x16 De-Multiplexer

In this section, let us implement 1x16 De-Multiplexer using 1x8 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x8 De-Multiplexer has single input, three selection lines and eight outputs. Whereas, 1x16 De-Multiplexer has single input, four selection lines and sixteen outputs.

So, we require two **1x8 De-Multiplexers** in second stage in order to get the final sixteen outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x16 De-Multiplexer.

Let the 1x16 De-Multiplexer has one input I, four selection lines s_3, s_2, s_1 & s_0 and outputs Y_{15} to Y_0. The **block diagram** of 1x16 De-Multiplexer using lower order Multiplexers is shown in the following figure.



The common **selection lines s_2, s_1 & s_0** are applied to both 1x8 De-Multiplexers. The outputs of upper 1x8 De-Multiplexer are Y_{15} to Y_8 and the outputs of lower 1x8 DeMultiplexer are Y_7 to Y_0.

The other **selection line, s_3** is applied to 1x2 De-Multiplexer. If s_3 is zero, then one of the eight outputs of lower 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s_2, s_1 & s_0. Similarly, if s3 is one, then one of the 8 outputs of upper 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s_2, s_1 & s_0.

Decoder

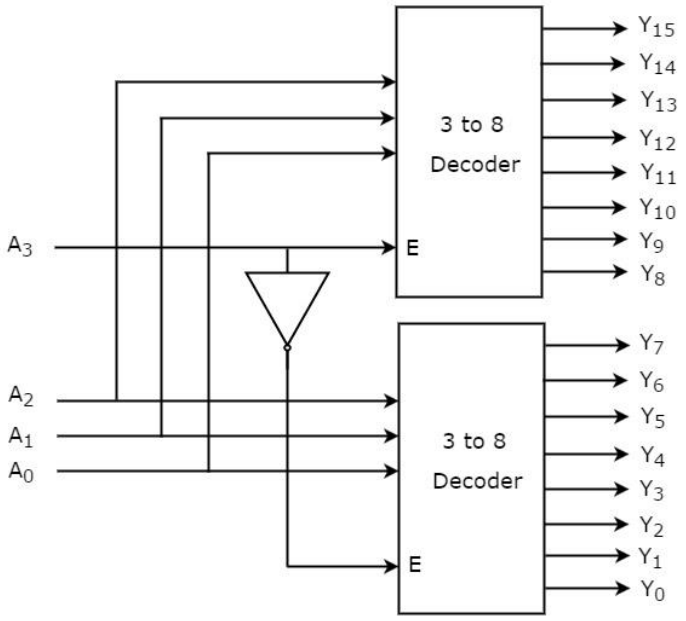
Decoder is a combinational circuit that has 'n' input lines and maximum of 2^n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing but the **min terms** of 'n' input variables (lines), when it is enabled.

1 of 16 Decoder

In this section, let us implement **4 to 16 decoder using 3 to 8 decoders**. We know that 3 to 8 Decoder has three inputs A_2, A_1 & A_0 and eight outputs, Y_7 to Y_0. Whereas, 4 to 16 Decoder has four inputs A_3, A_2, A_1 & A_0 and sixteen outputs, Y_{15} to Y_0

We know the following formula for finding the number of lower order decoders required.

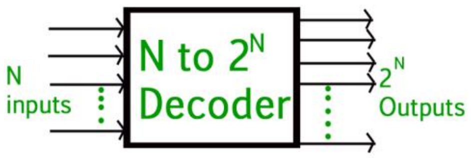
Therefore, we require two 3 to 8 decoders for implementing one 4 to 16 decoder. The **block diagram** of 4 to 16 decoder using 3 to 8 decoders is shown in the following figure.



The parallel inputs A_2, A_1 & A_0 are applied to each 3 to 8 decoder. The complement of input, A3 is connected to Enable, E of lower 3 to 8 decoder in order to get the outputs, Y_7 to Y_0. These are the **lower eight min terms**. The input, A_3 is directly connected to Enable, E of upper 3 to 8 decoder in order to get the outputs, Y_{15} to Y_8. These are the **higher eight min terms**.

BCD to Decimal Decoder

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



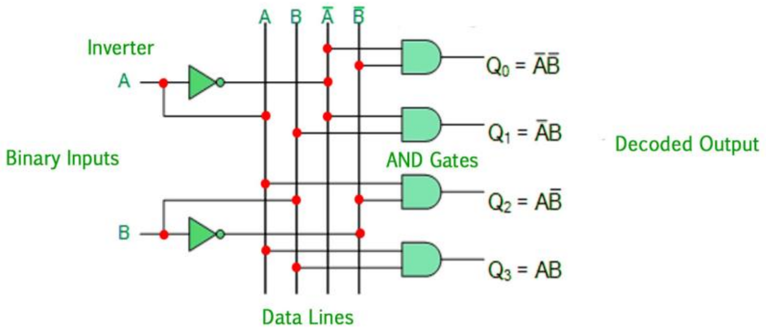
Binary Decoder

- Binary Decoders are another type of digital logic device that has inputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines, so a decoder that has a set of two or more bits will be defined as having an n-bit code, and therefore it will be possible to represent 2^n possible values.
- If a binary decoder receives n inputs it activates one and only one of its 2^n outputs based on that input with all other outputs deactivated. If the n -bit coded information has unused combinations, the decoder may have fewer than 2^n outputs.

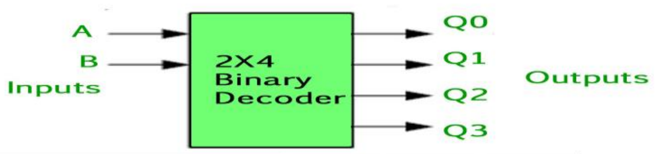
- Example, an inverter (NOT-gate) can be classified as a 1-to-2 binary decoder as 1-input and 2-outputs is possible. i.e an input A can give either A or A complement as the output.
- Then we can say that a standard combinational logic decoder is an n-to-m decoder, where m <= 2^n, and whose output, Q is dependent only on its present input states.
- Their purpose is to generate the 2^n (or fewer) minterms of n input variables. Each combination of inputs will assert a unique output.

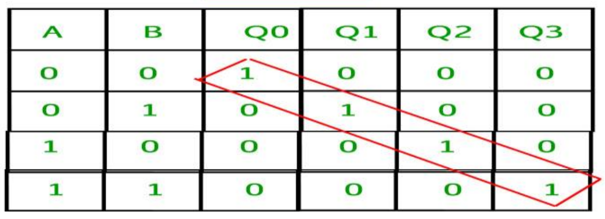
A Binary Decoder converts coded inputs into coded outputs, where the input and output codes are different and decoders are available to “decode” either a Binary or BCD (8421 code) input pattern to typically a Decimal output code.
Practical “binary decoder” circuits include 2-to-4, 3-to-8 and 4-to-16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

<u>2-to-4 Binary Decoder</u>



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





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

The output values will be:

Qo=A’B’
Q1=A’B
Q2=AB’
Q3=AB

The binary inputs A and B determine which output line from Q0 to Q3 is “HIGH” at logic level “1” while the remaining outputs are held “LOW” at logic “0” so only one output can be active (HIGH) at any one time. Therefore, whichever output line is “HIGH” identifies the binary code present at the input, in other words, it “decodes” the binary input.

Some binary decoders have an additional input pin labeled “Enable” that controls the outputs from the device. This extra input allows the outputs of the decoder to be turned “ON” or “OFF” as required. The output is only generated when the Enable input has value 1; otherwise, all outputs are 0. Only a small change in the implementation is required: the Enable input is fed into the AND gates which produce the outputs.

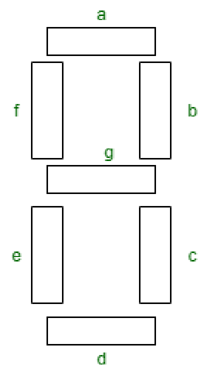
If Enable is 0, all AND gates are supplied with one of the inputs as 0 and hence no output is produced. When Enable is 1, the AND gates get one of the inputs as 1, and now the output depends upon the remaining inputs. Hence the output of the decoder is dependent on whether the Enable is high or low.

Seven Segment Decoder

Light Emitting Diode (LED) is the most widely used semiconductor which emits either visible light or invisible infrared light when forward biased. Remote controls generate invisible light. A Light emitting diode (LED) is an optical electrical energy into light energy when voltage is applied.

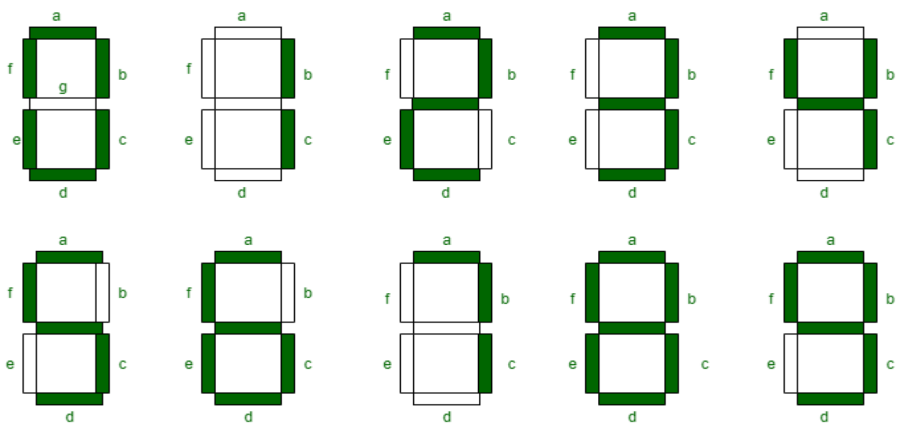
Seven Segment Displays

Seven segment displays are the output display device that provide a way to display information in the form of image or text or decimal numbers which is an alternative to the more complex dot matrix displays. It is widely used in digital clocks, basic calculators, electronic meters, and other electronic devices that display numerical information. It consists of seven segments of light emitting diodes (LEDs) which is assembled like numerical 8.



Working of Seven Segment Displays

The number 8 is displayed when the power is given to all the segments and if you disconnect the power for ‘g’, then it displays number 0. In a seven segment display, power (or voltage) at different pins can be applied at the same time, so we can form combinations of display numerical from 0 to 9. Since seven segment displays can not form alphabet like X and Z, so it can not be used for alphabet and it can be used only for displaying decimal numerical magnitudes. However, seven segment displays can form alphabets A, B, C, D, E, and F, so they can also used for representing hexadecimal digits.



We can produce a truth table for each decimal digit

Decimal Digit	Individual Segments Illuminated						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Therefore, Boolean expressions for each decimal digit which requires respective light emitting diodes (LEDs) are ON or OFF. The number of segments used by digit: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 are 6, 2, 5, 5, 4, 5, 6, 3, 7, and 6 respectively. Seven segment displays must be controlled by other external devices where different types of microcontrollers are useful to communicate with these external devices, like switches, keypads, and memory.

Types of Seven Segment Displays

According to the type of application, there are two types of configurations of seven segment displays: common anode display and common cathode display.

1. In common cathode seven segment displays, all the cathode connections of LED segments are connected together to logic 0 or ground. We use logic 1 through a current limiting resistor to forward bias the individual anode terminals a to g.

2. Whereas all the anode connections of the LED segments are connected together to logic 1 in common anode seven segment display. We use logic 0 through a current limiting resistor to the cathode of a particular segment a to g.

Common anode seven segment displays are more popular than cathode seven segment displays, because logic circuits can sink more current than they can source and it is the same as connecting LEDs in reverse.

Applications of Seven Segment Displays

Common applications of seven segment displays are in:

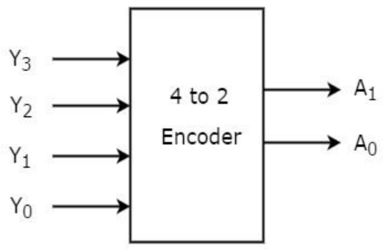
1. Digital clocks
2. Clock radios
3. Calculators
4. Wristwatchers
5. Speedometers
6. Motor-vehicle odometers
7. Radio frequency indicators

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

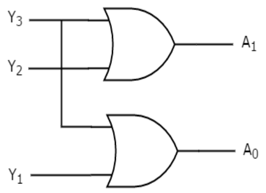
1	0	0	0	1	1

From Truth table, we can write the **Boolean functions** for each output as

A{1}=Y{3}+Y{2}

A{0}=Y{3}+Y{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

Exclusive-OR gate

The full form of Ex-OR gate is **Exclusive-OR** gate. Its function is same as that of OR gate except for some cases, when the inputs having even number of ones.

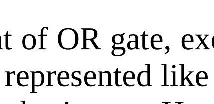
The following table shows the **truth table** of 2-input Ex-OR gate.

A	**B**	**Y = A⊕B**
0	0	0
0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input Ex-OR gate. The truth table of Ex-OR gate is same as that of OR gate for first three rows. The only modification is in the fourth row. That means, the output (Y) is zero instead of one, when both the inputs are one, since the inputs having even number of ones.

Therefore, the output of Ex-OR gate is ‘1’, when only one of the two inputs is ‘1’. And it is zero, when both inputs are same.

Below figure shows the **symbol** of Ex-OR gate, which is having two inputs A, B and one output, Y.



Ex-OR gate operation is similar to that of OR gate, except for few combination(s) of inputs. That’s why the Ex-OR gate symbol is represented like that. The output of Ex-OR gate is ‘1’, when odd number of ones present at the inputs. Hence, the output of Ex-OR gate is also called as an **odd function**.

Parity Bit Generator

There are two types of parity bit generators based on the type of parity bit being generated. **Even parity generator** generates an even parity bit. Similarly, **odd parity generator** generates an odd parity bit.

Even Parity Generator

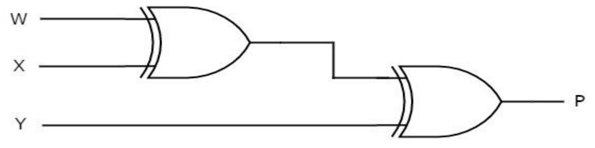
Now, let us implement an even parity generator for a 3-bit binary input, WXY. It generates an even parity bit, P. If odd number of ones present in the input, then even parity bit, P should be '1' so that the resultant word contains even number of ones. For other combinations of input, even parity bit, P should be '0'. The following table shows the **Truth table** of even parity generator.

Binary Input WXY	Even Parity bit P
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

From the above Truth table, we can write the **Boolean function** for even parity bit as

P={W}'{X}'Y+{W}'X{Y}'+W{X}'{Y}'+WXY

The following figure shows the **circuit diagram** of even parity generator.

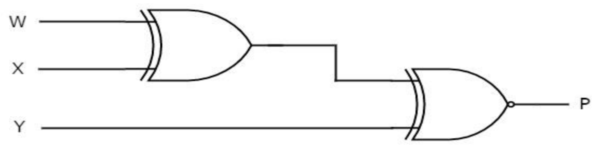


This circuit consists of two **Exclusive-OR gates** having two inputs each. First ExclusiveOR gate having two inputs W & X and produces an output W ⊕ X. This output is given as one input of second Exclusive-OR gate. The other input of this second Exclusive-OR gate is Y and produces an output of W ⊕ X ⊕ Y.

Odd Parity Generator

If even number of ones present in the input, then odd parity bit, P should be '1' so that the resultant word contains odd number of ones. For other combinations of input, odd parity bit, P should be '0'.

Follow the same procedure of even parity generator for implementing odd parity generator. The **circuit diagram** of odd parity generator is shown in the following figure.



The above circuit diagram consists of Ex-OR gate in first level and Ex-NOR gate in second level. Since the odd parity is just opposite to even parity, we can place an inverter at the output of even parity generator. In that case, the first and second levels contain an ExOR gate in each level and third level consist of an inverter.

Parity Checker

There are two types of parity checkers based on the type of parity has to be checked. **Even parity checker** checks error in the transmitted data, which contains message bits along with even parity. Similarly, **odd parity checker** checks error in the transmitted data, which contains message bits along with odd parity.

Even parity checker

Now, let us implement an even parity checker circuit. Assume a 3-bit binary input, WXY is transmitted along with an even parity bit, P. So, the resultant word (data) contains 4 bits, which will be received as the input of even parity checker.

It generates an **even parity check bit, E**. This bit will be zero, if the received data contains an even number of ones. That means, there is no error in the received data. This even parity check bit will be one, if the received data contains an odd number of ones. That means, there is an error in the received data.

The following table shows the **Truth table** of an even parity checker.

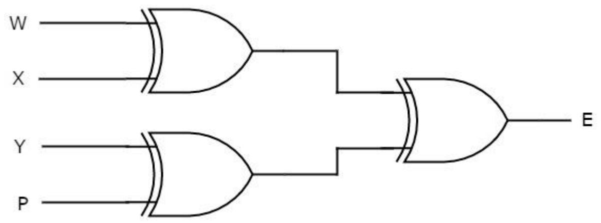
4-bit Received Data WXYP	Even Parity Check bit E
0000	0
0001	1
0010	1
0011	0
0100	1
0101	0
0110	0
0111	1
1000	1
1001	0

1010	0
1011	1
1100	0
1101	1
1110	1
1111	0

From the above Truth table, we can observe that the even parity check bit value is ‘1’, when odd number of ones present in the received data. That means the Boolean function of even parity check bit is an **odd function**. Exclusive-OR function satisfies this condition. Hence, we can directly write the **Boolean function** of even parity check bit as

E=W +X + Y + P

The following figure shows the **circuit diagram** of even parity checker.



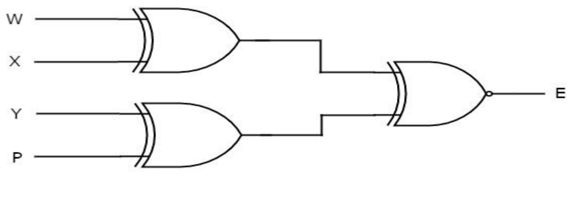
This circuit consists of three **Exclusive-OR gates** having two inputs each. The first level gates produce outputs of W + X & Y + P. The Exclusive-OR gate, which is in second level produces an output of W + X + Y + P.

