UNIT-I

Introduction to Binary System and Codes;

Digital Systems:

- Digital systems can be refer to the present technology period as digital age.
- Digital systems are used in communication, business transactions, traffic control, internet and so on.
- We have digital telephones, digital televisions, digital versatile discs, digital cameras, & also digital computers
- These devices have graphical user interfaces (GUIs), which enable them to execute commands that appear to the user to be simple.
- It involves in precise execution of sequence of simple instructions
- The most striking property of digital computer is its generality.
- It can follow a sequence of instructions, called a program, that operates on a given data.
- one characteristic of digital system is their ability to represent and manipulate discrete elements of information.
- Any set that is restricted to a finite number of elements contain discrete information.

  Eg: 10 decimal digits.
  26 letters of alphabet.
  52 playing cards.
  64 squares of a chessboard.

- In this case, the discrete elements were the digits.
- From, this the term digital computer emerged.

- A digital system is simply one that receives input, processes (or) controls activity, and outputs information in a discrete (or) noncontinuous manner.
- The information may be encoded (or) represented in a variety of ways. it can be in the familiar base 10 number system or in some other number system, such as base 2 (binary)
- Digital design is the application of set of rules and techniques for developing digital circuits and subsystems to create a solution for some problem.

- Discrete elements of information are represented in digital system by physical quantities called signals.
- Electrical signals such as voltage and current are most common.
- Electronic (circuits) devices called transistors predominate in the circuitry that implement these signals.
- The signals in electronic digital systems use just two discrete values and are therefore said to be binary.
- A binary digit called a bit, has two values : 0 and 1
- Discrete elements of information are represented with group of bits called binary codes.

  Eg: Decimal digits 0 to 9 are represented with four bits.
- The general-purpose digital computer is the best-known example of digital system.
- The major parts of a computer are a memory unit, a central processing unit, and input-output units.
- A digital computer is a powerful instrument that can perform not only arithmetic computations, but also logical operations.
- In addition, it can be programmed to make decisions based on internal and external commands.
- A digital system is an interconnection of digital modules.
- To understand the operation of each digital module, it is necessary to have a basic knowledge of digital circuits and their logical function.
- A major trend in digital design methodology is the use of HDL

- An HDL resembles a programming language and is suitable for describing digital circuits in textual form.
- It is used to simulate a digital system to verify its operation before hardware is built.

Binary Numbers:

- A decimal number such as 7,392 represents a quantity equals to

$$\begin{array}{r} 7000 + \\ 300 + \\ 90 + \\ 2 + \\ \hline \end{array} \quad 7 \text{ thousands} + 3 \text{ hundred} + 9 \text{ tens} + 2 \text{ units.}$$

- The thousands, hundreds, etc. are powers of 10 implied by position of the coefficients (symbols) in the number.

It can be expressed as

$$7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

- In general, a number with a decimal point is represented by a series of coefficients.

$$a_5 a_4 a_3 a_2 a_1 a_0 . a_{-1} a_{-2} a_{-3}$$

- The coefficients a_j are any of the 10 digits (0, 1, 2, 3, ..., 9), and the subscript value j gives the place value and, hence, the power of 10 by which the coefficient must be multiplied.
- Thus, a decimal number can be expressed as

$$10^5 a_5 + 10^4 a_4 + 10^3 a_3 + 10^2 a_2 + 10^1 a_1 + 10^0 a_0 + 10^{-1} a_{-1} + 10^{-2} a_{-2} + 10^{-3} a_{-3}$$

with $a_7 = 7$, $a_2 = 30$, $a_1 = 9$, $a_0 = 2$

- A decimal number system is said to be of base or radix 10, because it uses 10 digits and coefficients are multiplied by powers of 10.

M.JOSHNA

- The binary number system is a different number system.
- The coefficients of the binary number system have only two possible values: 0 and 1
- Each coefficient a_j is multiplied by a power of the radix, eg. 2^i and the results are added to obtain the decimal equivalent of the number.

Eg: Binary number 11010.11 is equivalent to decimal number 26.75

$$1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 + 2^{-1}\times1 + 2^{-2}\times1 = 26.75$$

$$16 + 8 + 2 + 0.5 + 0.25$$

- There are different number systems. In general, a number expressed in base-r system has coefficients multiplied by powers of r.

$$a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + a_{n-2} \cdot r^{n-2} + \cdots + a_1 r^1 + a_0 r^0 + a_{-1} r^{-1} + \cdots + a_{-m} \cdot r^{-m}$$

- The coefficients a_j range in values from 0 to $r-1$.

Eg: base-5 number

$$(4021.2)_5 = 4\times5^3 + 0\times5^2 + 2\times5^1 + 1\times5^0 + 2\times5^{-1}$$

$$\frac{1}{5} = .2$$

$$\frac{125\times4}{500} = 2$$

$$= 500 + 10 + 1 + 0.4$$

$$= 511.4$$

- The coefficient values for base 5 can be only 0, 1, 2, 3, 4
- The octal number system is a base-8 system that has 8 digits : 0, 1, 2, 3, 4, 5, 6, 7.

Eg: $(127.4)_8 = 1\times8^2 + 2\times8^1 + 7\times8^0 + 4\times8^{-1}$

$= 64 + 16 + 7 + 0.5$

$= (87.5)_{10}$

Note: The digits 8 and 9 can't appear in a octal number.

- It is customary to borrow the needed r digits for the coefficients from the decimal system when the base of the number is less than 10.
- The letters of the alphabet are used to supplement the 10 decimal digits when the base of the number is greater than 10.

For eg:

In the hexadecimal (base-16) number system, the first 10 digits are borrowed from the decimal system. The letters A, B, C, D, E and F are used for the digits 10, 11, 12, 13, 14 and 15

Eg: A Hexadecimal number

$(B\ 6\ 5\ F)_{16} = 11\times16^3 + 6\times16^2 + 5\times16^1 + 15\times16^0$

$= (46,687)_{10}$

- The hexadecimal system is used to represent long strings of bits in the addresses, instructions, and data in digital systems.

Eg: B 6 5 F is used to represent

1011 0110 0101 1111

- The conversion from binary to decimal can be obtained by adding only the numbers with powers of two corresponding to the bits that are equal to 1.

For Eg: $(110101)_2 = 32 + 16 + 4 + 1 = (53)_{10}$

- In computer work

2^{10} = K (kilo) = 10^3 2^{20} = M (mega) = 10^6 2^{30} = G (giga) = 10^9

2^{40} = T (Tera) = 10^{12} 2^{50} = P (peta) = 10^{15} 2^{60} = X = 10^{18}

- Computer capacity is usually given in bytes.
- A byte is equal to eight bits and can represents the code of one keyboard character.
- A computer hard disk with 4 GiB of storage has a capacity of

$4G = 2^2 \cdot 2^{30} = 2^{32}$ = (approximately 4 billion bytes)

- Arithmetic operations with binary numbers in base r follow the same rules as for decimal numbers.

Eg: addition, subtraction and multipliplication of two binary numbers as follows.

Augend	101101	minuend:	101101	multiplicand:	1011
Addend	+ 100111	subtrahend:	- 100111	Multiplier:	101
	1010100		000110		1011

Addition:

- The sum of two binary numbers is calculated by the same rules as in decimal except that the digits of sum in any significant position can be only 0 or 1.
- Any carry obtained in a given significant position is used by the pair of digits one significant position higher.

Subtraction:

- Subtraction is more complicated.
- The rules are still the same as in decimal, except that the borrow in a given significant position adds 2 to a minuend digit.
- A borrow in the decimal system adds 10 to a minuend digit.

Multiplication:

- Multiplication is simple.
- The multiplier digits are always 1 or 0.
- Therefore, the partial products are equal either to a shifted (left) copy of multiplicand or to 0.

Number - Base Conversions

- Representation of numbers in a different radix are said to be equivalent if they have same decimal representation.

For eg: $(0011)_8$ and $(1001)_2$ are equivalent, both have same decimal value. 9.

- The conversion of a number in base-r to decimal is done by expanding the number in a power series and adding all the numbers

and accumulating the remainders.

Eg: Convert decimal 41 to binary

$$
\begin{array}{r|l}
2 & 41 \\
2 & 20 - 1 \\
2 & 10 - 0 \\
2 & 5 - 0 \\
2 & 2 - 1 \\
 & 1 - 0
\end{array}
$$

$\therefore (41)_{10} = (101001)_2$

Eg: Convert decimal $(153)_{10}$ to octal $(\ \ )_8$

$$
\begin{array}{r|l}
8 & 153 \\
8 & 19 - 1 \\
 & 2 - 3
\end{array}
$$

$\therefore (153)_{10} = (231)_8$

Eg: Convert $(41)_{10}$ to octal

$$
\begin{array}{r|l}
8 & 41 \\
 & 5 - 1
\end{array}
$$

$\therefore (41)_{10} = (51)_8$

Eg: Convert $(0.6875)_{10}$ to binary

$0.6875 \times 2 = 1.3750 = 1$

$0.3750 \times 2 = 0.750 = 0$

$0.750 \times 2 = 1.50 = 1$

Eg: $(0.513)_{10}$ to octal

$(0.513) \times 8 = 4.104$

$0.104 \times 8 = 0.832$

$0.832 \times 8 = 6.656$

$0.656 \times 8 = 5.248$

$0.248 \times 8 = 1.984$

$(0.513)_{10} = (0.40651\ldots)_8$

$\therefore (41.6875)_{10} = (101001.1011)_2$

$\therefore (153.513)_{10} = (231.406517)_8$

Octal and Hexadecimal Numbers

- the conversions from and to binary, octal and hexadecimal plays an important role in digital computer, because shorter patterns of 0's & 1's hex characters are easier to recognize than long patterns of 0's & 1's
- since $2^3 = 8$ & $2^4 = 16$, each octal digit corresponds to three binary digits and each hexadecimal digit corresponds to four binary digits

⇒ The first 16 numbers in the decimal, binary, octal & hexadecimal as follows.

Decimal (base 10)	Binary (base 2)	octal (base 8)	Hexadecimal (base 16)
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
			9

11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

- The conversion from binary to octal is accomplished by partitioning the binary number into group of three digits each, starting from the binary point and proceeding to the left and to the right.
- The corresponding octal digit is assigned to each group.

Eg: $(10\ 110\ 001\ 101\ 011.\ 111\ 100\ 000\ 110)_2$

$\Rightarrow (2\ 6\ 1\ 5\ 3.\ 7\ 4\ 0\ 6)_8$

- Conversion from binary to hexadecimal is similar, except that the binary number is divided into groups of four digits.

$(10\ 1100\ 0110\ 1011.\ 1111\ 0000\ 0110)_2$

$\Rightarrow (2\ C\ 6\ B.\ F\ 0\ 6)_{16}$

- Conversion from octal (or) hexadecimal to binary is done by representing the each octal digit to its three-digit binary equivalent.

- similarly, each hexadecimal digit is converted to its four-digit binary equivalent.

Eg: $(673.124)_8 = (110\ 111\ 011.\ 001\ 010\ 100)_2$

$(306.D)_{16} = (0011\ 0000\ 0110.\ 1101)_2$

Compliments of Numbers:

- Compliments are used in digital computers to simplify the subtraction operation and for logic operation.
- Simplifying operations leads to simpler, less expensive circuits to implement the operations.

There are two types of complements for each base-r system.

1. Radix Complement (r's complement)
2. Diminished Radix Complement. ((r-1)'s complement)

- The two types referred to as the 2's complement and 1's complement for binary numbers & 10's complement and 9's complement for decimal numbers.