Odd Parity Checker

Assume a 3-bit binary input, WXY is transmitted along with odd parity bit, P. So, the resultant word (data) contains 4 bits, which will be received as the input of odd parity checker.

It generates an **odd parity check bit, E**. This bit will be zero, if the received data contains an odd number of ones. That means, there is no error in the received data. This odd parity check bit will be one, if the received data contains even number of ones. That means, there is an error in the received data.

Follow the same procedure of an even parity checker for implementing an odd parity checker. The **circuit diagram** of odd parity checker is shown in the following figure.



The above circuit diagram consists of Ex-OR gates in first level and Ex-NOR gate in second level. Since the odd parity is just opposite to even parity, we can place an inverter at the output of even parity checker. In that case, the first, second and third levels contain two Ex-OR gates, one Ex-OR gate and one inverter respectively.

REVIEW QUESTIONS

1) Write associate and commutative law.
2) Define multiplexer.
3) Explain in detail about seven segment decoder.
4) Define karnaugh map.
5) How to fine pair, quad and octet in k-map.
6) Explain about don’t care condition.
7) Explain about parity checker.
8) Discuss about XOR gates.

UNIT – III

Arithmetic Circuits

- **Binary Addition**
- **Binary Subtraction**
- **2’s Complement Representation**
- **2’s Complement Arithmetic**
- **Arithmetic Building Blocks**

ARITHMETIC CIRCUITS

Binary Addition

It is a key for binary subtraction, multiplication, division. There are four rules of binary addition.

Case	A	+	B	Sum	Carry
1	0	+	0	0	0
2	0	+	1	1	0
3	1	+	0	1	0
4	1	+	1	0	1

In fourth case, a binary addition is creating a sum of (1 + 1 = 10) i.e. 0 is written in the given column and a carry of 1 over to the next column.

Example − Addition

```
0011010 + 001100 = 00100110        1 1          carry
                                 0 0 1 1 0 1 0  = 26₁₀
                                +0 0 0 1 1 0 0  = 12₁₀
                                ──────────────
                                 0 1 0 0 1 1 0  = 38₁₀
```

Binary Subtraction

Subtraction and Borrow, these two words will be used very frequently for the binary subtraction. There are four rules of binary subtraction.

Case	A	-	B	Subtract	Borrow
1	0	-	0	0	0
2	1	-	0	1	0
3	1	-	1	0	0
4	0	-	1	0	1

Example − Subtraction

0011010 - 001100 = 00001110

$$\begin{array}{rl} 1\;1\quad & \text{borrow} \\ 0\,0\,\cancel{1}\,\cancel{1}\,0\,1\,0 & =26_{10} \\ -0\,0\,0\,1\,1\,0\,0 & =12_{10} \\ \hline 0\,0\,0\,1\,1\,1\,0 & =14_{10} \end{array}$$

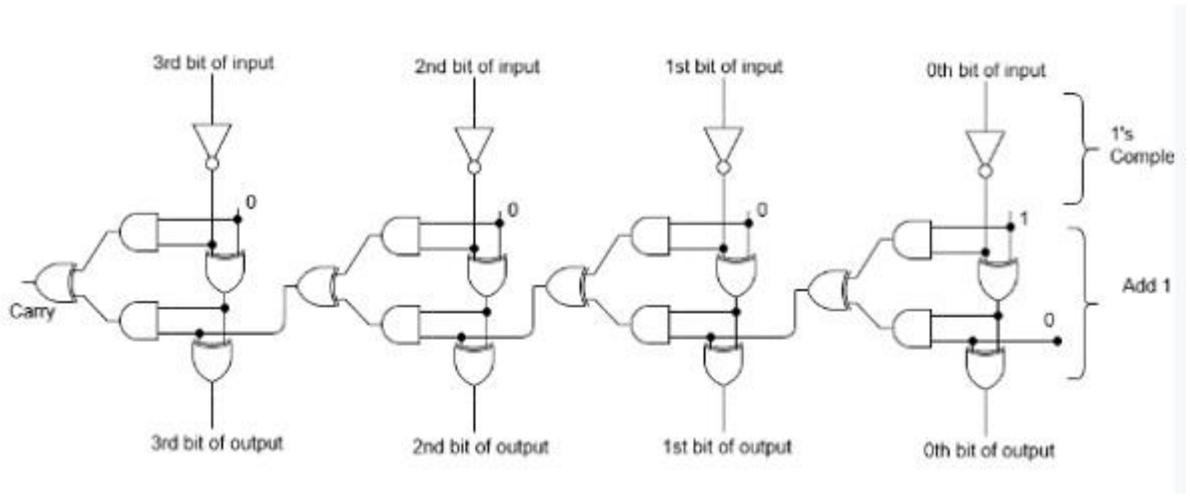
2’s Complement Representation

Binary Number System is one the type of most popular Number Representation techniques that used in digital systems. In the Binary System, there are only two symbols or possible digit values, i.e., 0 (off) and 1 (on). Represented by any device that only 2 operating states or possible conditions.

Generally, there are two types of complement of Binary number: 1’s complement and 2’s complement. To get 1’s complement of a binary number, simply invert the given number. For example, 1’s complement of binary number 110010 is 001101. To get 2’s complement of binary number is 1’s complement of given number plus 1 to the least significant bit (LSB). For example 2’s complement of binary number 10010 is (01101) + 1 = 01110.

2’s Complement of a Binary Number

There is a simple algorithm to convert a binary number into 2’s complement. To get 2’s complement of a binary number, simply invert the given number and add 1 to the least significant bit (LSB) of given result. Implementation of 4-bit 2’s complementation number is given as following below.



Eg

Find 2’s complement of binary number 10101110.

Simply invert each bit of given binary number, which will be 01010001. Then add 1 to the LSB of this result, i.e., 01010001+1=01010010 which is answer.

Eg

Find 2’s complement of binary number 10001.001.

Simply invert each bit of given binary number, which will be 01110.110 Then add 1 to the LSB of this result, i.e., 01110.110+1=01110.111 which is answer.

<u>Eg</u>

Find 2's complement of each 3 bit binary number.

Simply invert each bit of given binary number, then add 1 to LSB of these inverted numbers,

Binary number	1's complement	2's complement
000	111	000
001	110	111
010	101	110
011	100	101
100	011	100
101	010	011
110	001	010
111	000	001

<u>Uses of 2's Complement Binary Numbers</u>

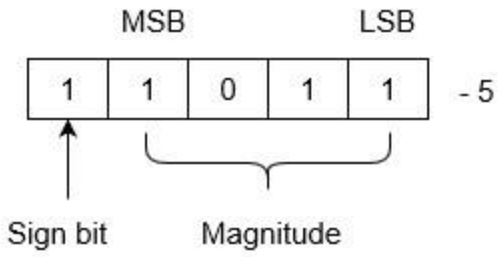
There are various uses of 2's complement of Binary numbers, mainly in signed Binary number representation and various arithmetic operations for Binary numbers, e.g., additions, subtractions, etc. Since 2's complement representation is unambiguous, so it very useful in Computer number representation.

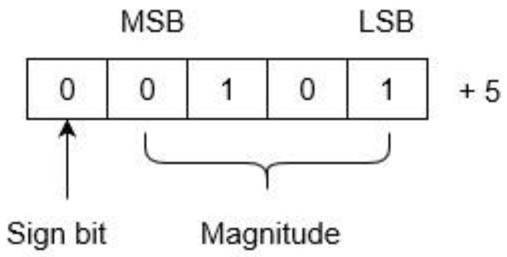
<u>2's Complementation in Signed Binary number Representation</u>

Positive numbers are simply represented as simple Binary representation. But if the number is negative then it is represented using 2's complement. First represent the number with positive sign and then take 2's complement of that number.

<u>Eg</u>

Let we are using 5 bits registers. The representation of -5 and +5 will be as follows:





+5 is represented as it is represented in sign magnitude method. -5 is represented using the following steps:

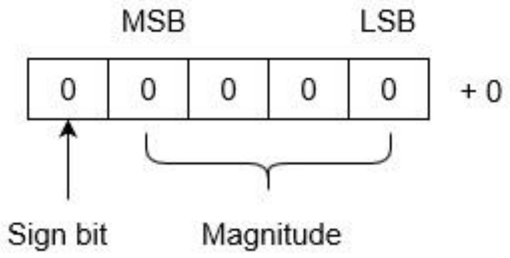
(i) +5 = 0 0101

(ii) Take 2's complement of 0 0101 and that is 1 1011. MSB is 1 which indicates that number is negative.

MSB is always 1 in case of negative numbers.

Range of Numbers −For k bits register, positive largest number that can be stored is $(2^{(k-1)}-1)$ and negative lowest number that can be stored is $-(2^{(k-1)})$.

The advantage of this system is that 0 has only one representation for -0 and +0. Zero (0) is considered as always positive (sign bit is 0) in 2's complement representation. Therefore, it is unique or unambiguous representation.



Lets see arithmetic operations: Subtractions and Additions in 2's complement binary numbers.

<u>2's Complement Arithmetic</u>

<u>Subtractions by 2's Complement</u>

The algorithm to subtract two binary number using 2's complement is explained as following below −

- Take 2's complement of the subtrahend
- Add with minuend
- If the result of above addition has carry bit 1, then it is dropped and this result will be positive number.
- If there is no carry bit 1, then take 2's complement of the result which will be negative

Note that subtrahend is number that to be subtracted from the another number, i.e., minuend.

Also, note that adding *end-around carry-bit* occurs only in 1's complement arithmetic operations but not 2's complement arithmetic operations.

Eg (Case-1: When Carry bit 1) −Evaluate 10101 - 00101

According to above algorithm, take 2's complement of subtrahend 00101, which will be 11011, then add both of these. So, 10101 + 11011 =1 10000. Since, there is carry bit 1, so dropped this carry bit 1, and take this result will be 10000 will be positive number.

Eg (Case-2: When no Carry bit) −Evaluate 11001 - 11100

According to above algorithm, take 2's complement of subtrahend 11110, which will be 00100. Then add both of these, So, 11001 + 00100 =11101. Since there is no carry bit 1, so take 2's complement of above result, which will be 00011, and this is negative number, i.e, 00011, which is the answer.

Similarly, you can subtract two mixed (with fractional part) binary numbers.

Additions by 2's Complement

There are difference scenario for addition of two binary numbers using 2's complement. These are explained as following below.

Case-1 − Addition of positive and negative number when positive number has greater magnitude:

When positive number has greater magnitude, then take simply 2's complement of negative number and carry bit 1 is dropped and this result will be positive number.

Example −Add 1110 and -1101.

So, take 2's complement of 1101, which will be 0011, then add with given number. So, 1110+0011=1 0001, and carry bit 1 is dropped and this result will be positive number, i.e., +0001.

Note that if the register size is big then use sign extension method of MSB bit to preserve sign of number.

Case-2 − Addition of positive and negative number when negative number has greater magnitude −

When the negative number has greater magnitude, then take 2's complement of negative number and add with given positive number. Since there will not be any end-around carry bit, so take 2's complement of the result and this result will be negative.

Example −Add 1010 and -1100 in five-bit registers.

Note that there are five-bit registers, so these new numbers will have 01010 and -01100. Now take 2's complement of 01100 which will be 10100 and add 01010+10100=11110. Then take 2's complement of this result, which will be 00010 and this will be negative number, i.e., -00010, which is the answer.

Case-3 − Addition of two negative numbers −

You need to take 2's complement for both numbers, then add these 2's complement of numbers. Since there will always be end-around carry bit, so it is dropped. Now, take 2's complement also of previous result, so this will be negative number.

Alternatively, you can add both of these Binary numbers and take result which will be negative only.

Example − add -1010 and -0101 in five bit-register.

These five bit numbers are -01010 and -00101. Add 2's complements of these numbers, 10110+11011 =1 10001. Since, there is a carry bit 1, so it is dropped. Now take the 2's complement of this result, which will be 01111 and this number is negative, i.e, -01111, which is answer.

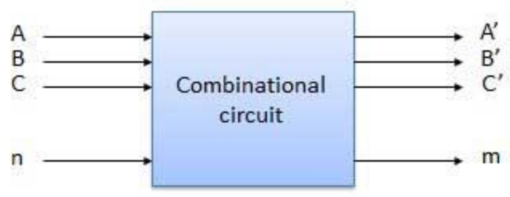
Note that 2's complement arithmetic operations are much easier than 1's complement because of there is no addition of *end-around-carry-bit.*

Arithmetic Building Blocks

Combinational circuit is a circuit in which we combine the different gates in the circuit, for example encoder, decoder, multiplexer and demultiplexer. Some of the characteristics of combinational circuits are following −

- The output of combinational circuit at any instant of time, depends only on the levels present at input terminals.
- The combinational circuit do not use any memory. The previous state of input does not have any effect on the present state of the circuit.
- A combinational circuit can have an n number of inputs and m number of outputs.

Block diagram

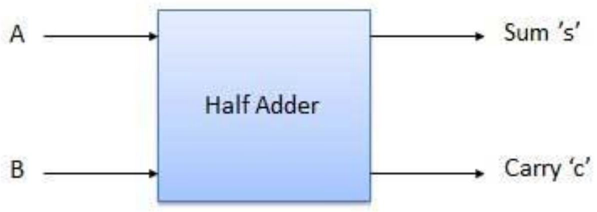


We're going to elaborate few important combinational circuits as follows.

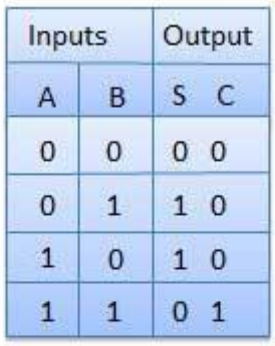
Half Adder

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

Block diagram

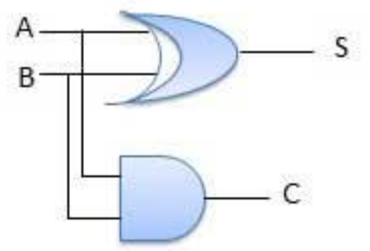


Truth Table



Inputs		Output
A	B	S C
0	0	0 0
0	1	1 0
1	0	1 0
1	1	0 1

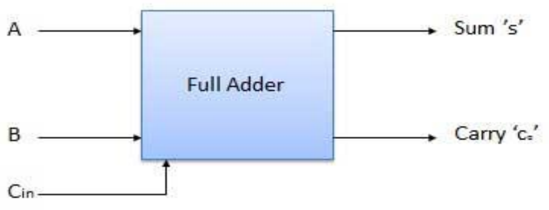
Circuit Diagram



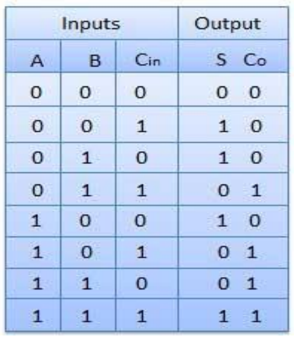
Full Adder

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

Block diagram

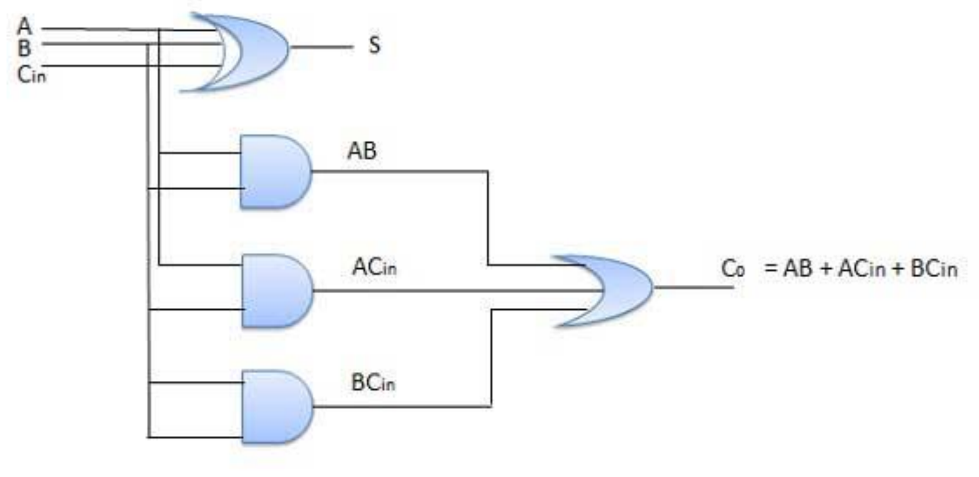


Truth Table



Inputs			Output
A	B	C_{in}	S C_o
0	0	0	0 0
0	0	1	1 0
0	1	0	1 0
0	1	1	0 1
1	0	0	1 0
1	0	1	0 1
1	1	0	0 1
1	1	1	1 1

Circuit Diagram



REVIEW QUESTIONS

1) Define 2’s complement.
2) Explain about half adder.
3) How to perform 2’s complement arithmetic.
4) Explain about binary addition.
5) Write a short note on full adder.
6) Discuss about binary subtraction.

Unit IV:

- Basic Computer Organization and Design
    - Instruction codes
    - Stored program organization
    - Computer Registers
    - Common bus system
    - Computer Instructions
    - Timing and Control
    - Instruction cycle: Fetch and Decode
    - Register reference Instructions
- Micro Programmed Control
    - Control memory organization
    - Address Sequencing
    - Micro instruction format
    - Symbolic micro instructions
    - Symbolic micro program
    - Binary micro program

Basic computer organization and design

Instruction Code

- An instruction code is a group of bits that instruct the computer to perform a specific operation.

Operation Code

- The operation code of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement.
- The number of bits required for the operation code of an instruction depends on the total number of operations available in the computer.
- The operation code must consist of at least n bits for a given 2^n(or less) distinct operations.

Accumulator (AC)

- Computers that have a single-processor register usually assign to it the name accumulator (AC) accumulator and label it AC.
- The operation is performed with the memory operand and the content of AC.

Stored Program Organization

- The simplest way to organize a computer is to have one processor register andan instruction code format with two parts.
- The first part specifies the operation to be performed and the second specifiesan address.
- The memory address tells the control where to find an operand in memory.

- This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.
- The following figure 2.1 shows this type of organization.

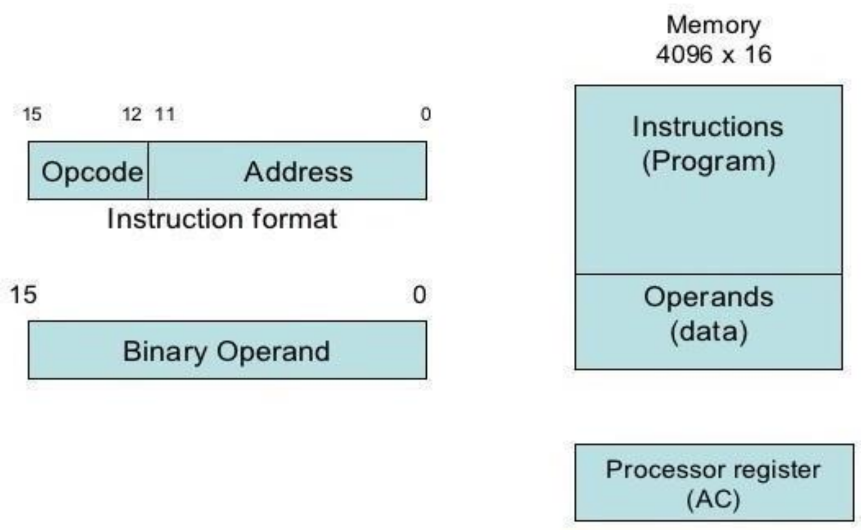


Figure 2.1: Stored Program Organization

- Instructions are stored in one section of memory and data in another.
- For a memory unit with 4096 words, we need 12 bits to specify an address since 2^{12}= 4096.
- If we store each instruction code in one 16-bit memory word, we have available four bits for operation code (abbreviated opcode) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand.
- The control reads a 16-bit instruction from the program portion of memory.
- It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory.
- It then executes the operation specified by the operation code.
- Computers that have a single-processor register usually assign to it the name accumulator and label it AC.
- If an operation in an instruction code does not need an operand from memory, the rest of the bits in the instruction can be used for other purposes.
- For example, operations such as clear AC, complement AC, and increment AC operate on data stored in the AC register. They do not need an operand from memory. For these types of operations, the second part of the instruction code (bits 0 through 11) is not needed for specifying a memory address and can be used to specify other operations for the computer.

Direct and Indirect addressing of basic computer:

- The second part of an instruction format specifies the address of an operand, the instruction is said to have a **direct address**.
- In **Indirect address**, the bits in the second part of the instruction designate an address of a memory word in which the address of the operand is found.
- One bit of the instruction code can be used to distinguish between a direct and an indirect address.

15 14 12 11 0

I	Opcode	Addres s

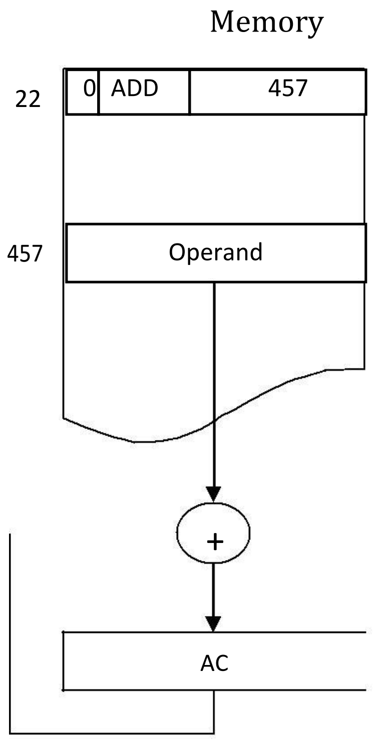


Figure 2.2: Direct Address

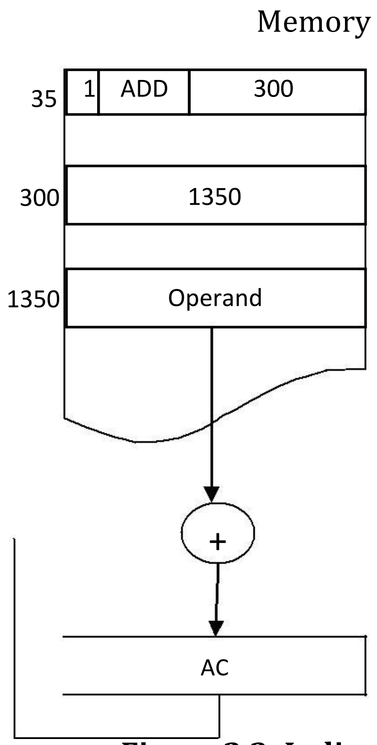


Figure 2.3: Indirect Address

- It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by I.
- The mode bit is 0 for a direct address and 1 for an indirect address.
- A direct address instruction is shown in Figure 2.2. It is placed in address 22 in memory.
- The I bit is 0, so the instruction is recognized as a direct address instruction.
- The opcode specifies an ADD instruction, and the address part is the binary equivalent of 457.
- The control finds the operand in memory at address 457 and adds it to the content of AC.
- The instruction in address 35 shown in Figure 2.3 has a mode bit I = 1, recognized as an indirect address instruction.
- The address part is the binary equivalent of 300.
- The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC.
- The indirect address instruction needs two references to memory to fetch an operand.
    1. The first reference is needed to read the address of the operand
    2. Second reference is for the operand itself.
- The memory word that holds the address of the operand in an indirect address instruction is used as a pointer to an array of data.

Direct Address	**Indirect Address**

When the second part of an instruction code specifies the address of an operand, the instruction is said to have a direct address.	When the second part of an instruction code specifies the address of a memory word in which the address of the operand, the instruction is said to have a direct address.
For instance the instruction MOV R0 00H. R0, when converted to machine language is the physical address of register R0. The instruction moves 0 to R0.	For instance the instruction MOV @R0 00H, when converted to machine language, @R0 becomes whatever is stored in R0, and that is the address used to move 0 to. It can be whatever is stored in R0.

Computer Registers:

- It is necessary to provide a register in the control unit for storing the instruction code after it is read from memory.
- The computer needs processor registers for manipulating data and a register for holding a memory address.
- These requirements dictate the register configuration shown in Figure 2.4.

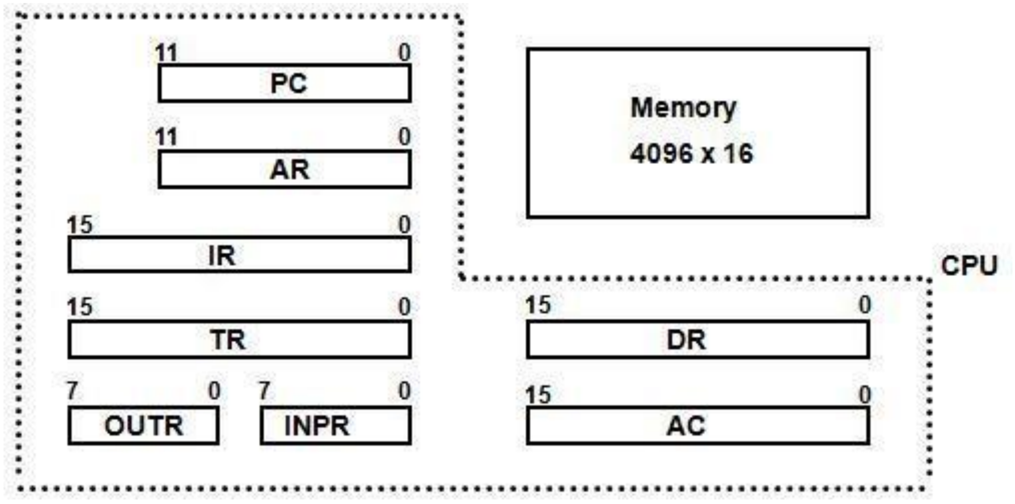


Figure 2.4: Basic Computer Register and Memory

- The data register (DR) holds the operand read from memory.
- The accumulator (AC) register is a general purpose processing register.
- The instruction read from memory is placed in the instruction register (IR).
- The temporary register (TR) is used for holding temporary data during the processing.
- The memory address register (AR) has 12 bits.
- The program counter (PC) also has 12 bits and it holds the address of the next instruction to be read from memory after the current instruction is executed.
- Instruction words are read and executed in sequence unless a branch instruction is encountered. A branch instruction calls for a transfer to a nonconsecutive instruction in the program.

- Two registers are used for input and output. The input register (INPR) receives an 8-bit character from an input device. The output register (OUTR) holds an 8-bit character for an output device.

Register Symbol	Bits	Register Name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

Table 2.1: List of Registers for Basic Computer

Common Bus System for basic computer register:

Requirement of common bus System

- The basic computer has eight registers, a memory unit and a control unit.
- Paths must be provided to transfer information from one register to another and between memory and register.
- The number of wires will be excessive if connections are between the outputs of each register and the inputs of the other registers. An efficient scheme for transferring information in a system with many register is to use a common bus.
- The connection of the registers and memory of the basic computer to a common bus system is shown in figure 2.5.
- The outputs of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S2, S1, and S0.
- The number along each output shows the decimal equivalent of the required binary selection.
- The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition. The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and S2 S1 S0 = 1 1 1.
- Four registers, DR, AC, IR, and TR have 16 bits each.
- Two registers, AR and PC, have 12 bits each since they hold a memory address.
- When the contents of AR or PC are applied to the 16-bit common bus, the four most significant bits are set to 0's. When AR and PC receive information from the bus, only the 12 least significant bits are transferred into the register.
- The input register INPR and the output register OUTR have 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus.

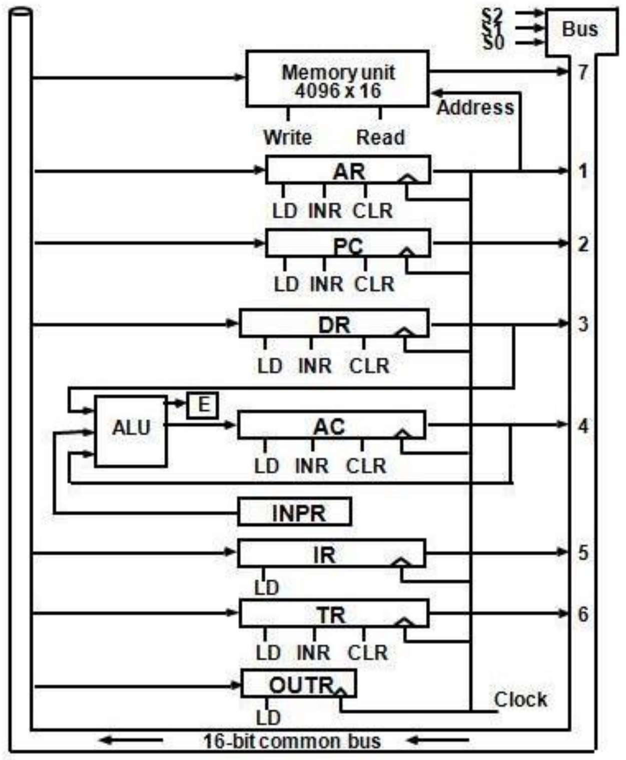


Figure 2.5: Basic computer registers connected to a common bus

- Five registers have three control inputs: LD (load), INR (increment), and CLR (clear). Two registers have only a LD input.
- AR must always be used to specify a memory address; therefore memory address is connected to AR.
- The 16 inputs of AC come from an adder and logic circuit. This circuit has three sets of inputs.
  1. Set of 16-bit inputs come from the outputs of AC.
  2. Set of 16-bits come from the data register DR.
  3. Set of 8-bit inputs come from the input register INPR.
- The result of an addition is transferred to AC and the end carry-out of the addition is transferred to flip-flop E (extended AC bit).
- The clock transition at the end of the cycle transfers the content of the bus into the designated destination register and the output of the adder and logic circuit into AC.

Computer Instructions and Format with its types:

- The basic computer has three instruction code formats, as shown in figure 2.6.
- Each format has 16 bits. The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.

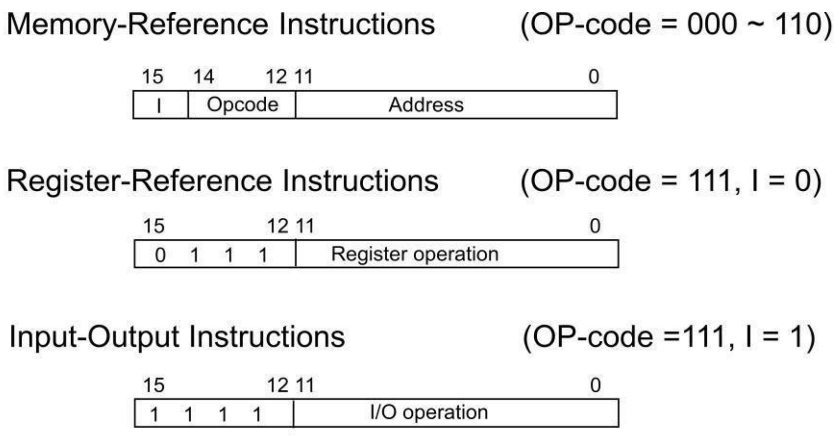


Figure 2.6: Basic computer instruction format

- A **memory-reference instruction** uses 12 bits to specify an address and one bit to specify the addressing mode I. I is equal to 0 for direct address and to 1 for indirect address.
- The **register reference instructions** are recognized by the operation code 111 with a 0 in the leftmost bit (bit 15) of the instruction. A register-reference instruction specifies an operation on or a test of the AC register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed.
- An **input-output instruction** does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

Control Unit with timing diagram:

- The block diagram of the control unit is shown in figure 2.7.
- Components of Control unit are
  1. Two decoders
  2. A sequence counter
  3. Control logic gates
- An instruction read from memory is placed in the instruction register (IR). In control unit the IR is divided into three parts: I bit, the operation code (12-14) bit, and bits 0 through 11.
- The operation code in bits 12 through 14 are decoded with a 3 X 8 decoder.
- Bit-15 of the instruction is transferred to a flip-flop designated by the symbol I.
- The eight outputs of the decoder are designated by the symbols D0 through D7. Bits 0 through 11 are applied to the control logic gates. The 4-bit sequence counter can count in binary from 0 through 15.The outputs of counter are decoded into 16 timing signals T0 through T15.
- The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of 4 X 16 decoder. Once in awhile, the counter is cleared to 0, causing the next timing signal to be T0.

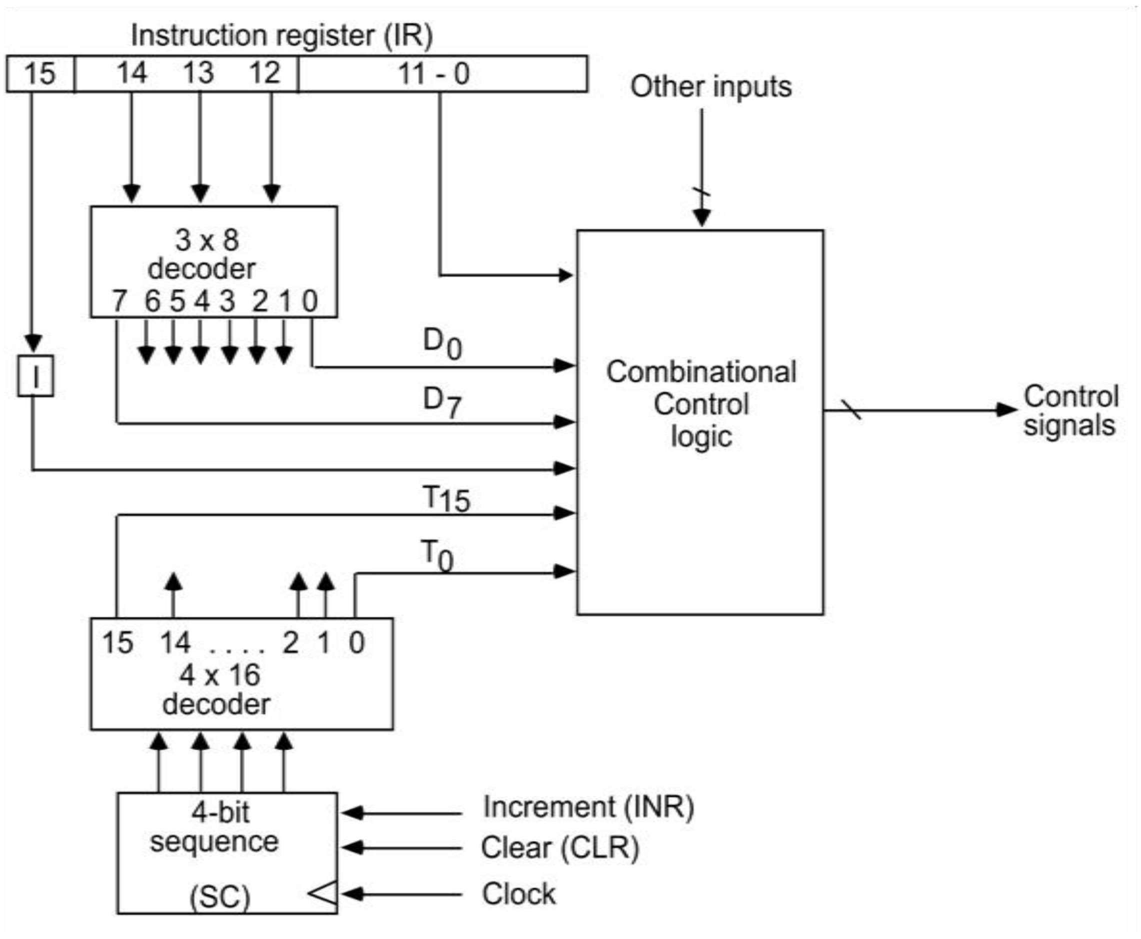


Figure 2.7: Control unit of basic computer

- As an example, consider the case where SC is incremented to provide timing signals T0, T1, T2, T3 and T4 in sequence. At time T4, SC is cleared to 0 if decoder output D3 is active. This is expressed symbolically by the statement

  D3T4: SC ← 0

Timing Diagram:

- The timing diagram figure2.8 shows the time relationship of the control signals.
- The sequence counter SC responds to the positive transition of the clock.
- Initially, the CLR input of SC is active.
- The first positive transition of the clock clears SC to 0, which in turn activates the timing T0 out of the decoder. T0 is active during one clock cycle. The positive clock transition

  labeled T0 in the diagram will trigger only those registers whose control inputs are connected to timing signal T0.
- SC is incremented with every positive clock transition, unless its CLR input is active.
- This procedures the sequence of timing signals T0, T1, T2, T3 and T4, and so on. If SC is not cleared, the timing signals will continue with T5, T6, up to T15 and back to T0.