Diminished Radix Complement:

Given a number N in base r having n digits

The $(r-1)$'s complement of N is $(r^n - 1) - N$

for decimal number: $r = 10$, $r - 1 = 9$

So, (Diminished Radix Complement) 9's complement of N is

$$(10^n - 1) - N$$

Eg: if $n = 4$

$$(10^4 - 1) - N \Rightarrow 9999 - N$$

∴ 9's complement of a decimal number is obtained by subtracting

The 9's complement of 547600 is

$$999999 - 547600 \Rightarrow 452399$$

The 9's complement of 012398 is

$$999999 - 012398 \Rightarrow 987601$$

For binary numbers $r = 2$ $r-1 = 1$

1's complement of N is $(2^n - 1) - N$

if $n = 4$ $\quad 2^n = 2^4 = (10000)_2$ & $2^4 - 1 = (1111)_2$

$\therefore$ 1's complement of N is $(2^n - 1) - N$

$= (1111) - N$

Thus, the 1's complement of binary number is obtained by subtracting each digit from 1.

we can have either $1 - 0 = 1$

(or)

$1 - 1 = 0$

1's complement of 1011000 is 0100111

1's complement of 0101101 is 1010010

$\therefore$ The 1's complement of a binary number is formed by changing 1's to 0's and 0's to 1's.

- $(r-1)$'s complement of octal or hexadecimal number is obtained by subtracting each digit from 7 (or) F (decimal 15), respectively

Radix Complement

The r's complement of an n-digit number N in base r is defined as $r^n - N$ for $N \neq 0$ & as 0 for $N = 0$.

For decimal $r = 10$

$$r^n - N \Rightarrow 10^n - N.$$

∴ 10's complement of N can be formed by leaving all least significant 0's unchanged, subtracting the first non zero least significant digit from 10, and subtracting all higher digits from 9. Thus

10's complement of 012398 is 987602

10's complement of 246700 is 753300

- Similarly 2's complement of N can be formed by leaving all least significant 0's and the first 1 unchanged and replacing 1's with 0's and 0's with 1's in all other higher significant digits.

```
  10 10 10 10 10 10 10
 1 0  0  0  0  0  0  0
   1  1  0  1  1  0  0
 ---------------------
   0  0  1  0  1  0  0
```

2's complement of 1101100 is 0010100

2's complement of 0110111 is 1001001

- If the original number N contains the radix point, the point should be removed temporarily in order to form the r's & (r-1)'s complement.
- The radix point is then restored to the complemented number.
- The complement of the complement restores the number to its original value.

Subtraction with complements

The subtraction of two n-digit unsigned numbers M-N in base r can be done as follows.

1. Add the minuend M to the r's complement of the subtrahend N. Mathematically $M + (r^n - N) = M - N + r^n$
2. If $M \geq N$, the sum will produce an end carry r^n, which can be discarded; what left is the result M-N.
3. If $M < N$, the sum does not produce an end carry and is equal to $r^n - (N - M)$, which is r's complement of (N-M). To obtain the answer in a familiar form, take the r's complement of sum & place a negative sign in front.

Eg: using 10's complement, subtract 72532 − 3250

M = 72532

10's complement of N = 96750

Sum = 169282

Discard end carry 10^5 = 100000

69285

Eg: using 10's complement, substract 3250 − 72532

M = 03250

10's complement of N = 27468

30718

There is no end carry

Eg: Given the two binary numbers $x = 1010100$ & $y = 1000011$, perform subtraction a) $x - y$ & b) $y - x$ by using 2's complement

a) $x = 1\ 0\ 1\ 0\ 1\ 0\ 0$

2's complement of $y = 0\ 1\ 1\ 1\ 1\ 0\ 1$

$1\ 0\ 0\ 1\ 0\ 0\ 0\ 1$

Discard end carry $2^7 = 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0$

$0\ 0\ 1\ 0\ 0\ 0\ 1$

b) $y = 1\ 0\ 0\ 0\ 0\ 1\ 1$

2's complement of $x = 0\ 1\ 0\ 1\ 1\ 0\ 0$

$1\ 1\ 0\ 1\ 1\ 1\ 1$

There is no end carry

∴ The answer is $y - x = -$(2's complement of 1101111)

$= -0010001$

Examples to work:

1) Convert the following to decimal and then binary

a) $(1876)_{16}$ b) $(AB22)_{16}$ c) $(1212)_8$ d) $(1556)_8$

2) Convert the following hexa decimal numbers into an equivalent binary numbers

(i) 58 (ii) 234 (iii) ABC (iv) FB18

3) Subtract the following decimal numbers using 9's complent

(i) 347 − 265 (ii) 49 − 84 (iii) 349 − 436 (iv) 9 − 4

4) Convert the following to octal and Hexadecimal

a) 1001011010111_2 b) $1011010000010 1_2$

c) 791_{10} d) 1600_{10} e) 197_{10}

Eg: Using i's complement

a) $x - y = 1010100 - 1000011$

$x = 1010100$

i's complement of $y = 0111100$

Sum $= 10010000$

End-around carry: + 1

0010001

b) $y - x = 1000011 - 1010100$

$y = 1000011$

i's complement of $x = 0101011$

Sum $= 1101110$

There is no end-carry. $\therefore y - x = -$ (i's complent of 1101110)

$= -0010001$

- In (r-1)'s complement removing the end around carry and adding 1 to the sum is referred to as an end-around carry
- The procedure with end-around carry is also applicable to

Signed Binary Numbers:

- Positive integers (including zero) can be represented as unsigned no's.
- In ordinary arithmetic, a negative no is indicated by minus sign and a positive number by a plus sign.
- Because of hardware limitations, computers must represent everything with binary digits.
- It is customary to represent (everything) the sign with a bit placed in the leftmost position of the number.
- The convention is to make the sign bit 0 for positive & 1 for negative.
- Both signed and unsigned binary numbers consists of a string of bits when represented in a computer.
- The user determines whether the number is signed or not unsigned.
- If the binary number is signed, then the left most bit represents the sign and the rest of the bits represent number.
- If the binary number is unsigned, then the left most bit is the most significant bit of the number.

Eg: the string of bits 01001 is considered as

9 unsigned number

+9 signed number.

the string of bits 11001 is considered as

25 unsigned number

-9 signed number.

- In this notation, the number consists of a magnitude and a symbol (+ or −) or a bit (0 or 1) indicating the sign.

Signed Complement System

- when arithmetic operations are implemented in a computer, it is more convenient to use a different system, referred to as the signed complement system, for representing negative numbers
- In this system a negative number is indicated by its complement
- whereas the signed magnitude system negates a number by changing its sign, the signed complement system negates a number by taking its complement.

Eg: Consider the number 9, represented in binary with eight bits

Binary equivalent of +9 is 00001001

there are 3 ways to represent −9

- signed magnitude representation 10001001
- signed is complement representation 11110110
- signed 2's complement representation 11110111

Representation of Signed numbers using 2's (or) 1's complement method

1. If the number is positive, the magnitude is represented in its true binary form and a sign bit 0 is placed in front of the most MSB

Table:

Signed Binary Numbers

Decimal	signed-2's complement	signed-1's complement	signed magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	—	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1110	1110
-7	1001	1000	1111
-8	1000	—	—

Special case in 2's complement: whenever a signed number has a 1 in the sign bit and all of the magnitude bits, the decimal equivalent is -2^n. where n is the number of bits in magnitude. For Eg: 1000 = -8, 10000 = -16

Arithmetic Addition

The addition of two signed binary numbers with negative numbers represented in signed-2's-complement form is obtained from the addition of the two numbers, including their sign bits. A carry out of the sign-bit position is discarded.

Eg:

```
+6   00000110        -6   11111010
+13  00001101        +13  00001101
---  --------        ---  --------
+19  00010011        +7  [1]00000111   carry has to be discarded.

+6   00000110        -6   11111010
-13  11110011        -13  11110011
---  --------        ---  --------
-7   [1]11111001     -19 [1]11101101   carry has to be
```

Arithmetic Subtraction

- Subtraction of two signed binary numbers when negative numbers are in 2's complement form is simple and can be stated as follows.

Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign-bit position is discarded.

- This procedure is adopted because a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed, as is demonstrated by the following relationship.

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

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

Eg: $(-6) - (-13) = +7$

```
  1 1 1 1 1 0 1 0   —   1 1 1 1 0 0 1 1
  1 1 1 1 1 0 1 0
  0 0 0 0 1 1 0 1
 ---------------
1]0 0 0 0 0 1 1 1      carry has to be discarded.
```

$+6 - (+13)$

```
  0 0 0 0 0 1 1 0
  1 1 1 1 0 0 1 1
 ---------------
  1 1 1 1 1 0 0 1
```

Binary Codes

- A n-bit binary code is a group of n-bits that can have upto 2^n distinct combinations of 0's and 1's.
- Each combination represents one element of the set that is being coded.
- A set of 4 elements can be coded with two bits, with each element assigned one of the bit combinations: 00, 01, 10, 11.
- A set of 8 elements requires a three bit code and a set of 16 elements requires a four bit code.
- The bit combination of an n-bit code is determined from the count in binary from 0 to $2^n - 1$.
- Each element must be assigned a unique binary bit combination, and no two elements can have the same value; otherwise the code assignment will be ambiguous.
- The minimum number of bits required to code 2^n distinct quantities is n, there is no maximum number of bits that ~~required to code~~ may be used for a binary code.

Binary-Coded Decimal Code:

- A BCD code is one, in which the digits of a decimal number are encoded one at a time into groups of four binary digits.
- These codes combine the features of decimal and binary numbers.
- In order to represent decimal digits 0, 1, 2, ..., 9, it is

- Such a sequence of binary digits which represents a decimal digit is called a code word.
- A binary code will have some unassigned bit combinations if the number of elements in the set is not a multiple power of 2.
- The 10 decimal digits from such a set.
- A binary code that distinguishes among 10 elements must contain at least four bits, but 6 out of 16 possible combinations remain un-~~assigned~~ Assigned.
- This scheme is called Binary Coded Decimal and is commonly referred to as BCD.

The following table gives the four-bit code for one decimal digit

- A number with K ~~bits~~ decimal digits will require 4K bits in BCD.

Binary Coded Decimal (BCD)

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

- the binary combinations 1010 through 1111 are not used and have no meaning in BCD.

Eg: Consider decimal $(185)_{10}$ & its corresponding value in BCD & binary

$$(185)_{10} = (0001\ 1000\ 0101)_{BCD} = (10111001)_2$$

BCD Addition:

- Consider the addition of two decimal digits in BCD, together with a possible carry from a previous less significant pair of digits.
- since each digit does not exceed 9, sum cannot be greater than

$$9+9+1 = 19. \quad \text{with 1 being a previous carry.}$$

- Suppose we add the BCD digits as if they were binary numbers.
- Then the binary sum will produce a result in the range from 0 to 19.
- In binary, this range will be from 0000 to 10011. But in BCD, it is from 0000 to 11001, with the first 1 being a carry and the next four bits being the BCD sum.
- when the binary sum is equal to or less than 1001 (without carry), the corresponding BCD digit is correct.
- when the binary sum is greater than or equal to 1010, the result is an invalid BCD digit.
- the addition of $6 = (0110)_2$ to the binary sum converts it to the correct digit and also produces a carry as required.

- This is because a carry in the most significant bit position and

M.JOSHNA

Consider the following BCD additions.

```
  4    0100        4    0100        8    1000
+ 5    0101       +8    1000        9    1001
---   -----       ---  -----       --  -----
  9    1001      [12]   1100       17  10001
                 (+6) + 0110           + 0110
                      ------           ------
                    1 0010             1 0111
                    1    2             1    7
```

The addition of two n-digit unsigned BCD numbers follows the same procedure.

consider $184 + 576 = 760$ in BCD

```
BCD     0001    1000     0100      184
        0101    0111     0110    + 576
       ------------------------
      1 0111   10000     1010
                0110     0110
               -----    -----     ----
                0110     0000      760
```

BCD Subtraction

- The BCD substraction is performed by substracting the digits of each 4-bit group of the substrahend from the corresponding 4-bit group of the minuend in binary starting from the LSD.
- If there is no borrow from the next higher group then no correction is required.