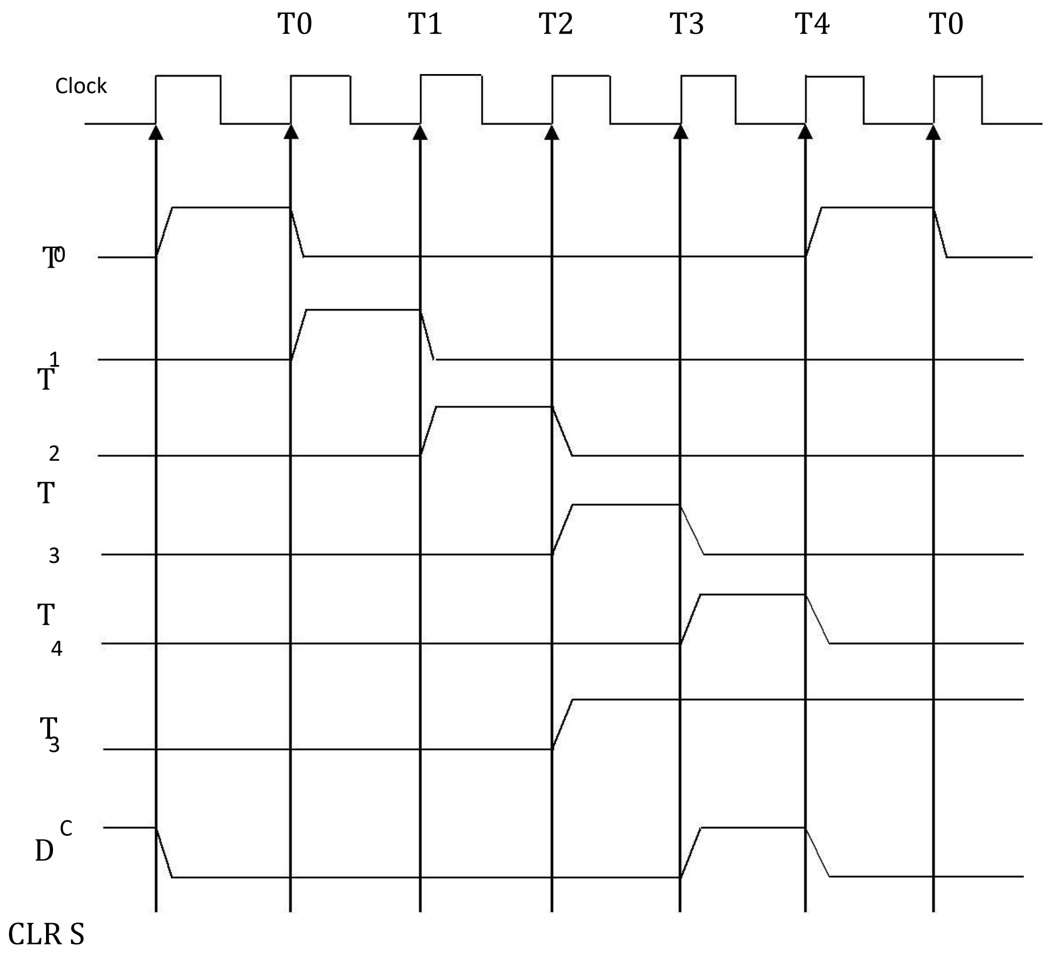
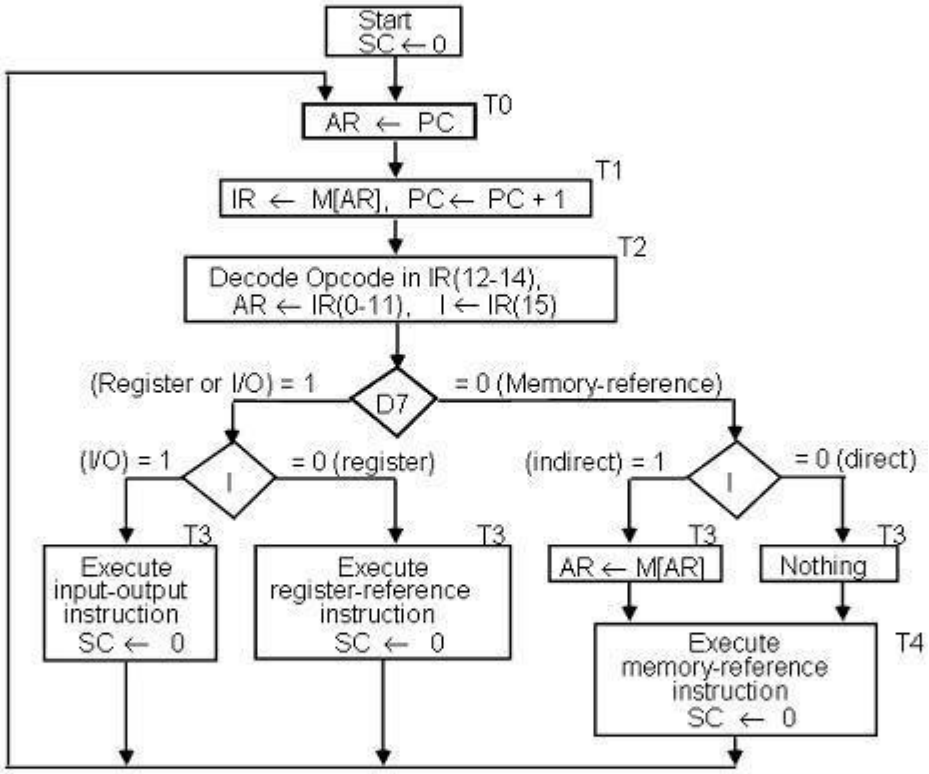


Figure 2.8: Example of control timing signals

- The last three waveforms shows how SC is cleared when D3T4 = 1. Output D3 from the operation decoder becomes active at the end of timing signal T2. When timing signal T4 becomes active, the output of the AND gate that implements the control function D3T4 becomes active.
- This signal is applied to the CLR input of SC. On the next positive clock transition the counter is cleared to 0. This causes the timing signal T0 to become active instead of T5 that would have been active if SC were incremented instead of cleared.

Instruction cycle

- A program residing in the memory unit of the computer consists of a sequence of instructions. In the basic computer each instruction cycle consists of the following phases:
  1. Fetch an instruction from memory.
  2. Decode the instruction.
  3. Read the effective address from memory if the instruction has an indirect address.
  4. Execute the instruction.
- After step 4, the control goes back to step 1 to fetch, decode and execute the nex instruction.
- This process continues unless a HALT instruction is encountered.



Start
SC ← 0
T0
AR ← PC
T1
IR ← M[AR], PC ← PC + 1
T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
T4
Execute memory-reference instruction SC ← 0

Figure 2.9: Flowchart for instruction cycle (initial configuration)

- The flowchart presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.
- If D7 = 1, the instruction must be register-reference or input-output type. If D7 = 0, the operation code must be one of the other seven values 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which now available in flip-flop I.
- If D7 = 0 and I = 1, we have a memory-reference instruction with an indirect address. It is then necessary to read the effective address from memory.
- The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T3.This can be symbolized as follows:

  D_7'IT3 : AR←M[AR]

  D_7'I'T3 : Nothing

  D_7I'T3 : Execute a register reference instruction

  D_7IT3 : Execute an input-output instruction

- When a memory-reference instruction with I = 0 is encountered, it is not necessary to do anything since the effective address is already in AR.
- However, the sequence counter SC must be incremented when D'7 I T3 = 1, so that the execution of the memory-reference instruction can be continued with timing
- variable T4.
- A register-reference or input-output instruction can be executed with the click associated with timing signal T3. After the instruction is executed, SC is cleared to 0

and control returns to the fetch phase with T0 =1. SC is either incremented or cleared to 0 with every positive clock transition.

Register reference instruction:

- When the register-reference instruction is decoded, D7 bit is set to 1.
- Each control function needs the Boolean relation D7 I' T3

15 12	11 0
0 1 1 1	Register Operation

- There are 12 register-reference instructions listed below:

	r:	SC▯0	Clear SC
CLA	rB11:	AC ▯ 0	Clear AC
CLE	rB10:	E ▯ 0	Clear E
CMA	rB9:	AC ▯ AC’	Complement AC
CME	rB8:	E ▯ E’	Complement E
CIR	rB7:	AC ▯ shr AC, AC(15) ▯ E, E ▯ AC(0)	Circular Right
CIL	rB6:	AC ▯ shl AC, AC(0) ▯ E, E ▯ AC(15)	Circular Left
INC	rB5:	AC ▯ AC + 1	Increment AC
SPA	rB4:	if (AC(15) = 0) then (PC ▯ PC+1)	Skip if positive
SNA	rB3:	if (AC(15) = 1) then (PC ▯ PC+1	Skip if negative
SZA	rB2:	if (AC = 0) then (PC ▯ PC+1)	Skip if AC is zero
SZE	rB1:	if (E = 0) then (PC ▯ PC+1)	Skip if E is zero
HLT	rB0:	S ▯ 0 (S is a start-stop flip-flop)	Halt computer

- These 12 bits are available in IR (0-11). They were also transferred to AR during time T2.
- These instructions are executed at timing cycle T3.
- The first seven register-reference instructions perform clear, complement, circular shift, and increment micro operations on the AC or E registers.
- The next four instructions cause a skip of the next instruction in sequence when condition is satisfied. The skipping of the instruction is achieved by incrementing PC.
- The condition control statements must be recognized as part of the control conditions. The AC is positive when the sign bit in AC(15) = 0; it is negative when AC(15) = 1. The content of AC is zero (AC = 0) if all the flip-flops of the register are zero.
- The HLT instruction clears a start-stop flip-flop S and stops the sequence counter from counting. To restore the operation of the computer, the start-stop flip-flop must be set manually.

<u>Microprogrammed control</u>

Hardwired Control Unit:

- When the control signals are generated by hardware using conventional logic design techniques, the control unit is said to be hardwired.

Micro programmed control unit:

- A control unit whose binary control variables are stored in memory is called a micro programmed control unit.

Dynamic microprogramming:

- A more advanced development known as *dynamic* microprogramming permits a micro program to be loaded initially from an auxiliary memory such as a magnetic disk.
- Control units that use dynamic microprogramming employ a writable control memory. This type of memory can be used for writing.

Control Memory:

- Control Memory is the storage in the micro programmed control unit to store the micro program.

Writeable Control Memory:

- Control Storage whose contents can be modified, allow the change in micro program and Instruction set can be changed or modified is referred as Writeable Control Memory.

Control Word:

- The control variables at any given time can be represented by a control word string of 1 's and 0's called a control word.

Micro operations:

- In computer central processing units, micro-operations (also known as a micro-ops or μops) are detailed low-level instructions used in some designs to implement complex machine instructions (sometimes termed macro-instructions in this context).

Micro instruction:

- A symbolic micro program can be translated into its binary equivalent by means of anassembler.
- Each line of the assembly language micro program defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, micro operations, CD, BR, and AD.

Micro program:

- A sequence of microinstructions constitutes a micro program.
- Since alterations of the micro program are not needed once the control unit is in operation, the control memory can be a read-only memory (ROM).
- ROM words are made permanent during the hardware production of the unit.
- The use of a micro program involves placing all control variables in words of ROM for use by the control unit through successive read operations.
- The content of the word in ROM at a given address specifies a microinstruction.

Microcode:

- Microinstructions can be saved by employing subroutines that use common sections ofmicrocode.
- For example, the sequence of micro operations needed to generate the effective address ofthe operand for an instruction is common to all memory reference instructions.
- This sequence could be a subroutine that is called from within many other routines toexecute the effective address computation.

Control Memory Organization:

- The general configuration of a micro-programmed control unit is demonstrated in the block diagram of Figure 4.1.
- The control memory is assumed to be a ROM, within which all control information is permanently stored.

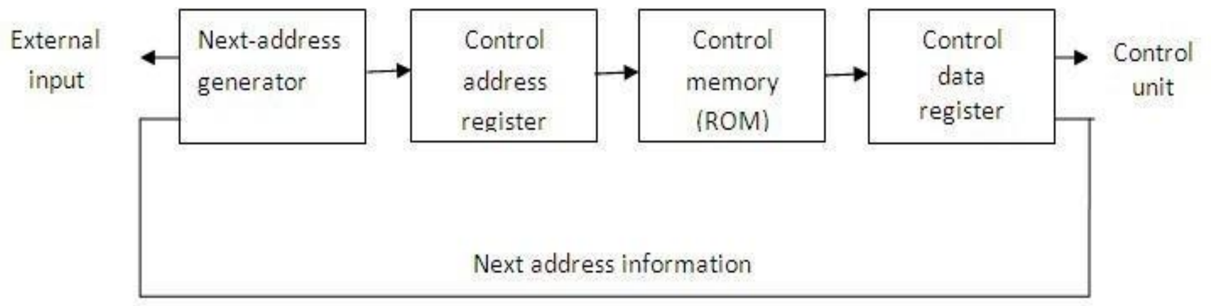


figure 4.1: Micro-programmed control organization

- The control memory address register specifies the address of the microinstruction, and the control data register holds the microinstruction read from memory.
- The microinstruction contains a control word that specifies one or more micro operations for the data processor. Once these operations are executed, the control must determine the next address.
- The location of the next microinstruction may be the one next in sequence, or it may be located somewhere else in the control memory.
- While the micro operations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.
- Thus a microinstruction contains bits for initiating micro operations in the data processor part and bits that determine the address sequence for the control memory.
- The next address generator is sometimes called a ***micro-program sequencer***, as it determines the address sequence that is read from control memory.
- Typical functions of a micro-program sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address, or loading an initial address to start the control operations.
- The control data register holds the present microinstruction while the next address is computed and read from memory.
- The data register is sometimes called a ***pipeline register***.
- It allows the execution of the micro operations specified by the control word simultaneously with the generation of the next microinstruction.
- This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.
- The main advantage of the micro programmed control is the fact that once the hardware configuration is established; there should be no need for further hardware or wiring changes.

- If we want to establish a different control sequence for the system, all we need to do is specify a different set of microinstructions for control memory.

Address Sequencing:

- Microinstructions are stored in control memory in groups, with each group specifying a routine.
- To appreciate the address sequencing in a micro-program control unit, let us specify the steps that the control must undergo during the execution of a single computer instruction.

Step-1:

- An initial address is loaded into the control address register when power is turned on in the computer.
- This address is usually the address of the first microinstruction that activates the instruction fetch routine.
- The fetch routine may be sequenced by incrementing the control address register through the rest of its microinstructions.
- At the end of the fetch routine, the instruction is in the instruction register of the computer.

Step-2:

- The control memory next must go through the routine that determines the effective address of the operand.
- A machine instruction may have bits that specify various addressing modes, such as indirect address and index registers.
- The effective address computation routine in control memory can be reached through a branch microinstruction, which is conditioned on the status of the mode bits of the instruction.
- When the effective address computation routine is completed, the address of the operand is available in the memory address register.

Step-3:

- The next step is to generate the micro operations that execute the instruction fetched from memory.
- The micro operation steps to be generated in processor registers depend on the operation code part of the instruction.
- Each instruction has its own micro-program routine stored in a given location of control memory.
- The transformation from the instruction code bits to an address in control memory where the routine is located is referred to as a ***mapping*** process.
- A mapping procedure is a rule that transforms the instruction code into a control memory address.

Step-4:

- Once the required routine is reached, the microinstructions that execute the instruction may be sequenced by incrementing the control address register.
- Micro-programs that employ subroutines will require an external register for storing the return address.
- Return addresses cannot be stored in ROM because the unit has no writing capability.
- When the execution of the instruction is completed, control must return to the fetch routine.
- This is accomplished by executing an unconditional branch microinstruction to the first address of the fetch routine.

In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control address register.
2. Unconditional branch or conditional branch, depending on status bit conditions.
3. A mapping process from the bits of the instruction to an address for control memory.
4. A facility for subroutine call and return.

Selection of address for control memory:

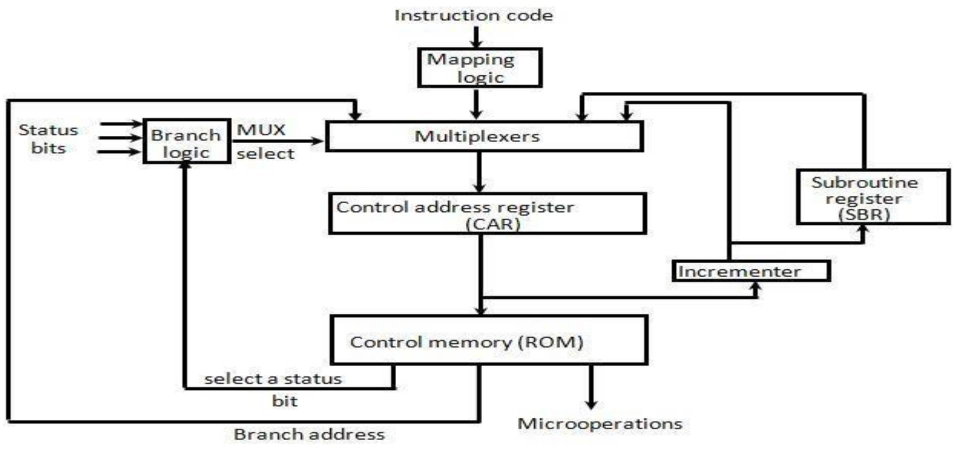


Figure 4.2: Selection of address for control memory

- Above figure 4.2 shows a block diagram of a control memory and the associated hardware needed for selecting the next microinstruction address.
- The microinstruction in control memory contains a set of bits to initiate micro operations in computer registers and other bits to specify the method by which the next address is obtained.
- The diagram shows four different paths from which the control address register (CAR) receives the address.
- The incrementer increments the content of the control address register by one, to select the next microinstruction in sequence.
- Branching is achieved by specifying the branch address in one of the fields of the microinstruction.
- Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition.
- An external address is transferred into control memory via a mapping logic circuit.
- The return address for a subroutine is stored in a special register whose value is then used when the micro-program wishes to return from the subroutine.
- The branch logic of figure 4.2 provides decision-making capabilities in the control unit.
- The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and input or output status conditions.
- The status bits, together with the field in the microinstruction that specifies a branch address, control the conditional branch decisions generated in the branch logic.
- A 1 output in the multiplexer generates a control signal to transfer the branch address from the microinstruction into the control address register.
- A 0 output in the multiplexer causes the address register to be incremented.

Mapping of an Instruction:

- A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine for an instruction is located.
- The status bits for this type of branch are the bits in the operation code part of the instruction.
- For example, a computer with a simple instruction format as shown in figure 4.3 has an operation code of four bits which can specify up to 16 distinct instructions.
- Assume further that the control memory has 128 words, requiring an address of seven bits.
- One simple mapping process that converts the 4-bit operation code to a 7-bit address for control memory is shown in figure 4.3.
- This mapping consists of placing a 0 in the most significant bit of the address, transferring the four operation code bits, and clearing the two least significant bits of the control address register.
- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions.
- If the routine needs more than four microinstructions, it can use addresses 1000000 through 1111111. If it uses fewer than four microinstructions, the unused memory locations would be available for other routines.
- One can extend this concept to a more general mapping rule by using a ROM to specify the mapping function.
- The contents of the mapping ROM give the bits for the control address register.
- In this way the micro program routine that executes the instruction can be placed in any desired location in control memory.
- The mapping concept provides flexibility for adding instructions for control memory as the need arises.

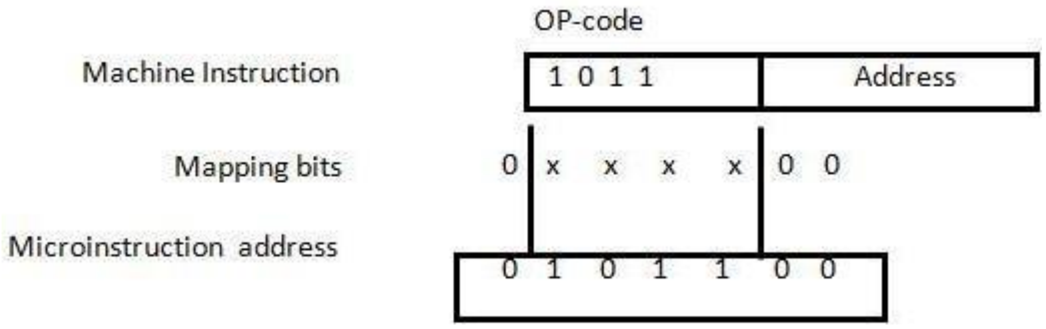


Figure 4.3: Mapping from instruction code to microinstruction address

Computer Hardware Configuration:

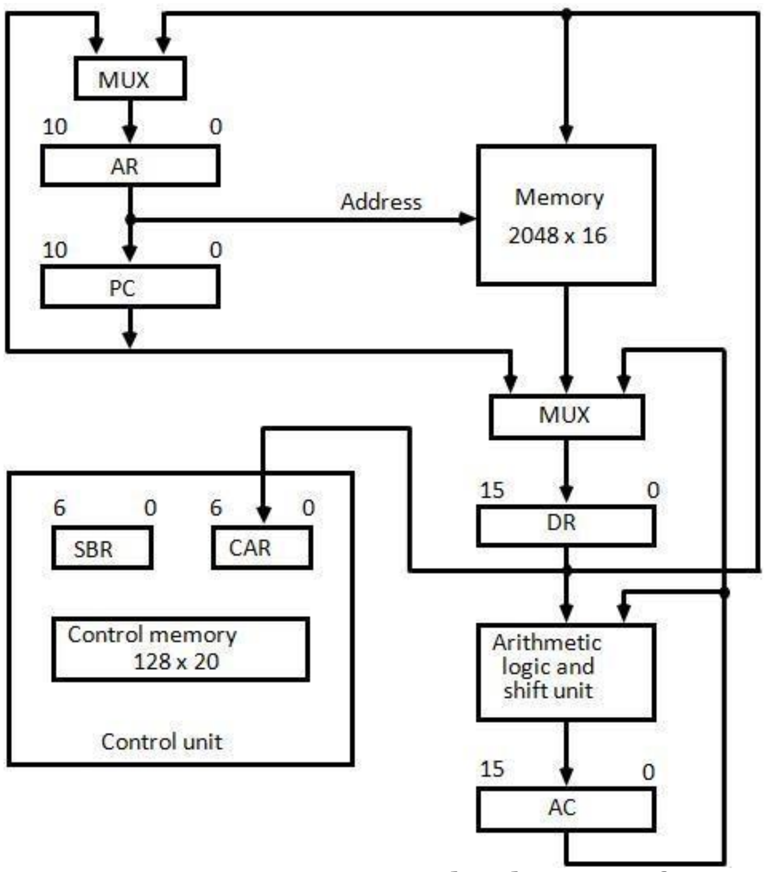


Figure 4.4: Computer hardware configuration

The block diagram of the computer is shown in Figure 4.4. It consists of,

1. Two memory units:

   Main memory -> for storing instructions and data,
   and Control memory -> for storing the micro
   program.
2. Six Registers:

   Processor unit register: AC(accumulator),PC(Program Counter), AR(Address Register), DR(Data Register)
   Control unit register: CAR (Control Address Register), SBR(Subroutine Register)
3. Multiplexers:

   The transfer of information among the registers in the processor is done through multiplexers rather than a common bus.
4. ALU:

   The arithmetic, logic, and shift unit performs microoperations with data from AC and DR and places the result in AC.DR can receive information from AC, PC, or memory.

- AR can receive information from PC or DR.
- PC can receive information only from AR.
- Input data written to memory come from DR, and data read from memory can go only to DR.

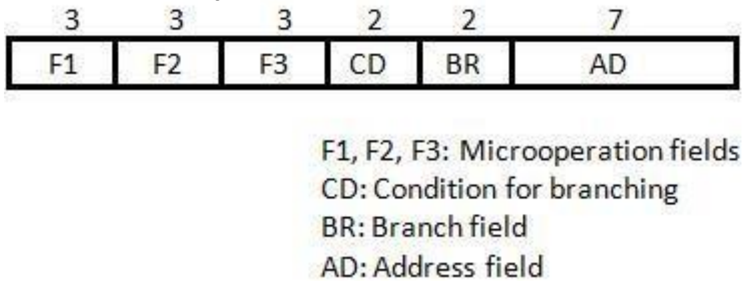
Microinstruction Format:

The microinstruction format for the control memory is shown in figure 4.5. The 20 bits of the microinstruction are divided into four functional parts as follows:

1. The three fields F1, F2, and F3 specify micro operations for the computer.
2. The micro operations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct micro operations. This gives a

total of 21 micro operations.

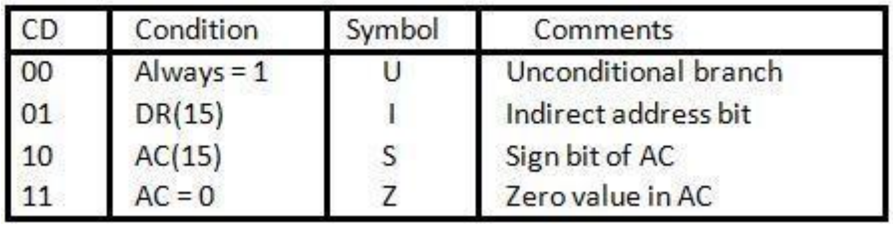
3. The CD field selects status bit conditions.
4. The BR field specifies the type of branch to be used.
5. The AD field contains a branch address. The address field is seven bits wide, since the control memory has 128 = 2^7 words.



F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

Figure 4.5: Microinstruction Format

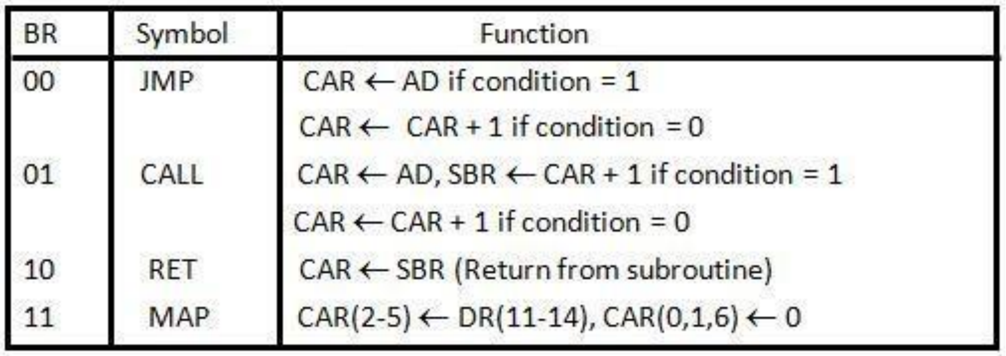
- As an example, a microinstruction can specify two simultaneous microoperations from F2 and F3 and none from F1.
  DR <- M[AR] with F2 = 100, PC <- PC + 1 with F3 = 101
- The nine bits of the micro operation fields will then be 000 100 101.
- The CD (condition) field consists of two bits which are encoded to specify four status bit conditions as listed in Table 4.1.



CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

Table 4.1: Condition Field

- The BR (branch) field consists of two bits. It is used, in conjunction with the address field AD, to choose the address of the next microinstruction shown in Table 4.2.



BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1
		$CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2\text{-}5) \leftarrow DR(11\text{-}14)$, $CAR(0,1,6) \leftarrow 0$

Table 4.2: Branch Field

Symbolic Microinstruction:

- Each line of the assembly language micro program defines a symbolic microinstruction.
- Each symbolic microinstruction is divided into five fields: label, micro operations, CD,BR, and AD. The fields specify the following Table 4.3.

1.	Label	The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2.	Micro operations	It consists of one, two, or three symbols, separated by commas, from those defined in Table 5.3. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no micro operations. This will be translated by the assembler to nine zeros.
3.	CD	The CD field has one of the letters U, I, S, or Z.
4.	BR	The BR field contains one of the four symbols defined in Table 5.2.
5.	AD	The AD field specifies a value for the address field of the microinstruction in one of three possible ways: i. With a symbolic address, this must also appear as a label. ii. With the symbol NEXT to designate the next address in sequence. iii. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

Table 4.3: Symbolic Microinstruction

Symbolic and Binary Micro Program:

- Control memory: 128, 20-bit words
- First 64 words: Routines for 16 machine instructions
- Last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- Mapping: Opcode XXXX into 0XXXX00, first address for 16 routines are (0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60.

✓ Return back to the fetch routine, the execution of the third (**MAP**) microinstruction in the fetch routine results in a branch to address **0xxxx00**, where **xxxx** represents the four bits of the operation code.

Example:

- Suppose that the instruction is **STORE** instruction whose operation code is **0010**. The **MAP** microinstruction will transfer to **CAR** the address **0 0010 00** (decimal **8**), which is the start address for the **STORE** routine in control memory.

- The first address for the **ADD,** BRANCH and **EXCHANGE** routines are **0 0000 00** (decimal **0**), **0 0001 00** (decimal 4), and **0 0011 00** (decimal **12**) respectively. The first address for the other **12** routines are at address values **16, 20, 24, ... , 60**. This gives four words in control memory for each routine.

- In each routine we must provide microinstructions for:

    - ***Evaluating the effective address***
    - ***Executing the instruction***

The indirect address mode is associated with all memory- reference instructions. For purpose of saving in the number of control memory words, the microinstructions for the indirect address are stored as a subroutine. This subroutine, symbolized by **INDRCT**, is located right after the fetch routine in the control memory.

Table 8.5 shows the symbolic micro program of the microinstruction routines that execute the four computer instructions (**ADD, BRANCH, STORE, and EXCHANGE**) and the **INDRCT** and **fetch routines**.

Label	Microoperations	CD	BR	AD
Symbolic Microprograms for Computer Instructions ADD, BRANCH, STORE, EXCHANGE				
	ORG 0			
ADD:	**NOP**	**I**	**CALL**	**INDRCT**
	READ	**U**	**JMP**	**NEXT**
	ADD	**U**	**JMP**	**FETCH**
	ORG 4			
BRANCH:	**NOP**	**S**	**JMP**	**OVER**
	NOP	**U**	**JMP**	**FETCH**
OVER:	**NOP**	**I**	**CALL**	**INDRCT**
	ARTPC	**U**	**JMP**	**FETCH**
	ORG 8			
STORE:	**NOP**	**I**	**CALL**	**INDRCT**
	ACTDR	**U**	**JMP**	**NEXT**
	WRITE	**U**	**JMP**	**FETCH**
	ORG 12			
EXCHANGE:	**NOP**	**I**	**CALL**	**INDRCT**
	READ	**U**	**JMP**	**NEXT**
	ACTDR, DRTAC	**U**	**JMP**	**NEXT**
	WRITE	**U**	**JMP**	**FETCH**
Symbolic Microprograms for Subroutines FETCH & INDRCT				
	ORG 64			
FETCH:	**PCTAR READ,**	**U**	**JMP**	**NEXT**
	INCPC	**U**	**JMP**	**NEXT**
	DRTAR	**U**	**MAP**	
INDRCT:	**READ**	**U**	**JMP**	**NEXT**
	DRTAR	**U**	**RET**	

Unit V:

- Central Processing Unit
  - General Register Organization
  - Stack Organization
  - Addressing Modes
  - Data transfer and Manipulation
  - Program Control
  - CISC and RISC
- Parallel Processing
  - Pipeline
  - General Consideration
- Input – Output Organization
  - Peripheral devices
  - I/O Interface
- Memory Organization
  - Memory Hierarchy
  - Main Memory
  - Auxiliary Memory

Central processing unit

Introduction:

- The main part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU.
- The CPU is made up of three major parts, as shown in Fig. 8-1.

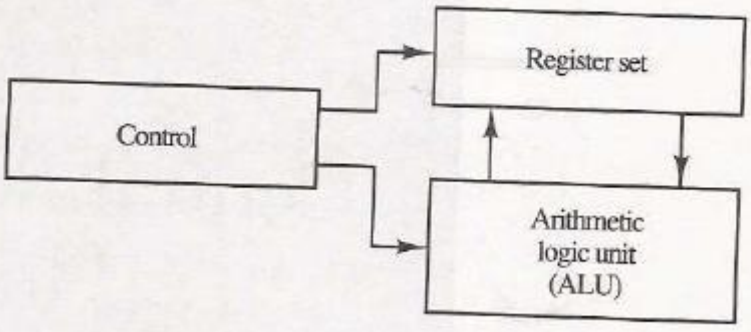


Figure 8-1 Major components of CPU.

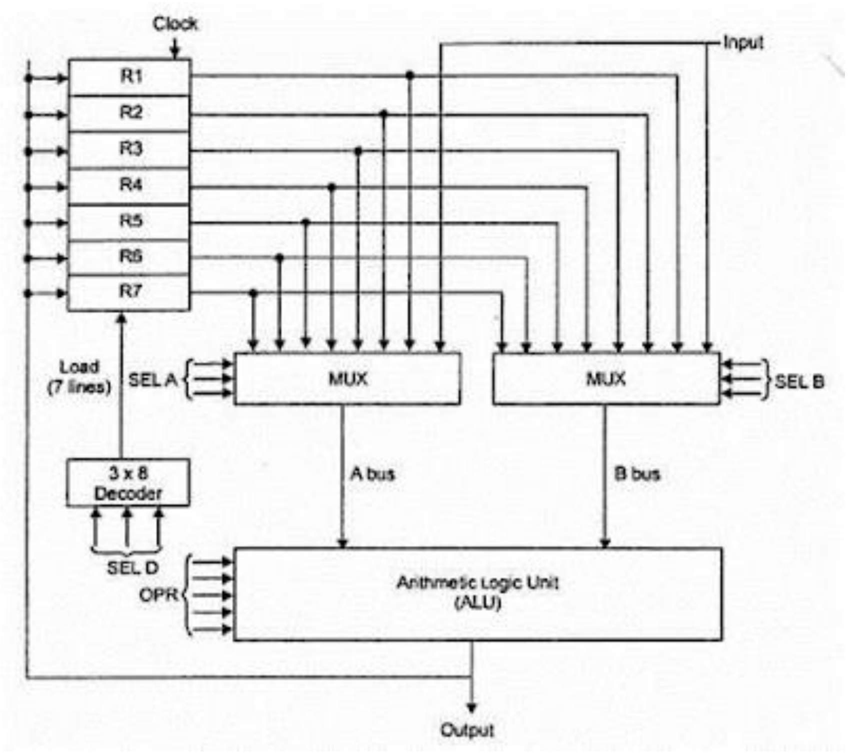
- The register set stores intermediate data used during the execution of the instructions.
- The arithmetic logic unit (ALU) performs the required micro operations for executing the instructions.
- The control unit supervises the transfer of information among the registers and instructs the ALU as to which operation to perform.

General Register organization

- Generally CPU has seven general registers. Register organization show how registers are selected and how data flow between register and ALU. A decoder is used to select a particular register.

- The output of each register is connected to two multiplexers to form the two buses A and B. The selection lines in each multiplexer select the input data for the particular bus.
- The A and B buses form the two inputs of an ALU. The operation select lines decide the micro operation to be performed by ALU. The result of the micro operation is available at the output bus. The output bus connected to the inputs of all registers, thus by selecting a destination register it is possible to store the result in it.

A bus organization for seven CPU register:



EXAMPLE:

- To perform the operation ***R3 = R1+R2*** We have to provide following binary selection variable to the select inputs.
    1. **SEL A** : **001** -To place the contents of R1 into bus A.
    2. **SEL B** : **010** - to place the contents of R2 into bus B
    3. **SEL OPR** : **10010** – to perform the arithmetic addition A+B
    4. **SEL REG or SEL D : 011** – to place the result available on output bus in R3.