(This is done to skip 6 illegal states)

Eg: a) 38 − 15 = 23

0011	1000	38 in BCD
0001	0101	15 in BCD
0010	0011	23 in BCD
2	3	No borrow. So, this is the correct difference.

b) 206 − 147 = 59

0010	0000	0110
0001	0100	0111
0	1011	1111
	0110	− 0110
0101		1001
5		9

Borrows are present subtract 0110

Other Decimal codes

- Binary codes for decimal digits require a minimum of four bits per digit.
- Many four different codes can be formulated by arranging four bits into 10 distinct combinations.
- Each code uses only 10 out of a possible 16 bit combinations that can be arranged with four bits
- the other six unused combinations have no meaning and should be avoided.
- BCD and the 2421 code are examples of weighted codes.
- In a weighted code, each bit position is assigned a weighting factor in such a way that each digit can be evaluated by adding the weights of all the 1's in the coded combination.

- The bit assignment 0110, for eg: is interpreted by weights to represent decimal 6. because $8\times0+4\times1+2\times1+0\times1=6$.
- Note that some digits can be coded in two possible ways in 2421 code.
- for instance, decimal 4 can be coded to bit combination 0100 or 1010, since both combinations add upto a total weight of 4.

Table: four different Binary Codes for decimal digits.

Decimal Digit	BCD 8421	2421	Excess-3	8,4,-2,-1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1011
6	0110	1100	1001	1010
7	0111	1101	1010	1001
8	1000	1110	1011	1000
9	1001	1111	1100	1111
unused codes	1010	0101	0000	0001
	1011	0110	0001	0010
	1100	0111	0010	0011
	1101	1000	1101	1100
	1110	1001	1110	1101
	1111	1010	1111	1110

- 2421 and the excess-3 are examples of self-complementing codes.
- Such codes have the property that the 9's complement of a decimal number is obtained directly by changing 1's to 0's and 0's to 1's

Eg: Decimal 395 excess-3 2421

0110 1100 1000 0011 1111 1011

9's compt of 395 is 604 1001 0011 0111 1100 0000 0100

6 0 4 6 0 4 ✓

- The excess-3 code has been used in some older computers because of its self-complementing property.
- Excess-3 is an unweighted code in which each coded combination is obtained from the corresponding binary value plus 3.
- The 8,4,-2,-1 code is an example of assigning both positive and negative weights to a decimal code.

Gray Code:

- The output data of many physical systems are quantities that are continuous.
- These data must be converted into digital form before they are applied to a digital systems.
- Continuous (or) analog information is converted into digital form by means of analog-to-digital converter.
- The advantage of Gray Code over the straight binary numbers sequence is that only one bit in the code group changes in going from one number to the next number.

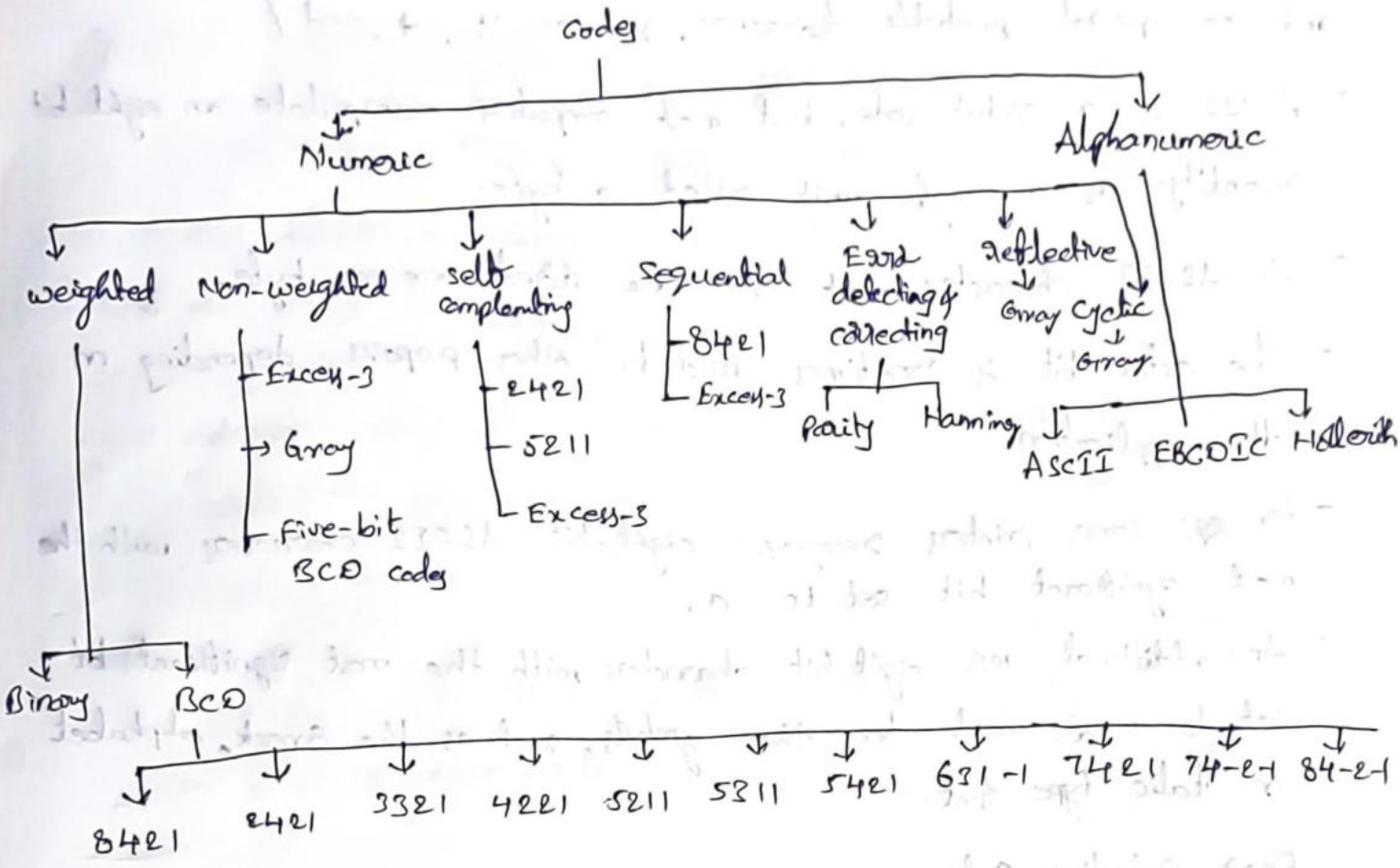
Eg: in going from 7 to 8, the gray code changes from 0100 to 1100. only first bit changes from 0 to 1; other three bits remain same.

By contrast, with binary numbers the change from 7 to 8 will

- the Gray code is a non-weighted code, and is not suitable for arithmetic operations.
- An N-bit Gray code can be obtained by reflecting an N-1 bit code about an axis at the end of the code, and putting the MSB of 0 above the axis and the MSB of 1 below the axis.

Reflection of Gray-Codes is shown in following table.

Gray Code				Decimal	4-bit binary
1-bit	2-bit	3-bit	4-bit		
0	0 0	0 0 0	0 0 0 0	0	0 0 0 0
1	0 1	0 0 1	0 0 0 1	1	0 0 0 1
	1 1	0 1 1	0 0 1 1	2	0 0 1 0
	1 0	0 1 0	0 1 0 0	3	0 0 1 1
		1 1 0	0 1 0 1	4	0 1 0 0
		1 1 1	0 1 1 0	5	0 1 0 1
		1 0 1	0 1 1 1	6	0 1 1 0
		1 0 0	1 0 0 0	7	0 1 1 1
			1 0 0 1	8	1 0 0 0
			1 0 1 0	9	1 0 0 1
			1 0 1 1	10	1 0 1 0
			1 1 0 0	11	1 0 1 1
			1 1 0 1	12	1 1 0 0
			1 1 1 0	13	1 1 0 1
			1 1 1 1	14	1 1 1 0
				15	1 1 1 1



ASCII character code

- Many applications of digital computers requires the handling of not only of numbers, but also of other characters or symbols, such as letters of alphabets.
- An alphanumeric character set is a set of elements that includes the 10 decimal digits, 26 letters of the alphabet, and a number of special characters.
- The standard binary code for the alphanumeric characters is the American standard code for Information Interchange (ASCII), which uses seven bits to code 128 characters.
- The seven bits of the code are designated by b_1 through b_7 with b_7 the most significant bit.
- The ascii code contains 94 graphic characters that can be printed & 34 non printing characters used for various control functions.

and 32 special printable characters, such as %, *, and $

- ASCII is a 7-bit code, but most computers manipulate an eight-bit quantity as a single unit called a byte.
- ∴ ASCII characters most often are stored one per byte.
- The extra bit is sometimes used for other purposes, depending on the application.
- for eg: Some printers recognize eight-bit ASCII characters with the most significant bit set to 0.
- An additional 128 eight-bit characters with the most significant bit set to 1 are used for other symbols, such as the Greek alphabet or italic type font.

Error-Detecting Code

- To detect errors in data communication and processing an eighth bit is sometimes added to the ASCII character to indicate its parity.
- A parity bit is an extra bit included with a message to make the total number of 1's either even (or) odd.

Consider the following two characters and their even and odd parity.

	with even parity	with odd parity
ASCII = 1000001	01000001	11000001
ASCII = 1010100	11010100	01010100

- The parity bit is helpful in detecting errors during the transmission of information from one location to another.
- This function is handled by generating an even parity bit at the sending end for each character.
- The eight bit characters that include parity bits are transmitted

- the parity of each character is checked at the receiving end.
- If the parity of the received character is not even, then at least one bit has changed value during the transmission.
- This method detects one, three, or any odd combination of errors in each character that is transmitted.
- An even combination of errors, however, goes undetected, and additional error detection codes may be needed to take care of that possibility.
- After detecting an error one possibility is to request retransmission of the message on the assumption that the error was random and will not occur again.
- Thus, if the receiver detects a parity error, it sends back the ASCII NAK control character consisting of an even-parity eight bits 10010101.
- If no error is detected, the receiver sends back an ACK control character, namely, 00000110.
- If, after a number of attempts, the transmission is still in error, a message can be sent to the operator to check for malfunctions

Representation

Due to limitations of hardware, computers must represent everything with 0's & 1's, including the sign of a number. As a consequence, it is customary to represent the sign bit placed in the left-most position of the number. The convention is to make the sign bit equal to 0 for positive numbers & to 1 for negative numbers.

In addition to the sign, a number may have a binary (or decimal) point. The position of binary point is needed to represent fractions, integers, (or) mixed integer-fraction numbers.

There are two ways of specifying the position of the binary point in a register.

1. Fixed Point Representation
2. Floating Point Representation.

- Fixed-Point Representation assumes that the binary point is always fixed in one position.

The two positions most widely used are

(i) a binary point in the extreme left of the register to make the stored number a fraction.

(ii) a binary point in the extreme right of the register to make the stored number an integer.

- The floating-point representation uses a second register to store a number that designates the position of decimal point in the first register.

M.JOSHNA

Floating-Point Representation

The floating point representation of a number has two parts. The first part represents a signed, fixed point number called the mantissa. The second part designates the position of binary (or decimal) point and is called exponent. The fixed point mantissa may be a fraction (or) an integer.

Eg: Consider the decimal number.

+ 6132.789

Floating point representation is

Fraction	Exponent
+0.6132789	+04

The value of the exponent indicates the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction.