Register and multiplexer input selection code

Binary code	**SELA**	**SELB**	**SEL-D or SEL-REG**
000	Input	Input	---
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4

101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

Operation with symbol

Operation selection code	**Operation**	**symbol**
0000	Transfer A	TSFA
0001	Increment A	INC A
0010	A+B	ADD
0011	A-B	SUB
0100	Decrement A	DEC
0101	A AND B	AND
0110	A OR B	OR
0111	A XOR B	XOR
1000	Complement A	COMA
1001	Shift right A	SHR
1010	Shift left A	SHL

Control word:

- The combined value of a binary selection inputs specifies the control word.
- It consists of four fields SELA, SELB and SELD or SELREG contains three bit each and SELOPR field contains four bits thus the total bits in the control word are 13-bits.

SEL A	**SELB**	**SELREG OR SELD**	**SELOPR**

Format of control word:

- The three bit of SELA select a source registers of the a input of the ALU.
- The three bits of SELB select a source registers of the b input of the ALU.
- The three bits of SELED or SELREG select a destination register using the decoder.
- The four bits of SELOPR select the operation to be performed by ALU.

CONTROL WORD FOR OPERATION R2 = R1+R3:

SEL A	**SEL B**	**SEL- D OR SEL- REG**	**SEL -OPR**
001	011	010	0010

MICROOPERATION	**SEL A**	**SEL B**	**SEL D OR SEL REG**	**SELOPR**	**CONTROL WORD**			
R2 = R1+R3	R1	R3	R2	ADD	001	011	010	0010

Stack Organization:

- A stack or last-in first-out (LIFO) is useful feature that is included in the CPU of most computers.
- Stack:
    - 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 of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off.
- In the computer stack is a memory unit with an address register that can count the address only.
- The register that holds the address for the stack is called a stack pointer (SP). It always points at the top item in the stack.
- The two operations that are performed on stack are the insertion and deletion.
- The operation of insertion is called *PUSH.*
- The operation of deletion is called *POP.*
- These operations are simulated by incrementing and decrementing the stack pointer register (SP).

Register Stack:

- A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.

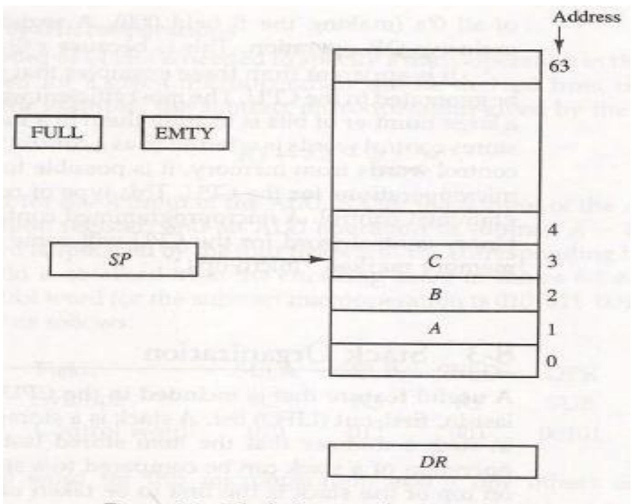


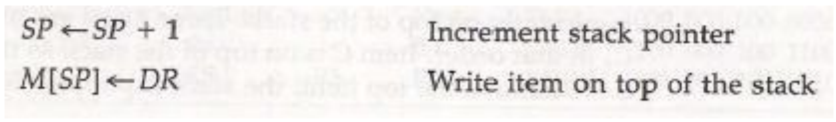
Figure 8-3 Block diagram of a 64-word stack.

- The below figure shows the organization of a 64-word register stack.
- The stack pointer register SP contains a binary number whose value is equal to the address of the word is currently on top of the stack. Three items are placed in the stack: A, *B, C, in* that order.
- In above figure C is on top of the stack so that the content of *SP* is 3.
- For removing the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of stack SP.
- Now the top of the stack is B, so that the content of SP is 2.
- Similarly for inserting the new item, the stack is pushed by incrementing SP and writing a word in the next- higher location in the stack.
- In a 64-word stack, the stack pointer contains 6 bits because $2^6 = 64$.
- Since *SP* has only six bits, it cannot exceed a number greater than 63 (111111 in binary).
- When 63 is incremented by 1, the result is 0 since 111111 + 1. = 1000000 in binary, but SP can accommodate only the six least significant bits.

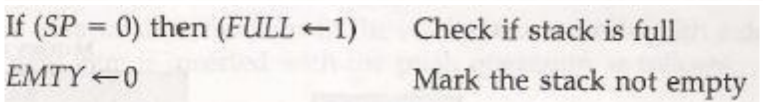
- Then the one-bit register FULL is set to 1, when the stack is full.
- Similarly when 000000 is decremented by 1, the result is 111111, and then the one-bit register EMTY is set 1 when the stack is empty of items.
- DR is the data register that holds the binary data to be written into or read out of the stack.

PUSH:

- Initially, *SP* is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, so that SP points to the word at address 0 and the stack is marked empty and not full.
- If the stack is not full (if FULL = 0), a new item is inserted with a push operation.
- The push operation is implemented with the following sequence of microoperations:



$SP \leftarrow SP + 1$	Increment stack pointer
$M[SP] \leftarrow DR$	Write item on top of the stack

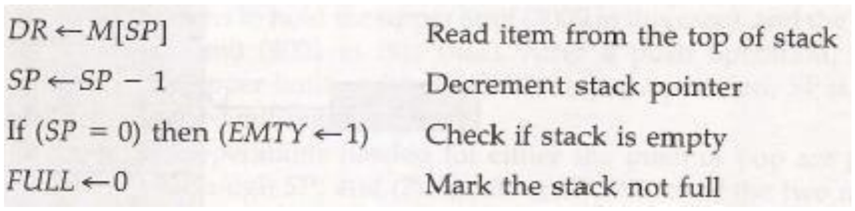


If $(SP = 0)$ then $(FULL \leftarrow 1)$	Check if stack is full
$EMTY \leftarrow 0$	Mark the stack not empty

- The stack pointer is incremented so that it points to the address of next-higher word.
- A memory write operation inserts the word from DR the top of the stack.
- The first item stored in the stack is at address 1.
- The last item is stored at address 0.
- If *SP* reaches 0, the stack is full of items, so FULL is to 1.
- This condition is reached if the top item prior to the last push way location 63 and, after incrementing *SP,* the last item is stored in location 0.
- Once an item is stored in location 0, there are no more empty registers in the stack, so the EMTY is cleared to 0.

POP:

- A new item is deleted from the stack if the stack is not empty (if EMTY = 0).
- The pop operation consists of the following sequence of min operations:



$DR \leftarrow M[SP]$	Read item from the top of stack
$SP \leftarrow SP - 1$	Decrement stack pointer
If $(SP = 0)$ then $(EMTY \leftarrow 1)$	Check if stack is empty
$FULL \leftarrow 0$	Mark the stack not full

- The top item is read from the stack into DR.
- The stack pointer is then decremented. If its value reaches zero, the stack is empty, so EMTY is set 1.
- This condition is reached if the item read was in location 1. Once this it is read out, SP is decremented and reaches the value 0, which is the initial value of SP.
- If a pop operation reads the item from location 0 and then is decremented, *SP* changes to 111111, which is equivalent to decimal 63 in above configuration, the word in address 0 receives the last item in the stack.

Memory Stack:

- In the above discussion a stack can exist as a stand-alone unit. But in the CPU implementation of a stack is done by assigning a portion of memory to a stack operation and using a processor register as stack pointer.

- The below figure shows a portion computer memory partitioned into three segments: program, data, and stack.

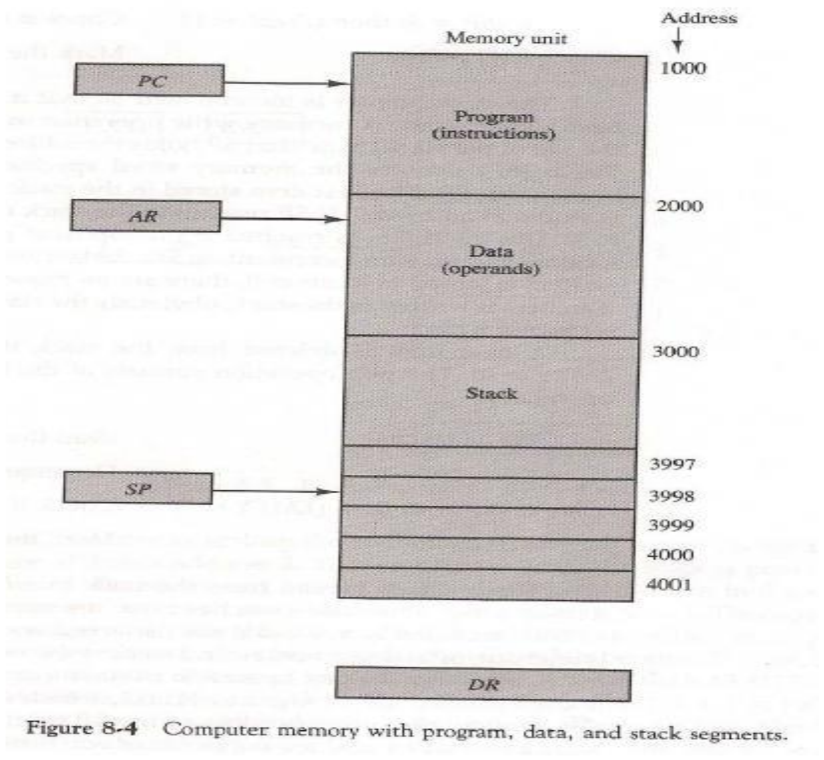


Figure 8-4 Computer memory with program, data, and stack segments.

- The program counter *PC* points at the address of the next instruction in program.
- The address register AR points at an array of data.
- The stack pointer SP points at the top of the stack.
- The three registers are connected to a common address bus, 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 exec phase to read an operand.
    - *SP* is used to push or pop items into or from stack.
- As shown in Fig. 8-4, the initial value of SP is 4001 and the stack grows with decreasing addresses.
- Thus the first item stored in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000.
- No provisions are available for stack limit checks.
- The items in the stack communicate with a data register *DR.* A new item is inserted with the push operation as follows:

SP ← SP-1

M [SP] ← DR

- The stack pointer is decremented so that it points at the address of the next word.
- A memory write operation inserts the word from DR into the top of stack. A new item is deleted with a pop operation as follows:

DR ← M [SP]
SP ← SP+1

- The top item is read from the stack into DR. The stack pointer is then decremented to point at the next item in the stack.
- Most computers do not provide hardware to check for stack overflow (full stack) or underflow (empty stack).

- The stack limits can be checked by using processor registers:
  - one to hold the upper limit (3000 in this case)
  - Other to hold the lower limit (4001 in this case).
- After a push operation, *SP* compared with the upper-limit register and after a pop operation, *SP* is a compared with the lower-limit register.
- The two micro operations needed for either the push or pop are

  (1) An access to memory through SP (2) Updating SP
- The advantage of a memory stack is that the CPU can refer to it without having specify an address, since the address is always available and automatically updated in the stack pointer.

Reverse Polish Notation:

- A stack organization is very effective for evaluating arithmetic expressions.
- The common arithmetic expressions are written in *infix notation,* with each operator written *between* the operands.
- Consider the simple arithmetic expression.

  A*B+C*D
- For evaluating the above expression it is necessary to compute the product A*B, store this product result while computing C*D, and then sum the two products.
- For doing this type of infix notation, it is necessary to scan back and forth along the expression to determine the next operation to be performed.
- The Polish mathematician Lukasiewicz showed that arithmetic expression can be represented in *prefix notation.*
- This representation, often referred to as *Polish notation,* places the operator before the operands. So it is also called as *prefix notation*.
- The *Postfix notation,* referred to as *reverse Polish notation (RPN),* places the operator after the operands.
- The following examples demonstrate the three representations

  Eg: A+B ----- > Infix notation

  +AB -------> Prefix or Polish notation

  AB+ --------> Post or reverse Polish notation
- The reverse Polish notation is in a form suitable for stack manipulation. The expression

  A*B+C*D

  Is written in reverse polish notation as

  AB* CD* +

And it is evaluated as follows

- ✓ Scan the expression from left to right.
- ✓ When operator is reached, perform the operation with the two operands found on the left side of the operator.
- ✓ Remove the two operands and the operator and replace them by the number obtained from the result of the operation.
- ✓ Continue to scan the expression and repeat the procedure for every operation encountered until there are no more operators.

- For the expression above it find the operator * after A and B. So it perform the operation A*B and replace A, B and * with the result.
- The next operator is a * and it previous two operands are C and D, so it perform the operation C*D and places the result in places C, D and *.
- The next operator is + and the two operands to be added are the two products, so we add the two quantities to obtain the result.
- The conversion from infix notation to reverse Polish notation must take into consideration the operational hierarchy adopted for infix notation.

- This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations before addition and subtraction operations.

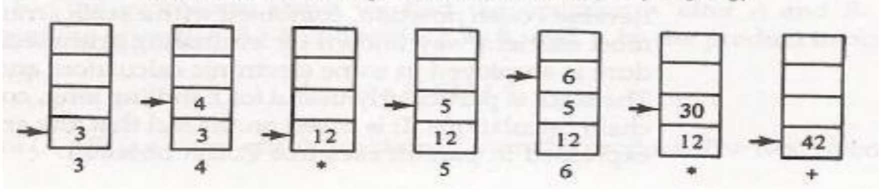
Evaluation of Arithmetic Expressions:

- Reverse Polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions.
- This procedure is employed in some electronic calculators and also in some computer.
- The following numerical example may clarify this procedure. Consider the arithmetic expression

  (3*4) + (5*6)

  In reverse polish notation, it is
  expressed as 34
  * 56* +
- Now consider the stack operations shown in Fig. 8-5.

Figure 8-5 Stack operations to evaluate 3 • 4 + 5 • 6.



- Each box represents one stack operation and the arrow always points to the top of the stack.
- Scanning the expression from left to right, we encounter two operands.
- First the number 3 is pushed into the stack, then the number 4.
- The next symbol is the multiplication operator *.
- This causes a multiplication of the two top most items the stack.
- The stack is then popped and the product is placed on top of the stack, replacing the two original operands.
- Next we encounter the two operands 5 and 6, so they are pushed into the stack.
- The stack operation results from the next * replaces these two numbers by their product.
- The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

Instruction Formats:

- The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register.
- The bits of the instruction are divided into groups called fields.
- The most common fields found in instruction formats are:
  1. An operation code field that specifies the operation to be perform
  2. An address field that designates a memory address or a processor register.
  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 fields in the instruct format of a computer depends on the internal organization of its registers.

➢ Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization.
2. General register organization.
3. Stack organization.

Single Accumulator Organization:

- ✓ In an accumulator type organization all the operations are performed with an implied accumulator register.
- ✓ The instruction format in this type of computer uses one address field.
- ✓ For example, the instruction that specifies an arithmetic addition defined by an assembly language instruction as
  - **ADD X**
- ✓ Where X is the address of the operand. The ADD instruction in this case results in the operation *AC* *AC* *+M[X]. AC* is the accumulator register and *M[X]* symbolizes the memory word located at address X.

General register organization:

- ✓ The instruction format in this type of computer needs three register address fields.
- ✓ Thus the instruction for an arithmetic addition may be written in an assembly language as

  ADD R1, R2, R3

  to denote the operation R1R2 + R3. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers.
- ✓ Thus the instruction **ADD R1, R2** would denote the operation R1 *R1 +* R2. Only register addresses for R1 and R2 need be specified in this instruction.
- ✓ General register-type computers employ two or three address fields in their instruction format.
- ✓ Each address field may specify a processor register or a memory word.
- ✓ An instruction symbolized by **ADD R1, X** would specify the operation R1 R1 + M[X].
- ✓ It has two address fields, one for register R1 and the other for the memory address X.

Stack organization:

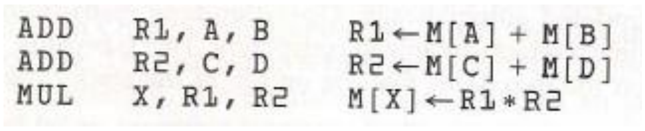
- ✓ The stack-organized CPU has PUSH and POP instructions which require an address field.
- ✓ Thus the instruction **PUSH X** will push the word at address X to the top of the stack.
- ✓ The stack pointer is updated automatically.
- ✓ Operation-type instructions do not need an address field in stack-organized computers.
- ✓ This is because the operation is performed on the two items that are on top of the stack.
- ✓ The instruction **ADD** in a stack computer consists of an operation code only with no address field.
- ✓ This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack.
- ✓ There is no need to specify operands with an address field since all operands are implied to be in the stack.

- Most computers fall into one of the three types of organizations.
- Some computers combine features from more than one organizational structure.
- The influence of the number of addresses on computer programs, we will evaluate the arithmetic statement

  X= (A+B) * (C+D)
- Using zero, one, two, or three address instructions and using the symbols ADD, SUB, MUL and DIV for four arithmetic operations; MOV for the transfer type operations; and LOAD and STORE for transfer to and from memory and AC register.
- Assuming that the operands are in memory addresses A, B, C, and D and the result must be stored in memory ar address X and also the CPU has general purpose registers R1, R2, R3 and R4.

Three Address Instructions:

- ✓ Three-address instruction formats can use each address field to specify either a processor register or a memory operand.
- ✓ The program assembly language that evaluates **X = (A+B) * *(C+*D**) is shown below, together with comments that explain the register transfer operation of each instruction.

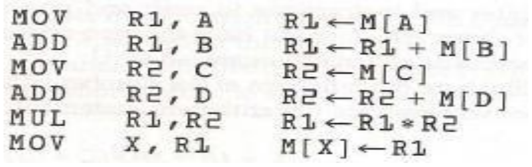


```
ADD    R1, A, B      R1←M[A] + M[B]
ADD    R2, C, D      R2←M[C] + M[D]
MUL    X, R1, R2     M[X]←R1*R2
```

- ✓ The symbol M [A] denotes the operand at memory address symbolized by A.
- ✓ The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions.
- ✓ The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two Address Instructions:

- ✓ Two-address instructions formats use each address field can specify either a processor register or memory word.
- ✓ The program to evaluate X = (A+B) * *(C+*D) is as follows



```
MOV    R1, A     R1←M[A]
ADD    R1, B     R1←R1 + M[B]
MOV    R2, C     R2←M[C]
ADD    R2, D     R2←R2 + M[D]
MUL    R1,R2     R1←R1*R2
MOV    X, R1     M[X]←R1
```

- ✓ The MOV instruction moves or transfers the operands to and from memory and processor registers.
- ✓ The first symbol listed in an instruction is assumed be both a source and the destination where the result of the operation transferred.

One Address Instructions:

- ✓ One-address instructions use an implied accumulator *(AC)* register for all data manipulation.
- ✓ For multiplication and division there is a need for a second register. But for the basic discussion we will neglect the second register and assume that the *AC* contains the result of all operations.
- ✓ The program to evaluate X=(A+B) * *(C+*D) is

```
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
```

- ✓ All operations are done between the AC register and a memory operand.
- ✓ T is the address of a temporary memory location required for storing the intermediate result.

Zero Address Instructions:

- ✓ A stack-organized computer does not use an address field for the instructions ADD and MUL.
- ✓ The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack.
- ✓ The following program shows how X = (A+B) * *(C+*D) will be written for a stack-organized computer.
  (TOS stands for top of 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
```

- ✓ To evaluate arithmetic expressions in a stack computer, it is necessary to convert the expression into reverse Polish notation.
- ✓ The name "zero-address" is given to this type of computer because of the absence of an address field in the computational instructions.

RISC Instructions:

- ✓ The instruction set of a typical RISC processor is use only load and store instructions for communicating between memory and CPU.
- ✓ All other instructions are executed within the registers of CPU without referring to memory.
- ✓ LOAD and STORE instructions that have one memory and one register address, and computational type instructions that have three addresses with all three specifying processor registers.
- ✓ The following is a program to evaluate X=(A+B)*(C+D)