Notation is : $+0.6132789 \times 10^{+4}$

Floating-Point is always interpreted to represent a number in the following form.

$$m \times r^{e}$$

m - mantissa

e - exponent (including their sign)

r - radix

Eg: Consider a binary number +1001.11 is represented with 8-bit fraction & 6-bit exponent as follows.

Fraction	Exponent
0 1001110	000100

∴ Floating point number is $m \times 2^e = +(.1001110) \times 2^{+4}$

Normalization

A Floating Point number is said to be normalized, if the most significant digit of the mantissa is non zero.

For eg: Decimal number 350 is normalized but 00035 is not.

Normalized numbers provide the maximum possible ~~prec~~ precision for the floating point number.

Standard forms of floating Point numbers

1. ANSI (American National Standards Institute)
2. IEEE (Institute of Electrical & Electronic Engineers)

The ANSI 32-bit floating point numbers in byte format. is given below.

Byte Format

Byte 1		Byte 2	Byte 3	Byte 4
SEEEE	.	MMMMMMMM	MMMMMMMM	MMMMMMMM
	↑ Binary Point		Mantissa	

S - Sign of Mantissa

E - Exponent bits in 2's complement

M - Mantissa bits.

Eg:

$13 = 1101 = 0.1101 \times 2^4$

$= 0\ 0000100\ \ 11010000\ \ 00000000\ \ 00000000$

$-17 = -10001 = -0.10001 \times 2^5$

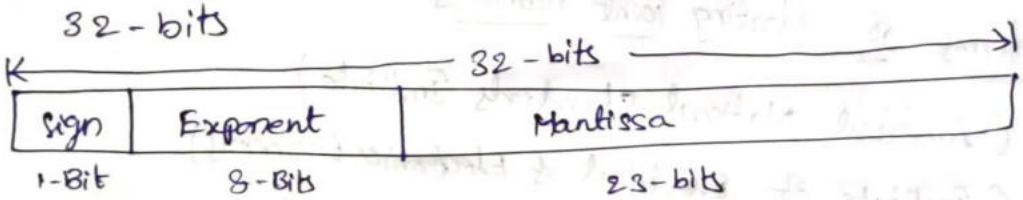
$= 10000101\ \ 10001000\ \ 00000000\ \ 00000000$

$-.125 = -0.001 = -.1 \times 2^{-2}$

$= 11111110\ \ 10000000\ \ 00000000\ \ 00000000$

IEEE format

1. Single Precision



Eg: 85.125

$(85)_{10} = (1010101)_2$

$(125)_{10} = (001)_2$

$(85.125)_{10} = 1010101.001$

$= 1.010101001 \times 2^6$

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

Biased Exponent : $127+6 = 133 = 10000101$

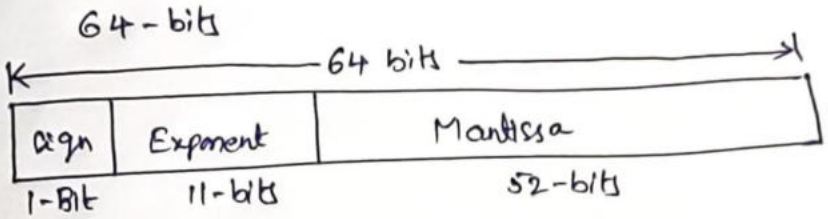
IEEE 754 Single Precision Representation is

0 10000101 01010100100000000000000

Hexa decimal is

45AA4000

2. Double Precision



Eg:

$(85.125) = 1.010101001 \times 2^6$

Biased Exponent : $1023 + 6 = 1029 = 10000000101$

Sign bit : 0

IEEE 754 Double precision is

0 10000000101 0101010010000...0 (49 bits - pad zero)

Hexa decimal value

4055480000000000

M.JOSHNA

M.JOSHNA

UNIT I-PART-II

Syllabus: Basic Structure Of Computers: Functional unit, Basic Operational concepts, Bus structures, System Software, Performance, The history of computer development.

Computer**:** **Computer is a fast electronic calculating machine that accepts digitized input information,** processing it according to a list of internally stored instructions and produces the resulting output information. The list of instructions is called as a Computer program and the internal storage is called as Computer memory.

Types of Languages: Just as humans use language to communicate, and different regions have different languages, computers also have their own languages that are specific to them. Different kinds of languages have been developed to perform different types of work on the computer. Basically, languages can be divided into two categories according to how the computer understands them.

1. **Low-Level Languages:** A language that corresponds directly to a specific machine. Low-level computer languages are either machine codes or are very close them. A computer cannot understand instructions given to it in high-level languages or in English. It can only understand and execute instructions given in the form of machine language i.e. binary. There are two types of low-level languages:

   1. **Machine Language:** a language that is directly interpreted into the hardware. Machine language is the lowest and most elementary level of programming language and was the first type of programming language to be developed. Machine language is basically the only language that a computer can understand and it is usually written in hex. It is represented inside the computer by a string of binary digits (bits) 0 and 1. The symbol 0 stands for the absence of an electric pulse and the 1 stands for the presence of an electric pulse. Since a computer is capable of recognizing electric signals, it understands machine language.

   Advantages:

   - Machine language makes fast and efficient use of the computer.
   - It requires no translator to translate the code. It is directly understood by the computer.

   Disadvantages:

   - All operation codes have to be remembered
   - All memory addresses have to be remembered.
   - It is hard to amend or find errors in a program written in the machine language.

   2. **Assembly Language:** A slightly more user-friendly language that directly corresponds to machine language. Assembly language was developed to overcome some of the many inconveniences of machine language. This is another low-level but very important language in which operation codes and operands are given in the form of alphanumeric symbols instead of 0's and l's.

   These alphanumeric symbols are known as mnemonic codes and can combine in a maximum of five-letter combinations e.g. ADD for addition, SUB for subtraction, START, LABEL etc. Because of this feature, assembly language is also known as 'Symbolic Programming Language.'

   Advantages:

   - Assembly language is easier to understand and use as compared to machine language.
   - It is easy to locate and correct errors.
   - It is easily modified.

Disadvantages:

- Like machine language, it is also machine dependent/specific.
- Since it is machine dependent, the programmer also needs to understand the hardware.

2. **High-Level Languages:** Any language that is independent of the machine. High-level computer languages use formats that are similar to English. The purpose of developing high-level languages was to enable people to write programs easily, in their own native language environment (English).

High-level languages are basically symbolic languages that use English words and/or mathematical symbols rather than mnemonic codes. Each instruction in the high-level language is translated into many machine language instructions that the computer can understand.

Advantages:

1.High-level languages are user-friendly
2.They are easier to learn.
3.They are easier to maintain
4. A program written in a high-level language can be translated into many machine languages and can run on any computer
5.programs developed in a high-level language can be run on any computer text

Disadvantages:

6. A high-level language has to be translated into the machine language by a translator, which takes up time

Computer Types: Basing capacity, technology used and performance of computer, they are classified into two types

→ According to computational ability
→ According to generation

According to computational ability (Based on Size, cost and performance):

There are mainly 4 types of computers. These include:

1.Micro computers
2.Mainframe computers
3.Mini computers
4.**Super computer**

1. **Micro computers**: -

Micro computers are the most common type of computers in existence today, whether at work in school or on the desk at home. These computers include:

1. **Desktop computer**
2. **Personal digital assistants (more commonly known as PDA's)**
3. **Palmtop computers**
4. **Laptop and notebook computers**

Micro computers were the smallest, least powerful and least expensive of the computers of the time. The first Micro computers could only perform one task at a time, while bigger computers ran multi-tasking operating systems, and served multiple users. Referred to as a personal computer or "desktop computer", Micro computers are generally meant to service one user (person) at a time. By the late 1990s, all personal computers run a multi-tasking operating system, but are still intended for a single user.

2. Mainframe Computers :-

The term Mainframe computer was created to distinguish the traditional, large, institutional computer intended to service multiple users from the smaller, single user machines. These computers are capable of handling and processing very large amounts of data easily and quickly. **A mainframe speed is so fast that it**

is measured in millions of tasks per milliseconds (MTM). While other computers became smaller, Mainframe computers stayed large to maintain the ever growing memory capacity and speed.

Mainframe computers are used in large institutions such as government, banks and large corporations. These institutions were early adopters of computer use, long before personal computers were available to individuals. "Mainframe" often refers to computers compatible with the computer architectures established in the 1960's. Thus, the origin of the architecture also affects the classification, not just processing power.

3. Mini Computers / Workstation :-

Mini computers, or Workstations, were computers that are one step above the micro or personal computers and a step below mainframe computers. They are intended to serve one user, but contain special hardware enhancements not found on a personal computer. They run operating systems that are normally associated with mainframe computers, usually one of the variants of the UNIX operating system.

4. **Super Computer**:-

A Super computer is a specialized variation of the mainframe. Where a mainframe is intended to perform many tasks, a Super computer tends to focus on performing a single program of intense numerical calculations. Weather forecasting systems, Automobile design systems, extreme graphic generator for example, are usually based on super computers.

Type	Word length	Memory	Processing speed	Application
Super computer	64-96 bits	256MB	400-10000mips	Sophisticated Scientific problems, Weather forecasting, Aerodynamics, Atomic Research etc
Main Frame	48-64 bits	128mb	30-100mips	Large industries, banks, airlines, NGO's.
Mini	32bits	96mb	10-30mips	Interactive and multi user environment.
Micro	8-32 bits	64MB	1-5MIPS	General purpose calculations, Industrial Control, Office Automation, e.t.c

According to Generations of Computers:

The history of computer development is often referred to in reference to the different generations of computing devices. Each generation of computer is characterized by a major technological development that fundamentally changed the way computers operate, resulting in increasingly smaller, cheaper, more powerful and more efficient and reliable devices.

1. First Generation (1940-1956): Vacuum Tubes:

The first computers used vacuum tubes for circuitry and magnetic drums for memory, and were often enormous, taking up entire rooms. They were very expensive to operate and in addition to using a great deal of electricity, generated a lot of heat, which was often the cause of malfunctions.

First generation computers relied on machine language, the lowest-level programming language understood by computers, to perform operations, and they could only solve one problem at a time. Input was based on punched cards and paper tape, and output was displayed on printouts.

Example: The UNIVAC and ENIAC computers are examples of first-generation computing devices. The UNIVAC was the first commercial computer delivered to a business client, the U.S. Census Bureau in 1951.

b) Second Generation (1956-1963): Transistors:-

Transistors replaced vacuum tubes and ushered in the second generation of computers. The transistor was invented in 1947 but did not see widespread use in computers until the late 1950s. The

transistor was far superior to the vacuum tube, allowing computers to become smaller, faster, cheaper, more energy-efficient and more reliable than their first-generation predecessors. Though the transistor still generated a great deal of heat that subjected the computer to damage, it was a vast improvement over the vacuum tube. Second-generation computers still relied on punched cards for input and printouts for output.

Second-generation computers moved from cryptic binary machine language to symbolic, or assembly, languages, which allowed programmers to specify instructions in words. High-level programming languages were also being developed at this time, such as early versions of COBOL and FORTRAN. These were also the first computers that stored their instructions in their memory, which moved from a magnetic drum to magnetic core technology.

The first computers of this generation were developed for the atomic energy industry.

1. Third Generation (1964-1971): Integrated Circuits

The development of the integrated circuit was the hallmark of the third generation of computers. Transistors were miniaturized and placed on silicon chips, called semiconductors, which drastically increased the speed and efficiency of computers.

Instead of punched cards and printouts, users interacted with third generation computers through keyboards and monitors and interfaced with an operating system, which allowed the device to run many different applications at one time with a central program that monitored the memory. Computers for the first time became accessible to a mass audience because they were smaller and cheaper than their predecessors.