```
LOAD    R1, A         R1←M[A]
LOAD    R2, B         R2←M[B]
LOAD    R3, C         R3←M[C]
LOAD    R4, D         R4←M[D]
ADD     R1, R1, R2    R1←R1 + R2
ADD     R3, R3, R2    R3←R3 + R4
MUL     R1, R1, R3    R1←R1*R3
STORE   X, R1         M[X]←R1
```

- ✓ The load instructions transfer the operands from memory to CPU register.
- ✓ The add and multiply operations are executed with data in the register without accessing memory.
- ✓ The result of the computations is then stored memory with a store in instruction.

Addressing Modes:

- The way the operands are chosen during program execution is dependent on the addressing mode of the instruction.
- Computers use addressing mode techniques for the purpose of accommodating one or both of
  the following provisions:
  - To give programming versatility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
  - To reduce the number of bits in the addressing field of the instruction
- Most addressing modes modify the address field of the instruction; there are two modes that need no address field at all. These are *implied* and *immediate* modes.

Implied Mode:

- ✓ In this mode the operands are specified 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.
- ✓ All register reference instructions that use an accumulator are implied mode instructions.
- ✓ Zero address in a stack organization computer is implied mode instructions.

Immediate Mode:

- ✓ In this mode the operand is specified in the instruction itself.
- ✓ In other words an immediate-mode instruction has an operand rather than an address field.
- ✓ Immediate-mode instructions are useful for initializing registers to a constant value.
  - The address field of an instruction may specify either a memory word or a processor register.
  - When the address specifies a processor register, the instruction is said to be in the register mode.

Register Mode:

- ✓ In this mode the operands are in registers that reside within the CPU.
- ✓ The particular register is selected from a register field in the instruction.

Register Indirect Mode:

- ✓ In this mode the instruction specifies a register in CPU whose contents give the address of the operand in memory.
- ✓ In other words, the selected register contains the address of the operand rather than the operand itself.
- ✓ The advantage of a register indirect mode instruction is that the address field of the instruction uses few bits to select a register than would have been required to specify a memory address directly.

Auto-increment or Auto-Decrement Mode:

- ➢ This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.

- The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory.
- Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated.
- The basic two mode of addressing used in CPU are *direct* and *indirect* address mode.

Direct Address Mode:

- ✓ In this mode the effective address is equal to the address part of the instruction.
- ✓ The operand resides in memory and its address is given directly by the address field of the instruction.
- ✓ In a branch-type instruction the address field specifies the actual branch address.

Indirect Address Mode:

- ✓ In this mode the address field of the instruction gives the address where the effective address is stored in memory.
- ✓ Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

- o A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU.
- o The effective address in these modes is obtained from the following computation:
  Effective address =address part of instruction + content of CPU register
- o The CPU register used in the computation may be the program counter, an index register, or a base register.
- o We have a different addressing mode which is used for a different application.

Relative Address Mode:

- ✓ In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

Indexed Addressing Mode:

- ✓ In this mode the content of an index register is added to the address part of the instruction to obtain the effective address.
- ✓ An index register is a special CPU register that contains an index value.

Base Register Addressing Mode:

- ✓ In this mode the content of a base register is added to the address part of the instruction to obtain the effective address.
- ✓ This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

Data Transfer and Manipulation:

- Most computer instructions can be classified into three categories:

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

Data Transfer Instructions:

- Data transfer instructions move data from one place in the computer to another without changing the data content.
- The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers themselves.
- Table 8-5 gives a list of eight data transfer instructions used in many computers.

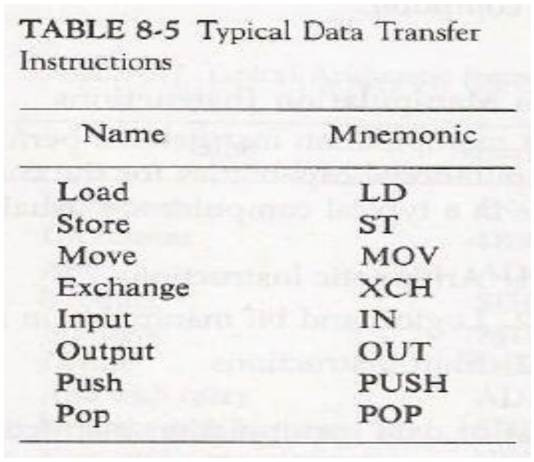


TABLE 8-5 Typical Data Transfer Instructions

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

- The *l**oad*** instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator.
- The ***store*** instruction designates a transfer from a processor register into memory.
- The ***move*** instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another and also between CPU registers and memory or between two memory words.
- The ***exchange*** instruction swaps information between two registers or a register and a memory word.
- The *i**nput*** and ***outpu**t* instructions transfer data among processor registers and input or output terminals.
- The ***push*** and ***pop*** instructions transfer data between processor registers and a memory stack.
- Different computers use different mnemonics symbols for differentiate the addressing modes.
- As an example, consider the *load to accumulator* instruction when used with eight different addressing modes.
- Table 8-6 shows the recommended assembly language convention and actual transfer accomplished in each case
- *ADR* stands for an address.
- *NBA* a number or operand.
- X is an index register.
- The @ character symbolizes an indirect addressing.
- R1 is a processor register.
- AC is the accumulator register.
- The $ character before an address makes the address relative to the program counter *PC.*
- The # character precedes the operand in an immediate-mode instruction.

Data Manipulation Instructions:

- Data manipulation 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 and bit manipulation instructions
3. Shift instructions

Arithmetic instructions

- ✓ The four basic arithmetic operations are addition, subtraction, multiplication and division.
- ✓ Most computers provide instructions for all four operations.
- ✓ Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by mean software subroutines.
- ✓ A list of typical arithmetic instructions is given in Table 8-7.

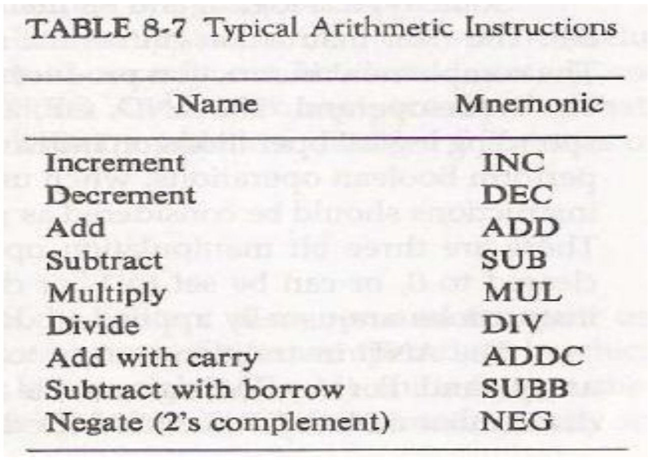


TABLE 8-7 Typical Arithmetic Instructions

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

- ✓ The increment instruction adds 1 to the value stored in a register or memory word.
- ✓ A number with all 1's, when incremented, produces a number with all 0's.
- ✓ The decrement instruction subtracts 1 from a value stored in a register or memory word.
- ✓ A number with all 0's, when decremented, produces number with all 1's.
- ✓ The add, subtract, multiply, and divide instructions may be use different types of data.
- ✓ The data type assumed to be in processor register during the execution of these arithmetic operations is defined by an operation code.
- ✓ An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.
- ✓ The mnemonics for three add instructions that specify different data
  types are shown below. ADDI Add two binary integer
  numbers
  ADDF Add two floating-point
  numbers ADDD Add two
  decimal numbers in BCD
- ✓ A special carry flip-flop is used to store the carry from an operation.
- ✓ The instruction "add carry" performs the addition on two operands plus the value of the carry the previous computation.
- ✓ Similarly, the "subtract with borrow" instruction subtracts two words and borrow which may have resulted from a previous subtract operation.

- ✓ The negate instruction forms the 2's complement number, effectively reversing the sign of an integer when represented it signed-2's complement form.

Logical and bit manipulation instructions

- ✓ Logical instructions perform binary operations on strings of bits store, registers.
- ✓ They are useful for manipulating individual bits or a group of that represent binary-coded information.
- ✓ The logical instructions consider each bit of the operand separately and treat it as a Boolean variable.
- ✓ By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in register memory words.
- ✓ Some typical logical and bit manipulation instructions are listed in Table 8-8.

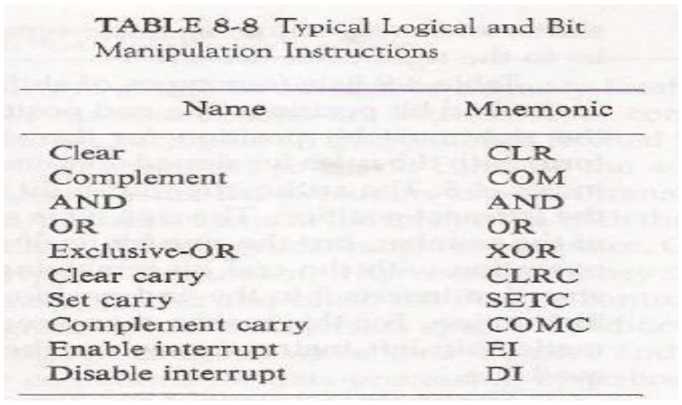


TABLE 8-8 Typical Logical and Bit Manipulation Instructions

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

- ✓ The clear instruction causes the specified operand to be replaced by 0's.
- ✓ The complement instruction produces the 1's complement by inverting all bits of the operand.
- ✓ The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands.
- ✓ The logical instructions can also be used to performing bit manipulation operations.
- ✓ There are three bit manipulation operations possible: a selected bit can cleared to 0, or can be set to 1, or can be complemented.
    - o The AND instruction is used to clear a bit or a selected group of bits of an operand.
    - o The OR instruction is used to set a bit or a selected group of bits of an operand.
    - o Similarly, the XOR instruction is used to selectively complement bits of an operand.
- ✓ Other bit manipulations instructions are included in above table perform the operations on individual bits such as a carry can be cleared, set, or complemented.
- ✓ Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

Shift Instructions

- ✓ 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 logical shifts, arithmetic shifts, or rotate-type operations.
- ✓ In either case the shift may be to the right or to the left.
- ✓ Table 8-9 lists four types of shift instructions.

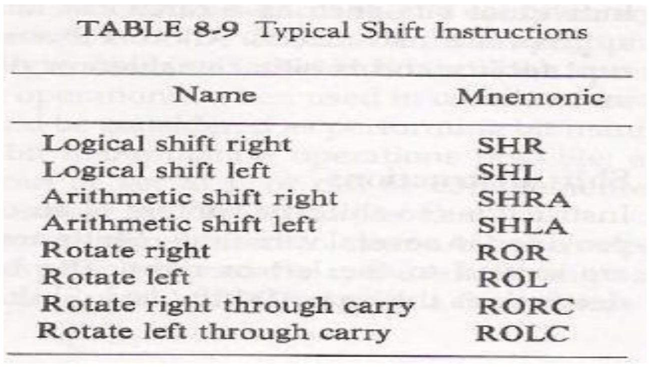


TABLE 8-9 Typical Shift Instructions

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

- ✓ The logical shift inset to the end bit position.
- ✓ The end position is the leftmost bit position for shift rights the rightmost bit position for the shift left.
- ✓ Arithmetic shifts usually conform to 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 unchanged.
- ✓ This is a shift-right operation with the end bit remaining the same.
- ✓ The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-instruction.
- ✓ The rotate instructions produce a circular shift. Bits shifted out at one of the word are not lost as in a logical shift but are circulated back into the other end.
- ✓ 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, shift the entire register to the left.

Program Control:

- o Program control instructions specify conditions for altering the content of the program counter.
- o The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution.
- o This instruction provides control over the flow of program execution and a capability for branching to different program segments.
- o Some typical program control instructions are listed in Table 8.10.

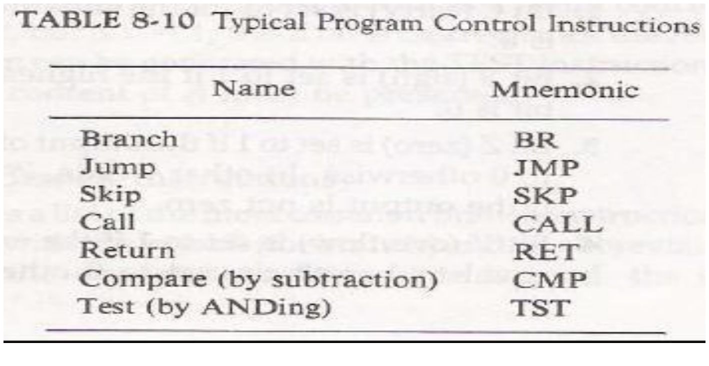


TABLE 8-10 Typical Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

- o Branch and jump instructions may be conditional or unconditional.
- o An unconditional branch instruction causes a branch to the specified address without

any conditions.

- The conditional branch instruction specifies a condition such as branch if positive or branch if zero.
- The skip instruction does not need an address field and is therefore a zero-address instruction.
- A conditional skip instruction will skip the next instruction if the condition is met. This is accomplished by incrementing program counter.
- The call and return instructions are used in conjunction with subroutines.
- The compare instruction forms a subtraction between two operands, but the result of the operation not retained. However, certain status bit conditions are set as a result of operation.
- Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands.

Status Bit Conditions:

- The ALU circuit in the CPU have status register for storing the status bit conditions.
- Status bits are also called *condition-code* bits or *flag* bits.

Figure 8-8 shows block diagram of an 8-bit ALU with a 4-bit status register.

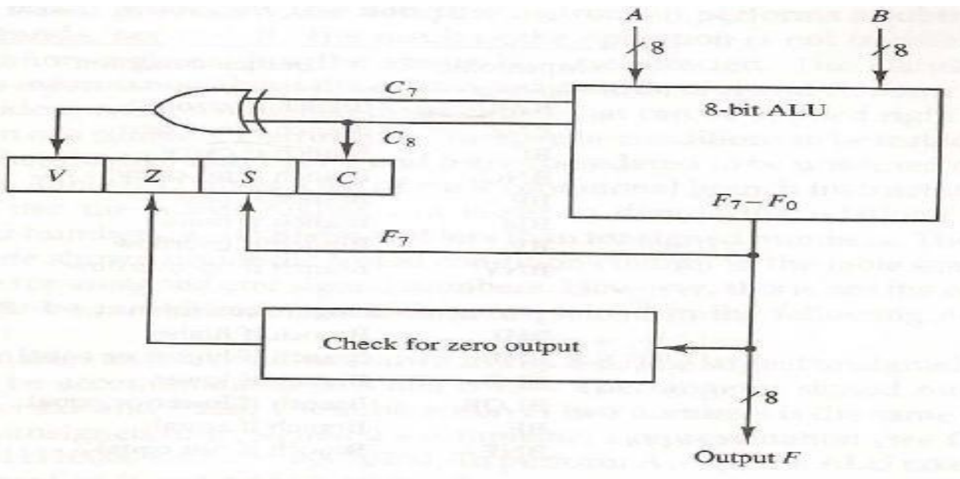


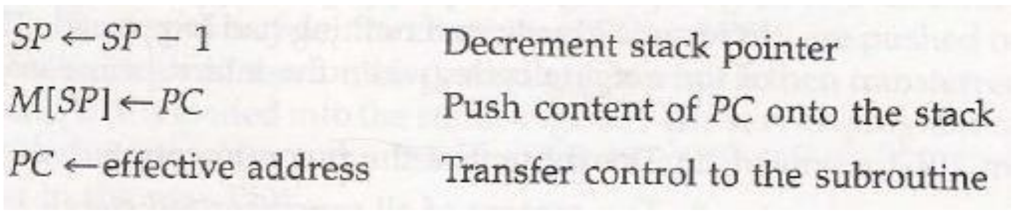
Figure 8-8 Status register bits.

- The four status bits are symbolized by C, S, *Z,* and V. The bits are set or cleared as a result of an operation performed in the ALU.
  - Bit C (carry) is set to 1 if the end carry C_8 is 1. It is cleared to 0 if the carry is 0.
  - S (sign) is set to 1 if the highest-order bit F_7 is 1. It is set to 0 if the bit is 0.
  - Bit *Z* (zero) is set to 1 if the output of the ALU contains all 0's. It is clear to 0 otherwise. In other words, *Z* = 1 if the output is zero and *Z =0 if the output is not zero.*
  - Bit V (overflow) is set to 1 if the exclusive-OR of the last two carries equal to 1, and cleared to 0 otherwise.
- The above status bits are used in conditional jump and branch instructions.

Subroutine Call and Return:

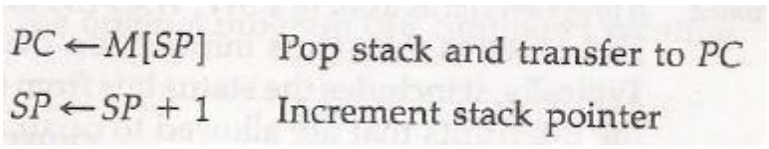
- A subroutine is self contained sequence of instructions that performs a given computational task.
- The most common names used are call subroutine, jump to subroutine, branch to subroutine, or branch and save return address.

- A subroutine is executed by performing two operations
  - The address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return
  - Control is transferred to the beginning of the subroutine.
- The last instruction of every subroutine, commonly called *return from subroutine,* transfers the return address from the temporary location in the program counter.
- Different computers use a different temporary location for storing the return address.
- The most efficient way is to store the return address in a memory stack.
- The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack.
- A subroutine call is implemented with the following microoperations:



$SP \leftarrow SP - 1$	Decrement stack pointer
$M[SP] \leftarrow PC$	Push content of PC onto the stack
$PC \leftarrow$effective address	Transfer control to the subroutine

- The instruction that returns from the last subroutine is implemented by the microoperations:



$PC \leftarrow M[SP]$	Pop stack and transfer to PC
$SP \leftarrow SP + 1$	Increment stack pointer

Program Interrupt:

- Program interrupt refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request.
- The interrupt procedure is similar to a subroutine call except for three variations:
  - The interrupt is initiated by an internal or external signal.
  - Address of the interrupt service program is determined by the hardware.
  - An interrupt procedure usually stores all the information rather than storing only PC content.

Types of interrupts:

- ✓ There are three major types of interrupts that cause a break in the normal execution of a program.
- ✓ They can be classified as
  - **External interrupts:**
    - These come from input—output (I/O) devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.
    - Ex: I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or power failure.
  - **Internal interrupts:**
    - These arise from illegal or erroneous use of an instruction or data.

- Internal interrupts are also called *traps.*
- Ex: interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

✓ Internal and external interrupts are initiated nals that occur in hardware of CPU.

- **Software interrupts**
  - A software interrupt is initiated by executing an instruction.
  - Software interrupt is a special call instruction that behaves like an interrupt rather than a subroutine call.

Reduced Instruction Set Computer:

- A computer with large number instructions is classified as a *complex instruction set computer,* abbreviated as CISC.
- The computer which having the fewer instructions is classified as a *reduced instruction set computer,* abbreviated as RISC.

CISC Characteristics:

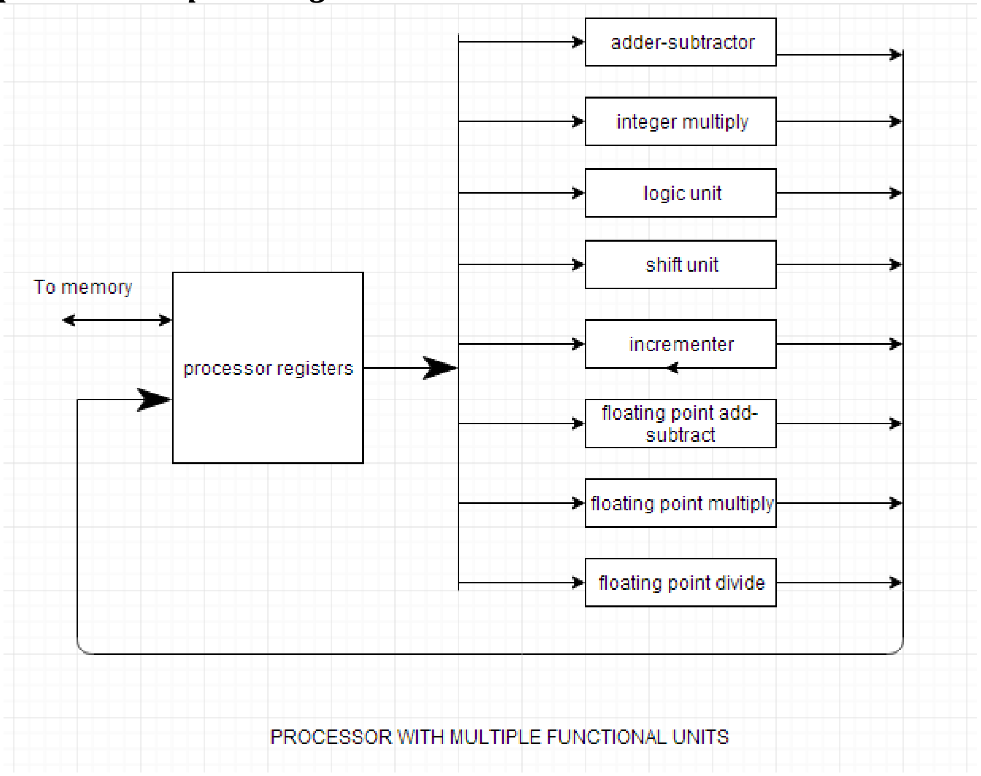
✓ A large number of instructions--typically from 100 to 250 instructions.
✓ Some instructions that perform specialized tasks and are used infrequently.
✓ A large variety of addressing modes—typically from 5 to 20 differ modes.
✓ Variable-length instruction formats
✓ Instructions that manipulate operands in memory

RISC Characteristics:

✓ Relatively few instructions
✓ Relatively few addressing modes
✓ Memory access limited to load and store instructions
✓ All operations done within the registers of the CPU
✓ Fixed-length, easily decoded instruction format
✓ Single-cycle instruction execution
✓ Hardwired rather than micro programmed control
✓ A relatively large number of registers in the processor unit
✓ Efficient instruction pipeline

Parallel Processing:

- Instead of processing each instruction sequentially, a parallel processing system provides concurrent data processing to increase the execution time.
- In this the system may have two or more ALU's and should be able to execute two or more instructions at the same time. The purpose of parallel processing is to speed up the computer processing capability and increase its throughput.
- ***Throughput*** is the number of instructions that can be executed in a unit of time.
- Parallel processing can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. At the higher level of complexity, parallel processing can be achieved by using multiple functional units that perform many operations simultaneously.



Data Transfer Modes of a Computer System

- According to the data transfer mode, computer can be divided into 4 major groups:

  1. SISD
  2. SIMD
  3. MISD
  4. MIMD

SISD (Single Instruction Stream, Single Data Stream)

- It represents the organization of a single computer containing a control unit, processor unit and a memory unit.
- Instructions are executed sequentially. It can be achieved by pipelining or multiple functional units.

SIMD (Single Instruction Stream, Multiple Data Stream)

- It represents an organization that includes multiple processing units under the control of a common control unit.
- All processors receive the same instruction from control unit but operate on different parts of the data.
- They are highly specialized computers. They are basically used for numerical problems that are expressed in the form of vector or matrix. But they are not suitable for other types of computations

MISD (Multiple Instruction Stream, Single Data Stream)

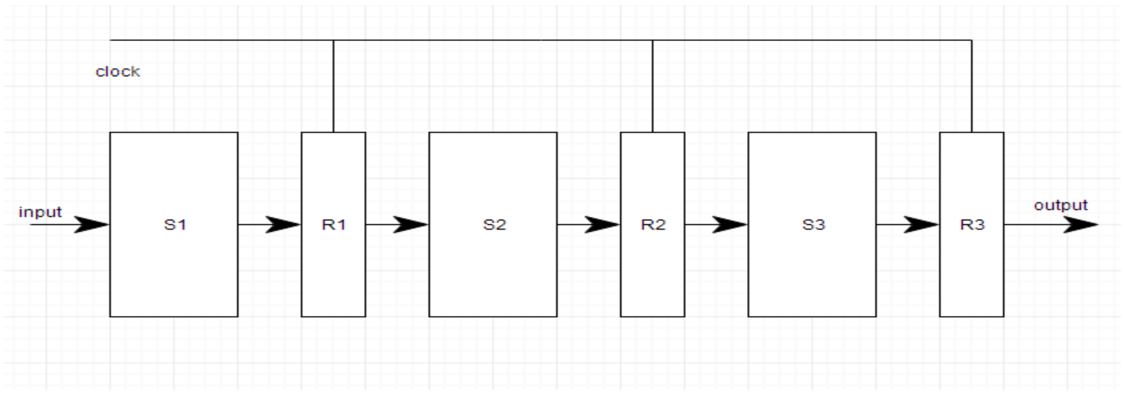
- It consists of a single computer containing multiple processors connected with multiple control units and a common memory unit.
- It is capable of processing several instructions over single data stream simultaneously.
- MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

MIMD (Multiple Instruction Stream, Multiple Data Stream)

- It represents the organization which is capable of processing several programs at same time.
- It is the organization of a single computer containing multiple processors connected with multiple control units and a shared memory unit.
- The shared memory unit contains multiple modules to communicate with all processors simultaneously.
- Multiprocessors and multicomputer are the examples of MIMD.
- It fulfills the demand of large scale computations.

Pipelining:

- Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as **pipeline processing**.
- Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end.
- Pipelining increases the overall instruction throughput.
- In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



- Pipeline system is like the modern day assembly line setup in factories. For example in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm.

Types of Pipeline

It is divided into 2 categories:

1. Arithmetic Pipeline
2. Instruction Pipeline

Arithmetic Pipeline

- Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed point numbers etc. For example: The input to the Floating Point Adder pipeline is:
  X=A*2^a
  Y=B*2^b

- Here A and B are mantissas (significant digit of floating point numbers), while **a** and **b** are exponents.
- The floating point addition and subtraction is done in 4 parts:
    - Compare the exponents.
    - Align the mantissas.
    - Add or subtract mantissas
    - Produce the result.
    - Registers are used for storing the intermediate results between the above operations.

Instruction Pipeline

- In this a stream of instructions can be executed by overlapping *fetch*, *decode* and *execute* phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system.
- An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration.

General Considerations:

- There are some factors that cause the pipeline to deviate its normal performance. Some of these factors are given below:

1. Timing Variations

- All stages cannot take same amount of time. This problem generally occurs in instruction processing where different instructions have different operand requirements and thus different processing time.

2. Data Hazards

- When several instructions are in partial execution, and if they reference same data then the problem arises. We must ensure that next instruction does not attempt to access data before the current instruction, because this will lead to incorrect results.

3. Branching

- In order to fetch and execute the next instruction, we must know what that instruction is. If the present instruction is a conditional branch, and its result will lead us to the next instruction, then the next instruction may not be known until the current one is processed.

4. Interrupts

- Interrupts set unwanted instruction into the instruction stream. Interrupts effect the execution of instruction.

5. Data Dependency

- It arises when an instruction depends upon the result of a previous instruction but this result is not yet available.

Advantages of Pipelining

1. The cycle time of the processor is reduced.
2. It increases the throughput of the system
3. It makes the system reliable.

Disadvantages of Pipelining

1. The design of pipelined processor is complex and costly to manufacture.
2. The instruction latency is more.

Input Output Organization:

Input Output Subsystem

- The I/O subsystem of a computer provides an efficient mode of communication between the central system and the outside environment.
- It handles all the input-output operations of the computer system.

Peripheral Devices

- Input or output devices that are connected to computer are called **peripheral devices**. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be the part of computer system. These devices are also called **peripherals**.
- For example: *Keyboards*, *display units* and *printers* are common peripheral devices.
- There are three types of peripherals:

  1. **Input peripherals**: Allows user input, from the outside world to the computer. Example: Keyboard, Mouse etc.
  2. **Output peripherals**: Allows information output, from the computer to the outside world. Example: Printer, Monitor etc
  3. **Input-Output peripherals**: Allows both input(from outised world to computer) as well as, output(from computer to the outside world). Example: Touch screen etc.

Interfaces

- Interface is a shared boundary between two separate components of the computer system which can be used to attach two or more components to the system for communication purposes.
- There are two types of interface:

  1. CPU Interface
  2. I/O Interface

Input-Output Interface

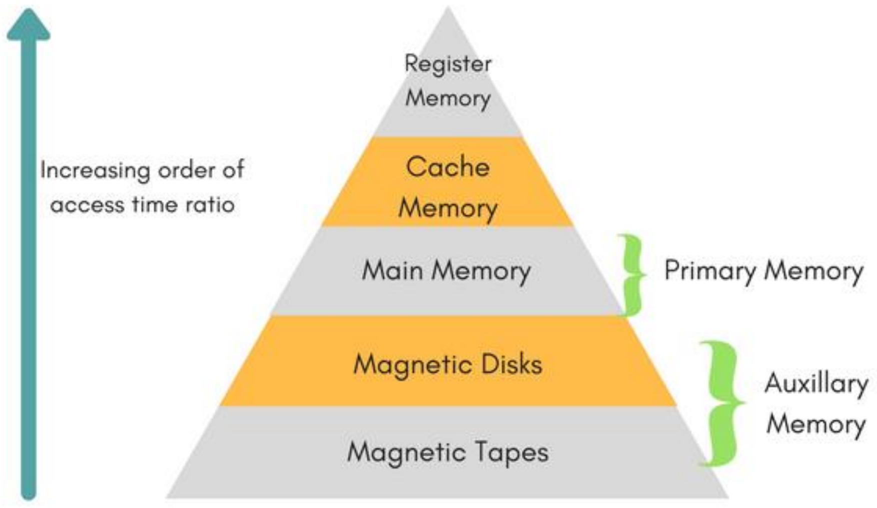
- Peripherals connected to a computer need special communication links for interfacing with CPU.
- In computer system, there are special hardware components between the CPU and peripherals to control or manage the input-output transfers. These components are called **input-output interface units** because they provide communication links between processor bus and peripherals.

- They provide a method for transferring information between internal system and input-output devices.

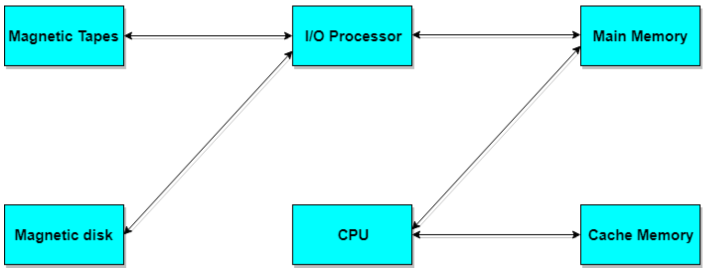
Memory Organization:

- A memory unit is the collection of storage units or devices together. The memory unit stores the binary information in the form of bits. Generally, memory/storage is classified into 2 categories:
    - **Volatile Memory**: This loses its data, when power is switched off.
    - **Non-Volatile Memory**: This is a permanent storage and does not lose any data when power is switched off.

Memory Hierarchy:



- The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.
- **Auxiliary memory** access time is generally **1000 times** that of the main memory, hence it is at the bottom of the hierarchy.
- The **main memory** occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).
- When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space in main memory for other programs that are currently in use.
- The **cache memory** is used to store program data which is currently being executed in the CPU. Approximate access time ratio between cache memory and main memory is about **1 to 7~10**



Memory Access Methods

- Each memory type, is a collection of numerous memory locations. To access data from any memory, first it must be located and then the data is read from the memory location. Following are the methods to access information from memory locations:

    1. **Random Access**: Main memories are random access memories, in which each memory location has a unique address. Using this unique address any memory location can be reached in the same amount of time in any order.
    2. **Sequential Access**: This method allows memory access in a sequence or in order.
    3. **Direct Access**: In this mode, information is stored in tracks, with each track having a separate read/write head.

Main Memory

- The memory unit that communicates directly within the CPU, Auxiliary memory and Cache memory, is called main memory. It is the central storage unit of the computer system. It is a large and fast memory used to store data during computer operations. Main memory is made up of **RAM** and **ROM**, with RAM integrated circuit chips holing the major share.

- RAM: Random Access Memory
    - **DRAM**: Dynamic RAM, is made of capacitors and transistors, and must be refreshed every 10~100 ms. It is slower and cheaper than SRAM.
    - **SRAM**: Static RAM, has a six transistor circuit in each cell and retains data, until powered off.
    - **NVRAM**: Non-Volatile RAM, retains its data, even when turned off. Example: Flash memory.
- ROM: Read Only Memory, is non-volatile and is more like a permanent storage for information. It also stores the **bootstrap loader** program, to load and start the operating system when computer is turned on. **PROM** (Programmable ROM), **EPROM**(Erasable PROM) and **EEPROM**(Electrically Erasable PROM) are some commonly used ROMs.

Auxiliary Memory

- Devices that provide backup storage are called auxiliary memory.
- **For example:** Magnetic disks and tapes are commonly used auxiliary devices.
- Other devices used as auxiliary memory are magnetic drums, magnetic bubble memory and optical disks.
- It is not directly accessible to the CPU, and is accessed using the Input/Output channels.

Cache Memory

- The data or contents of the main memory that are used again and again by CPU, are stored in the cache memory so that we can easily access that data in shorter time.
- Whenever the CPU needs to access memory, it first checks the cache memory. If the data is not found in cache memory then the CPU moves onto the main memory. It also transfers block of recent data into the cache and keeps on deleting the old data in cache to accomodate the new one.

Hit Ratio

- The performance of cache memory is measured in terms of a quantity called **hit ratio**. When the CPU refers to memory and finds the word in cache it is said to produce a **hit**. If the word is not found in cache, it is in main memory then it counts as a **miss**.
- The ratio of the number of hits to the total CPU references to memory is called hit ratio.
- Hit Ratio = Hit/(Hit + Miss)

Associative Memory

- It is also known as **content addressable memory (CAM)**.
- It is a memory chip in which each bit position can be compared.
- In this the content is compared in each bit cell which allows very fast table lookup.
- Since the entire chip can be compared, contents are randomly stored without considering addressing scheme.
- These chips have less storage capacity than regular memory chips.