2. Fourth Generation (1971-Present): Microprocessors

The microprocessor brought the fourth generation of computers, as thousands **of integrated circuits were built onto a single silicon chip**. What in the first generation filled an entire room could now fit in the palm of the hand. **The Intel 4004 chip**, developed in 1971, located all the components of the computer— from the central processing unit and memory to input/output controls—on a single chip.

In 1981 IBM introduced its first computer for the home user, and in 1984 Apple introduced the Macintosh. Microprocessors also moved out of the realm of desktop computers and into many areas of life as more and more everyday products began to use microprocessors.

As these small computers became more powerful, they could be linked together to form networks, which eventually led to the development of the Internet. Fourth generation computers also saw the development of GUIs, the mouse and handheld devices.

3. Fifth Generation (Present and Beyond): Artificial Intelligence)

Fifth generation computing devices, based on artificial intelligence, are still in development, though there are some applications, such as voice recognition, that are being used today. The use of parallel processing and superconductors is helping to make artificial intelligence a reality. Quantum computation and molecular and nanotechnology will radically change the face of computers in years to come. The goal of fifth-generation computing is to develop devices that respond to natural language input and are capable of learning and self-organization.

M.JOSHNA

Functional Unit (Or) Structure of a Computer System :

Every Digital computer systems consist of five distinct functional units. These units are as follows:

1. **Input unit**
2. **Memory unit**
3. **Arithmetic logic unit**
4. **Output unit**
5. **Control Unit**

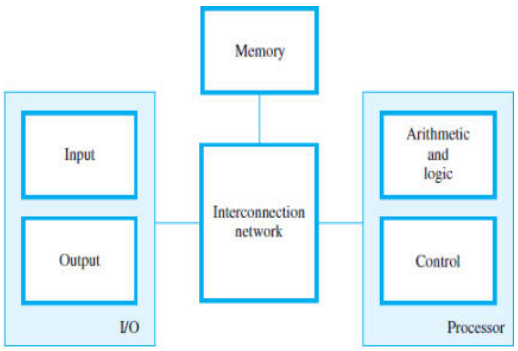


Figure 1.1 Basic functional units of a computer.

These units are interconnected by electrical cables to permit communication between them. A computer must receive both data and program statements to function properly and be able to solve problems. The method of feeding data and programs to a computer is accomplished by an input device. **Computer input devices read data from a source, such as magnetic disks, and translate that data into electronic impulses for transfer into the CPU**. **Example for input devices are a keyboard, a mouse, or a scanner**. **Central Processing Unit The brain of a computer system is the central processing unit (CPU). The CPU processes data transferred to it from one of the various input devices. It then transfers either an intermediate or final result of the CPU to one or more output devices. A central control section and work areas are required to perform calculations or manipulate data.** The CPU is the computing center of the system**. It consists of a control section, an arithmetic-logic section, and an internal storage section (main memory).** Each section within the CPU serves a specific function and has a particular relationship with the other sections within the CPU.

Input Unit: An input device is usually a **keyboard or mouse**, the input device is the conduit through which data and instructions enter a computer.

1. **The most common input device is the *keyboard*, which accepts letters, numbers, and commands from the user.**
2. **Another important type of input device is *the mouse*, which lets you select options from on- screen menus**. You use a mouse by moving it across a flat surface and pressing its buttons. A variety of other input devices work with personal computers, too:
3. **The trackball and touchpad are variations of the mouse and enable you to draw or point on the screen.**

   The joystick is a swiveling lever mounted on a stationary base that is well suited for playing video games

Memory unit: memory is used to store programs and data. **There are two classes of storage, called primary and secondary.**

Primary storage: It is a fast memory that operates at electronic speeds. Programs must stay in memory while they are being executed.**The memory contains a large number of semiconductor storage cells, each capable of storing one bit of information.** To provide easy access to any word in the memory, a distinct address is associated with each word location. Addresses are numbers that identify successive locations. A given word is accessed by specifying its address and issuing a control command.

The number of bits in each word is referred as the word length of the computer. Typical word lengths range from 16 to 64 bits.

Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of the processor.

1. **Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called *random access memory* (RAM).**
2. **The time required to access one word is called the *memory access time*.**
3. **The small, fast, Ram units are *called caches*. They are tightly coupled with the processor and are often contained on the same integrated circuit chip to achieve high performance**.
4. **The largest and slowest units are referred to as the *main memory*.**

Secondary storage: Secondary storage is used when large amounts of data and many programs have to be stored, particularly for information that is accessed infrequently.
Examples for secondary storage devices are *Magnetic Disks, Tape and Optical disks.*

Arithmetic-Logic Unit:- The arithmetic-logic section performs **arithmetic operations, such as addition, subtraction, multiplication, and division.**

Arithmetic-Logic Unit usually called the ALU is a digital circuit that performs two types of operations— arithmetic and logical.

Arithmetic operations are the fundamental mathematical operations consisting of addition, subtraction, multiplication and division.

Logical operations consist of comparisons. That is, two pieces of data are compared to see whether one is equal to, less than, or greater than the other. The ALU is a fundamental building block of the central processing unit of a computer.

Out put Unit:- An **output device** is any piece of computer hardware equipment used to communicate the results of data processing carried out by an information processing system (such as a computer) to the outside world.

In computing, input/output, or I/O, refers to the communication between an information processing system (such as a computer), and the outside world. Inputs are the signals or data sent to the system, and outputs are the signals or data sent by the system to the outside.

Examples of output devices:

1. Speaker
2. Headphones
3. Screen
4. Printer

Control Unit: All activities inside the machine are directed and controlled by the control unit. **Control Unit** is the part of the computer's central processing **unit** (CPU), which directs the **operation** of the processor. A **control unit** works by receiving input information to which it converts into **control** signals, which are then sent to the central processor

The Basic Operational Concepts of a Computer:-

1. **The program contains of a list of instructions is stored in the memory**.
2. **Individual instructions are brought from the memory into the processor, which execute the specified operations.**
3. **Data to be used as operands are also stored in the memory.**

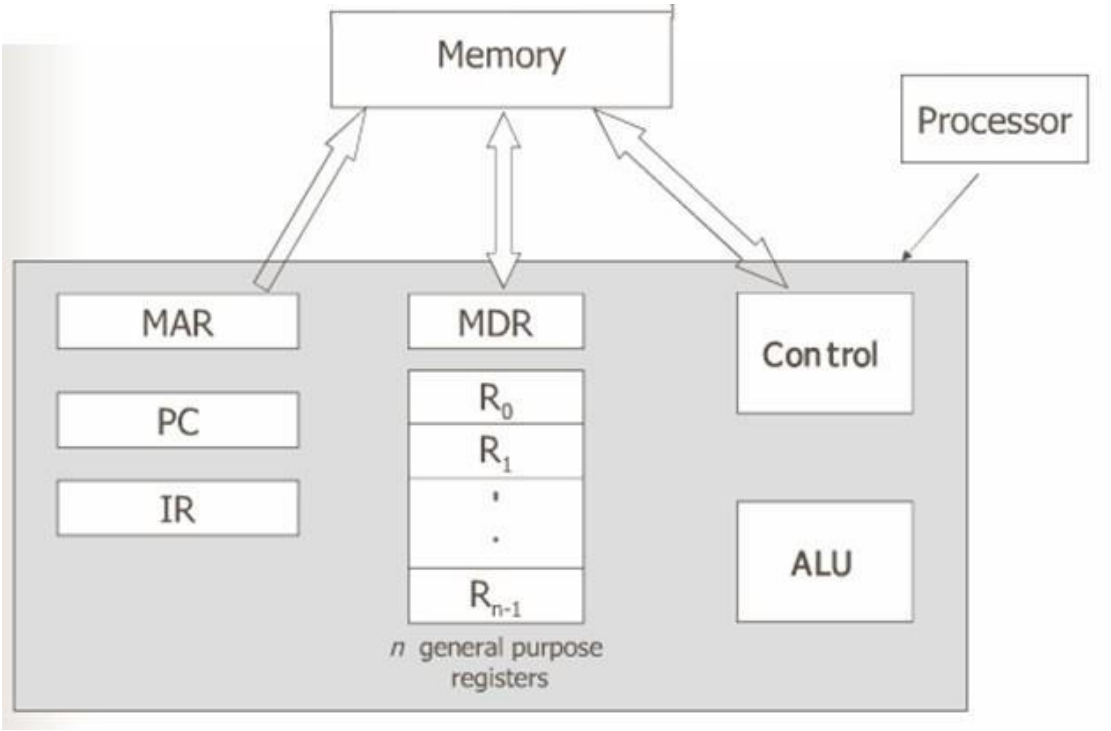
Add R1,R2,R3

In This instruction add is the operation perform on operands R1,R2 and place the result stored in R3.

M.JOSHNA

The top level view of the computer is as follows:

1. **Instruction register (IR):**
   1. **The instruction register holds the instruction that is currently being executed**.
   2. Its output is available to the control circuits, which generate the timing signals that control the various processing elements involved in executing the instruction.
2. **Program counter (PC):**
   1. The program counter is another specialized register.
   2. It keeps track of the execution of a program.
   3. **It contains the memory address of the next instruction to be fetched and executed.**
   4. During the execution of an instruction, the contents of the PC are updated to correspond to the address of the next instruction to be executed



3. **Memory address register (MAR) & Memory data register(MDR):-**
   1. These two registers facilitate communication with the memory.
   2. The MAR holds **the address of the location to be accessed**.
   3. The MDR contains the **data to be written into or read out of the addressed location.**

4. ***Operating steps for Program execution (or) Instruction Cycle :***
   1. **Execution of the program (stored in memory) starts when the PC is** set to point to the first instruction of the program.
   2. The contents of the **PC are transferred to the MAR** and a **Read control signal is sent to the memory.**
   3. **The addressed word is read out of the memory and loaded into the MDR**. **Next, the contents of the MDR are transferred to the IR**. At this point, the **instruction is ready to be decoded and executed.**
   4. **If the instruction involves an operation to be performed by the ALU, it is necessary to obtain the required operands.**

5.If an operand resides in memory (it could also be in a general purpose register in the processor), it has to be fetched by sending its address to the MAR and initiating a Read cycle.

6.When the operand has been read from the memory into the MDR, it is transferred from the MDR to ALU.

7.After one or more operands are fetched in this way, the ALU can perform the desired operation.

8.If the result of the operation is to be stored in the memory, then the result is ent to the MDR.

9.The address of the location where the result is to be stored is sent to the MAR, and a write cycle is initiated.

10. At some point during the execution of the current instruction, the contents of the PC are incremented so that the PC pints to the next instruction to be executed.

11. Thus, as soon as the execution of the current instruction is completed, a new instruction fetch may be started.

12. In addition to transferring data between the memory and the processor, the computer accepts data from input devices and sends data to output devices. Thus, some machine instructions with the ability to handle I/O transfers are provided.

Bus Structures:-

1. **BUS**:A group of lines(wires) that serves as a **connecting path for several devices of a computer is called a bus.**

The following are different types of busses:

1. Address Bus 2. Data Bus 3. Control Bus

The Data bus Carries(transfer) **data from one component (source) to other component (destination) connected to it. The data bus consists of 8, 16, 32 or more parallel signal lines.** The data bus lines are bi-directional. This means that CPU can read data on these lines from memory or from a port, as well as send data out on these lines to a memory location.

The Address bus is the set of lines that carry(transfer) address information about where in memory the data is to be transferred to or from. It is an unidirectional bus. **The address bus consists of 16, 20, 24 or more parallel signal lines.** On these lines CPU sends out the address of the memory location.

The Control Bus carries the **Control and timing information**. Including these three the following are various types of busses. They are

System Bus: A System Bus is usually a **combination of address bus, data bus, and control bus respectively**.

Internal Bus: The bus that operates **only with the internal circuitary of the CPU**.

External Bus: Buses which connects computer to **external devices is nothing but external bus.**

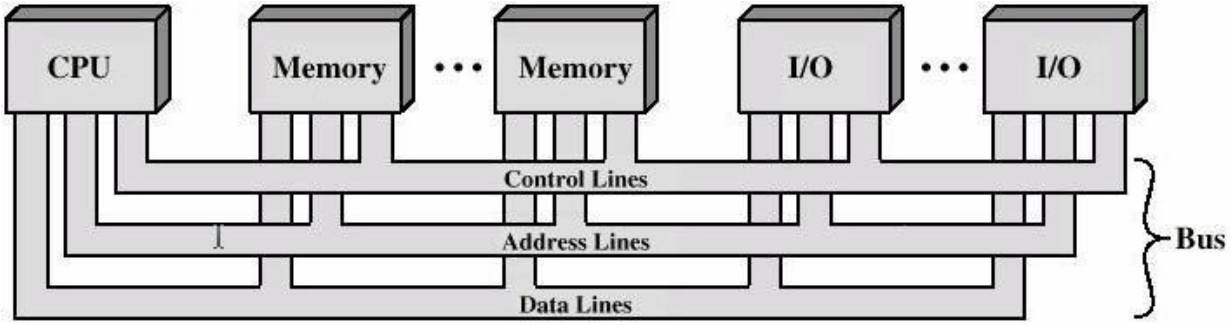
Back Plane: A Back Plane bus includes a row pf connectors into which system modules can be plugged in.

I/O Bus: The bus used by I/O devices to communicate with the CPU is usually reffered as I/O bus.

Synchronous Bus: While using Synchronous bus, data transmission between source and destination units takes place in a **given timeslot** which is already known to these units.

Asynchronous Bus: In this case the data transmission is governed by a special concept. That is handshaking control signals.

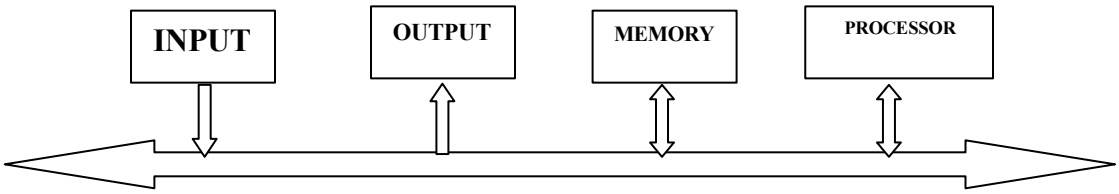
The Bus interconnection Scheme:-



M.JOSHNA

Single bus structure :-

1. A group of lines(wires) that serves as a **connecting path for several devices of a computer is called a bus.**
2. **In addition to the lines that carry the data, the bus must have lines for address and control purposes**.
3. The simplest way to interconnect functional units is to use a single bus, as shown below.



4. All units are connected to this bus. Because the bus can be used for only one transfer at a time, only two units can actively use the bus at any given time.
5. Bus control lines are used to arbitrate multiple requests for use of the bus.

ADVANTAGE:

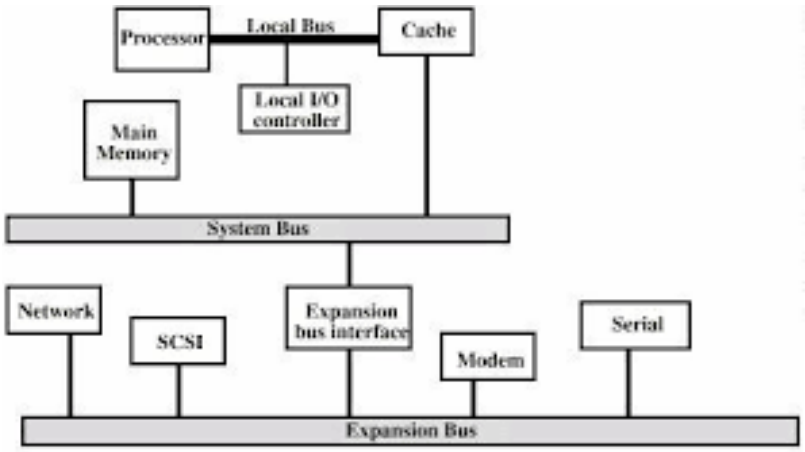
Its is **low cost and its flexibility** for attaching peripheral devices

DISADVANTAGE:

low performance because at time only one transfer

Traditional / Multiple bus Structure: There is a local bus that connects the processor to cache memory and that may support one or more local devices. There is also a cache memory controller that connects this cache not only to this local bus but also to the system bus.

On the system, the bus is attached to the main memory modules. In this way, I/O transfers to and from the main memory across the system bus do not interfere with the processor's activity. An expansion bus interface buffers data transfers between the system bus and the I/O controllers on the expansion bus.
Some typical I/O devices that might be attached to the expansion bus include: Network cards (LAN), SCSI (Small Computer System Interface), Modem, Serial Com etc..



Advantages: better performance

Disadvantage: increased cost.

M.JOSHNA

Software:-

A total computer system includes both software and Hardware .

1. **Hardware consists of physical components** and all associated equipment.
2. **Software refers to the collection programs that are written for the computer** and writing a program for a computer consists of specifying, directly or indirectly a sequence of machine instructions.
3. The computer software consists of the instructions and data that the computer manipulates to perform various data processing tasks.

Types:

1. Application software,
2. System software

System software: System software is used to run application software.

System software is a collection of programs that are executed as needed to perform functions such as

1. **Receiving and interpreting user commands.**
2. **Entering and editing application programs and sorting them as files in secondary storage devices.(Editor)**
3. **Managing the storage and retrieval of files in secondary storage devices.**
4. **Running standard application programs such as word processors, spread sheets, or games, with data supplied by the user.**
5. **Controlling I/O units to receive input information and produce output results.**
6. **Translating programs from high level language to low level language.(Assemblers)**
7. **Linking and running user-written application program with existing standard library routines, such as numerical computation packages.(Linker)**

Application software: Application software allows end users to accomplish one or more specific (not directly computer development related) tasks. Its usually written in high level languages, such as c ,c++, java. Typical applications include:

1. **Word processing**
2. **spreadsheet**
3. **computer games**
4. **databases**
5. **industrial automation**
6. **business software**
7. quantum chemistry and solid state physics software
8. telecommunications (i.e., the internet and everything that flows on it)
9. educational software
10. medical software
11. military software
12. molecular modeling software
13. image editing
14. simulation software
15. Decision making software

Compiler:- A **compiler** is a computer program (or set of programs) that **transforms source code written in a computer language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*).** The most common reason for wanting to transform source code is to create an executable program. The name "compiler" is primarily used for programs that translate

source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). A program that translates from a low level language to a higher level one is a *decompiler*. A program that translates between high-level languages is usually called a *language translator*, *source to source translator*

Linker: - Linker is a program in a system which helps to link a object modules of program into a single object file. It performs the process of linking. Linker are also called link editors. Linking is process of collecting and maintaining piece of code and data into a single file. Linker also link a particular module into system library. It takes object modules from assembler as input and forms an executable file as output for loader. Linking is performed at both compile time, when the source code is translated into machine code and load time, when the program is loaded into memory by the loader. Linking is performed at the last step in compiling a program.

Assembler: - An assembler is a program that converts assembly language into machine code. **It takes the basic commands and operations from assembly code and converts them into binary code that can be recognized by a specific type of processor**. Assemblers are similar to compilers in that they produce executable code. However, assemblers are more simplistic since they only convert low-level code (assembly language) to machine code. Since each assembly language is designed for a specific processor, assembling a program is performed using a simple one-to-one mapping from assembly code to machine code.

Loader:- A loader is a major component of an operating system that ensures all necessary programs and libraries are loaded, which is essential during the startup phase of running a program. It places the libraries and programs into the main memory in order to prepare them for execution.

Performance

Performance: - The most important measure of the performance of a computer is how quickly it can compute programs. The speed with which a computer executes programs is affected by the design of its hardware and its machine language instructions. To represent the performance of a processor, we should consider only the periods during which the processor is active.

At the start of execution, all program instructions and the required **data are stored in the memory as shown below. As execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache**. **When the execution of instruction calls for data located in the main memory, the data are fetched and a copy is placed in the cache**. Later, if the same instruction or data item is needed a second time, it is read directly from the cache.

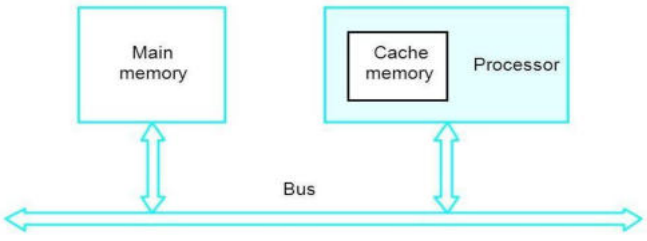


Figure 1.5. The processor cache.

M.JOSHNA

Computer performance is often described in terms of clock speed (usually in MHz or GHz). This refers to the cycles per second of the main clock of the CPU. Performance of a computer depends on the following factors.

1. Processor clock:-

1.Processor circuits are controlled by a timing signal called a clock. A clock is a microchip that regulates speed and timing of all computer functions.

2.**Clock Cycle** is the speed of a computer processor, or CPU, which is the amount of time between two pulses of an oscillator. Generally speaking, the higher number of pulses per second, the faster the computer processor will be able to process information

3.CPU **clock speed**, or **clock rate**, is measured in Hertz — generally in gigahertz, or GHz. A CPU's **clock speed rate** is a measure of how many **clock** cycles a CPU can perform per second

4.To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps, such that each step can be completed in one clock cycle.

5.The length P of one clock cycle is an important parameter that affects processor performance.

6.Its inverse is the clock rate, R = 1/P, which is measured in cycles per second.

7.If the clock rate is 500(MHz) million cycles per second, then the corresponding clock period is 2 nanoseconds.

2. Basic performance equation:- The **Performance Equation** is a term used in computer science. It refers to the calculation of the performance or speed of a central processing unit (CPU).

Basically the *Basic Performance Equation [BPE]* is an equation with 3 parameters which are required for the calculation of "Basic Performance" of a given system. It is given by

T = (N*S)/R

Where **'T'** is the ***processor time*** [Program Execution Time] required to execute a given program written in some high level language .The compiler generates a machine language object program corresponding to the source program.

'N' is the **total number of steps required to complete program execution**. 'N' is the actual number of instruction executions, not necessarily equal to the total number of machine language instructions in the object program. Some instructions are executed more than others (loops) and some are not executed at all (conditions).

'S' is the **average number of basic steps each instruction execution requires**, where each basic step is completed in one clock cycle. We say average as each instruction contains a variable number of steps depending on the instruction.

'R' is the **clock rate** [In cycles per second]

3. Pipelining and Super scalar operation:-

1.A substantial improvement in performance can be achieved by overlapping the execution of successive instructions, using a technique called pipelining.

2.Consider the instruction

3.Add R1, R2, R3

4.Which adds the contents of registers R1 and R2, and places the sum into R3

5.The contents of R1 and R2 are first transferred to the inputs of the ALU.

6.After the add operation is performed, the sum is transferred to R3.

7.Processor can read the next instruction from the memory while the addition operation is being performed.

8.Then, if that instruction also uses the ALU, its operands can be transferred to the ALU inputs at the same time that the result of add instruction is being transferred to R3.

9.Thus, pipelining increases the rate of executing instructions significantly.

4. Super scalar operation:-

1.A higher degree of concurrency can be achieved if multiple instruction pipelines are implemented in the processor.

2.This means that multiple function units are used, creating parallel paths through which different instructions can be executed in parallel.

3.With such an arrangement, it becomes possible to start the execution of several instructions in every clock cycle.

4. This mode of execution is called super scalar operation.

5.Clock rate:-

1.There are two possibilities for increasing the clock rate, R.

2.First, improving the Integrated Circuit technology makes logic circuit faster, which reduces the needed to complete a basic step. This allows the clock period, P, to be reduced and the clock rate, R, to be increased.

3.Second, reducing the amount of processing done in one basic step also makes it possible to reduce the clock period, P.

6. Instruction set: CISC and RISC:-

1.The terms CISC and RISC refer to design principles and techniques.

2.RISC: Reduced instruction set computers.

3.Simple instructions require a small number of basic steps to execute.

4.For a processor that has only simple instructions, a large number of instructions may by need to perform a given programming task. This could lead to a large value of N and a small value for S.

5.It is much easier to implement efficient pipelining in processors with simple instruction sets.

6.CISC: Complex instruction set computers.

7.Complex instructions involve a large number of steps.

8.If individual instructions perform more complex operations, fewer instructions will be needed, leading to a lower value of N and a larger value of S.

9.Complex instructions combined with pipelining would achieve good performance.

7.Optimizing Compiler:-

1.A compiler translates a high-level language program into a sequence of machine instructions.

2.To reduce N, we need to have a suitable machine instruction set and a compiler that makes good use of it.

3.An optimizing compiler takes advantage of various features of the target processor to reduce the product N * S.

4. The compiler may rearrange program instructions to achieve better performance.

8. Performance measurement:-

1.SPEC rating.

2.A nonprofit organization called" System Performance Evaluation Corporation" (SPEC) selects and publishes representative application programs for different application domains.

3.The SPEC rating is computed as follows.

4.SPEC rating = Running time on the reference computer
Running time on the computer under test.

5.Thus SPEC rating of 50 means that the computer under test is 50 times faster than the reference computer for these particular benchmarks.

6.The test is repeated for all the programs in the SPEC suite, and the geometric mean of the results is computed.

7.Let SPEC, be the rating for program 'i' in the suite.
The overall SPEC rating for the computer is
given by

$$\text{SPEC rating} = \prod_{i=1}^{n} (\text{SPEC}_i)$$

Where n is the number of programs in the suite.

M.JOSHNA

UNIT-2

Syllabus: Machine Instruction and Programs: Instruction and Instruction sequencing: Register Transfer notation, Assembly Language Notation, Basic Instruction types**, Addressing Modes, Basic Input / Output operations, the role of Stacks and queues in computer programming equation. Component of instructions:** logic instructions shift and rotate instructions.

Instructions and Instruction Sequencing: computer **programming consists of a sequence of small steps,** such as **adding two numbers**, **testing for particular condition** ,**reading the character from keyboard and sending a character to be displayed on screen.**

A computer must have instructions capable of performing **four types of operations:**

1. **Data transfers between the memory and the processor registers**
2. **Arithmetic and logic operations on data**
3. **Program sequencing and control**
4. **I/O transfers**

Register Transfer Notation: It is used to transfer information from one location to other location inside the computer. In RTN, source is always a value specified on right hand side of "← ". Destination is always a processor register, specified on left hand side.

Syntax :

Register ← Source

The right hand side of RTN is always denotes a value and the **left hand side is the name of a location where the value is to be placed**. **Source can be processor register, I/O register, memory location**, but destination register is always a processor register. **RTN uses square brackets to indicate content of location.** These braces are always placed only around the Source. For example,

1. **R3 ← [R1] + [23]**

   This operation that adds the contents of registers R1 and R2, and places their sum into register R3

2. **R_2 ← [LOC], means that the contents of memory location LOC are transferred into processor register R2.**

Assembly Language Notation: Assembly Language Notation is a type of notation which is used to represent machine instructions and programs.

For example:

LOAD LOC, R2

a generic instruction that causes the transfer, from memory location LOC to processor register R2, is specified by the statement

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R2 are overwritten. The name Load is appropriate for this instruction, because the contents read from a memory location are *loaded* into a processor register.

The second example :

ADD R4, R2, R3

Adding two numbers contained in processor registers R2 and R3 and placing their sum in R4 can be specified by the assembly-language statement

In this case, registers R2 and R3 hold the source operands, while R4 is the destination.

An *instruction* specifies an operation to be performed and the operands involved. In the above examples, we used the English words Load and Add to denote the required operations. **In the assembly-language instructions of actual (commercial) processors, such operations are defined by using *mnemonics*,** which are typically abbreviations of the words describing the operations. For example, the operation Load may be written as LD, while the operation Store, which transfers a word from a processor register to the memory, may be written as STR or ST. Assembly languages for different processors often use different mnemonics for a given operation.

Basic Instruction types: An instruction is of various lengths depending upon the number of addresses it contains. Generally CPU organization is of three types on the basis of number of address fields:

1. **Single Accumulator organization**
2. **General register organization**
3. **Stack organization**

1. **In first organization operation is done involving a special register called accumulator**.

2. **In second on multiple registers are used for the computation purpose**.

3.In third organization the work on stack basis operation due to which it does not contain any address field. On the basis of number of addresses instructions are classified as:

Three address Instructions: This instruction has three operands(address fields) to specify a register or a memory location.

1. **Syntax**

: **operation source1, source2**,

EX: **Add A, B, C [C<-[A]+[B])**

Where **A, B are called source operands**, **C is called destination operand.**

2. **Two address Instructions**: This instruction has two operands(address fields) to specify a register or a memory location.

3. **Syntax : operation source , destination**.

For example,

Add A, C (C ← [A] + [C])

4.**One address Instructions: This instruction has one operand** (address field) to specify a **register or a memory location. This use a implied Accumulator(AC) Register for data manipulation.** One operand is in AC and other is in register or memory location. Implied means that the CPU already know that one operand is in AC so there is no need to specify it. For example,

LOAD A (AC ← [A])

ADD B (AC ← [AC] + [B])

STORE C (C ← [AC])

5.**Zero address Instructions:** This instruction has zero address fields. A stack based computer do not use address field in instruction**. It uses stack operations PUSH and POP to perform operations.** To evaluate a expression first it is converted to revere Polish Notation i.e. Post fix Notation. For example,

Push A (TOS ← [A])

Push B (TOS ← [B])

Add (TOS ← [A] + [B])

Pop C (C ← [TOS])

Example: evaluate X = (A + B) * (C + D)

Three Address:

Add A, B, R1 (R1 ← [A] + [B])

Add C, D, R2 (R2 ← [C] + [D])

Mul R1, R2, X (X ← [R1] * [R2])

Two Address:

Move A, R1 (R1 ← [A])

Add B, R1 (R1 ← [R1] + [B])

Move C, R2 (R2 ←[C])

Add D, R2 (R2 ← [R2] + [D])

Mul R1, R2 (R2 ← [R1] * [R2])

Move R2, X X ← [R2]

One Address:

Load A (AC ← [A])

Add B (AC ← [AC] + [B])

Store T1 (T1 ← [AC])

Load C (AC ← [C])

Add D (AC ← [AC] + [D])

Mul T1 (AC ← [AC] * [T1])

Store X (X ← [AC])

Zero address:

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 ← ([A] + [B]) * ([C] + [D])**
Pop X	**(X ← [A] + [B]) * ([C] + [D])**

Instruction Execution and Straight-Line Sequencing: To begin **executing** a program, the address of its first **instruction** (I in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and **execute instructions**, one at a time, in the order of increasing addresses. This is called **straight-line sequencing**. For example, consider the following assembly instruction which add contents of two memory locations. i.e. C ← [A] + [B]. The following diagram shows a possible program segment for this task as it appears in the memory of a computer.

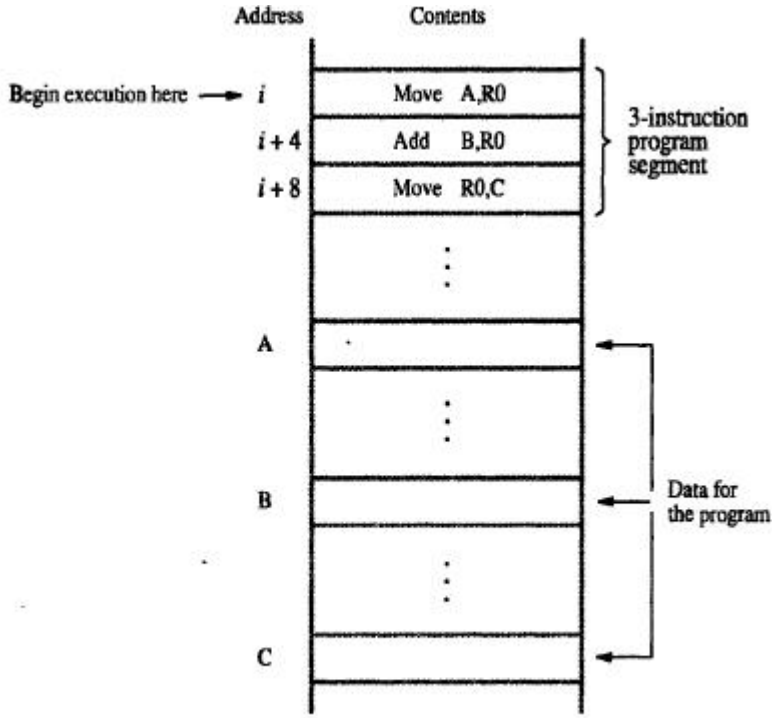


Figure 2.8 A program for C ← [A] + [B].

The four instructions of the program are in successive word locations, starting at location *i*. Since, each instruction is 4 bytes long, the second, third, and fourth instructions are at addresses *i* + 4, *i* + 8, and *i* + 12. The processor contains a register called the *program counter* (PC), which holds the address of the next instruction to be executed. To begin executing a program, the address of its first instruction (*i* in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing*. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Store instruction at location *i* + 12 is executed, the PC contains the value *i* + 16, which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the first phase, called *instruction fetch*, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the *instruction register* (IR) in the processor. At the start of the second phase, called *instruction execute*, the instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This involves a small number of steps such as fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location. At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction. When the execute phase of an instruction is completed, the PC contains the address of the next instruction, and a new instruction fetch phase can begin.

Branching: Normally, the instructions executed in linear fashion through the program, and the address of the instructions is obtained from PC in the control unit. This sequence is interrupted when a branch instruction is executed, at such a time the address field of the Branch instruction is inserted into the PC and the process continues. Consider the task of adding a list of *n* numbers. The following diagram shows straight line sequencing program to add list of n numbers.

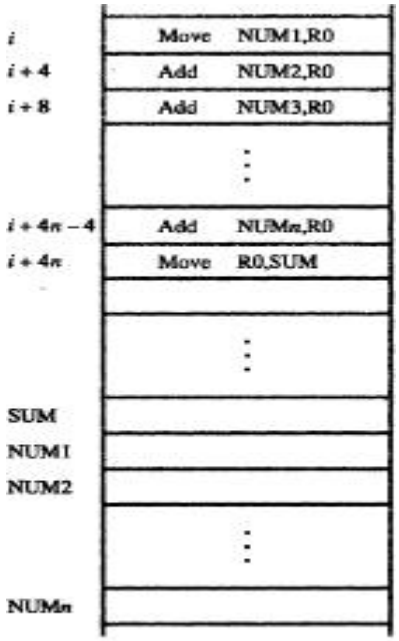


Figure 2.9 A straight-line program for adding *n* numbers.

The addresses of the memory locations containing the *n* numbers are symbolically given as NUM1, NUM2, . . . , NUM*n*, and separate Add instructions is used to add each number to the contents of register R0. After all the numbers have been added, the result is placed in memory location SUM.

Instead of using a long list of Add instructions, it is possible to implement a program loop in which the instructions read the next number in the list and add it to the current sum. To add all numbers, the loop has to be executed as many times as there are numbers in the list. The following shows the structure of the desired program. The body of the loop is a straight-line sequence of instructions executed repeatedly. It starts at location LOOP and ends at the instruction Branch > 0. During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to R0.

Assume that the number of entries in the list, *n*, is stored in memory location N, as shown. Register R1 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R1 at the beginning of the program. Then, within the body of the loop, the instruction

Decrement R1

reduces the contents of R1 by 1 each time through the loop. Execution of the loop is repeated as long as the contents of R1 are greater than zero.

Next use branch instruction. This type of instruction loads a new address into the program counter. As a result, the processor fetches and executes the instruction at this new address, called the *branch target*, instead of the instruction at the location that follows the branch instruction in sequential address order. A *conditional branch* instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

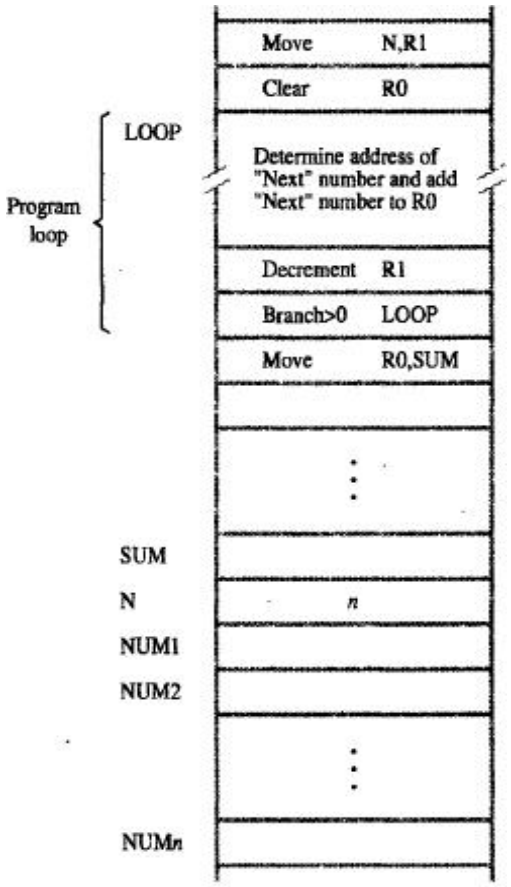


Figure 2.10 Using a loop to add *n* numbers.

The instruction

Branch > 0 LOOP

is a conditional branch instruction that causes a branch to location LOOP if the contents of register R1 are greater than zero. This means that the loop is repeated as long as there are entries in the list that are yet to be added to R0. At the end of the *n*th pass through the loop, the Subtract instruction produces a value of zero in R2, and, hence, branching does not occur. Instead, the Store instruction is fetched and executed. It moves the final result from R0 into memory location SUM.

Condition Codes: The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions. This is done by recording the required information in individual bits, often called condition code flags. These flags are grouped together in a special processor register called condition code register or status register. Individual condition code flags are set to 1 or cleared to 0, depending on outcome of operation performed. Four commonly used flags are

1. **N (negative):** Set to 1 if the result is negative; otherwise, cleared to 0.
2. **Z (zero):** Set to 1 if the result is zero; otherwise, cleared to 0.
3. **V (overflow):** Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0.
4. **C (carry):** Set to 1 if a carry-out results from the operation; otherwise, cleared to 0.

M.JOSHNA

Addressing Modes: The different ways for specifying the locations of instruction operands are known as *addressing modes*.

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*)	EA = [R*i*]
	(LOC)	EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address Value = a signed number

1. **Implementation of Variables and constants:** In assembly language**, a variable is represented by allocating a register** or **memory location to hold its value**. Thus, the value can be changed as needed using appropriate instructions. **Variables** can be represented by register and absolute addressing modes.

    1. **Immediate mode: Immediate mode:** The operand is given explicitly in the instruction. For example, the instruction

        MOV #200, R0

        Moves the value 200 to register R0. Constants are frequently used in high level languages. For example, the statement A = B + 6. This statement can be represented as

        MOV B, R1
        ADD #6, R1
        MOV R1, A

2. **Register mode: The operand is the contents of a processor register**; the name of the register is given in the instruction.
    1. **For example**, instruction

        Add R1, R2, R3

        Uses the Register mode for all three operands. Registers R1 and R2 hold the two source operands, while R3 is the destination.

2. **Absolute mode(direct): The operand is in a memory location**; the address of this location is given explicitly in the instruction. The Absolute mode is used in the instruction

Add A,B,C

Uses the Register mode for all three operands. Registers A and B hold the two source operands, while C is the destination.

1. **Indirection and Pointers:** Here, **the instruction does not give the operand or its address explicitly. Instead, it provides information from which memory address of the operand can be determined**. **This information is called as Effective address.**

Indirect mode: In this mode, the effective address of an operand is the contents of a register or memory location whose address appears in the instruction. Indirection can be represented by placing name of the register or the memory address given in the instruction in parenthesis. For example, to execute the **Add instruction shown below the processor uses the value B which is in register R1**, **as the effective address of operand**. It requests a read operation from memory to read the contents of location B. the value read is the required operand, which adds to the contents of register R0.

Add (R1),R0

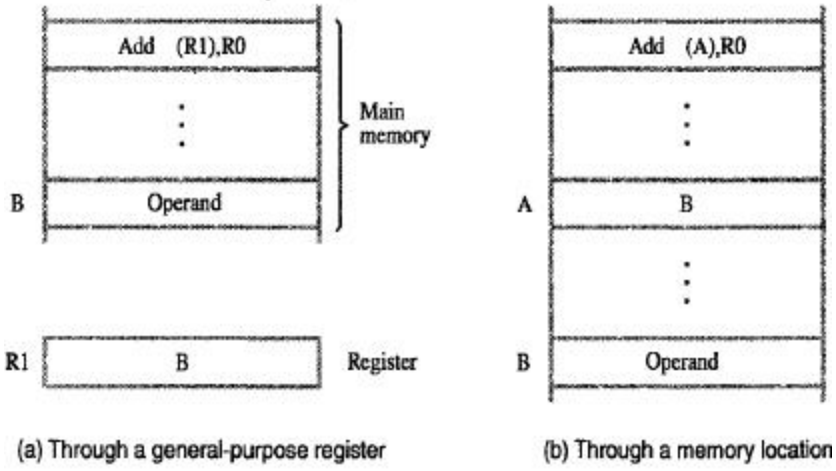


Figure 2.11 Indirect addressing.

Indirect addressing through memory location is also possible as shown above. In this case, the processor first reads the contents of memory location A, and then requests a second read operation using the value B as an address to obtain the operand. The register or memory location that contains the address of an operand is called a pointer.

Consider a program for adding a list of numbers using indirect addressing as shown below.

Address	Contents		
	Move	N,R1	Initialization
	Move	#NUM1,R2	
	Clear	R0	
LOOP	Add	(R2),R0	
	Add	#4,R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0,SUM	

Here, register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization section of the program loads the counter value n from memory location N to R1 and uses the immediate addressing mode to place the address value NUM1, which is the address of first number in the list, into R2. Then it clears R0 to 0. The instruction ADD (R2), R0 fetches the operand at location NUM1 and adds it ro R0. The second ADD instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2, when the above instruction is executed in the second pass through the loop and son on.

Consider the C-language statement A = * B; where B is a pointer variable. This statement may be compiled into

MOVE B, R1
MOVE (R1), A

Using indirect addressing through memory, the same action can be achieved with

MOVE (B), A

2. **Indexing and Arrays:** It is useful in dealing with lists and arrays.

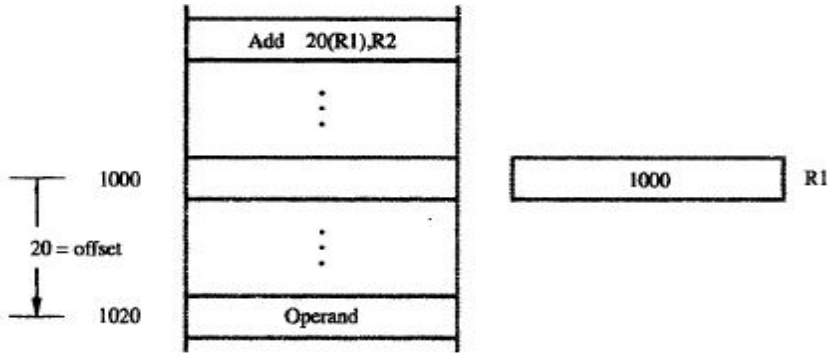
1. **Index Mode**, the effective address of the operand is generated by adding a constant value to the contents of a register. The register used may be either a general purpose register or index register. Symbolically index mode can be represented as

X (R_i)

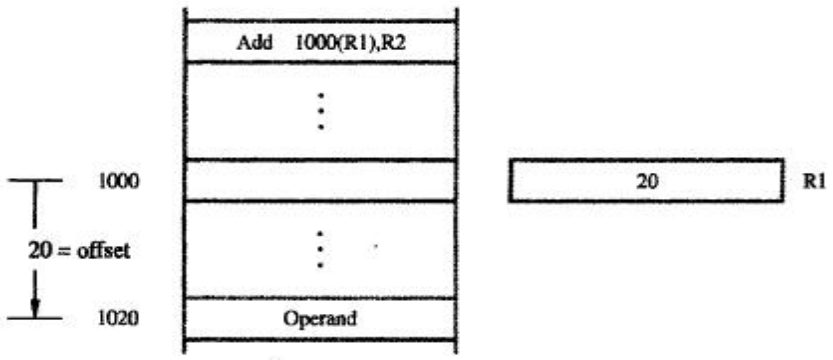
Where X denotes the constant value contained in the instruction and R_i is the name of the register involved. The effective address of the operand is given by

EA = X + [R_i]

The following shows the way of using Index mode.



(a) Offset is given as a constant



(b) Offset is in the index register

Figure 2.13 Indexed addressing.

In first case, the index register R1 contains the address of a memory location and the value x defines an offset from this address to the location where the operand is found.

In second case, the constant X corresponds to a memory address and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values: one is given explicitly in the instruction and the other is stored in a register.

Consider a simple example involving a list of test scores for students taking a given course. Assume that the list of scores beginning at location LIST as shown below.

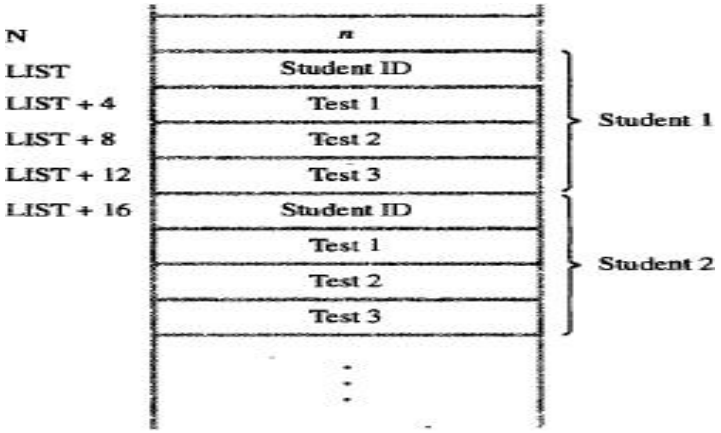


Figure 2.14 A list of students' marks.

A four word memory block comprises a record that stores the relevant information for each student. Each record consists of the students ID, followed by the scores the student earned on three tests. There are n students in the class, and the value n is stored in location N immediately in front of the list. Suppose our aim is to find the sum of all scores obtained on each of the tests and store these sums in memory locations SUM1, SUM2, and SUM3. A possible program for this task is given below.

On the first pass through the loop, test scores of the first student are added to the running sums held in registers R1, r2, and R3, which are initially cleared to zero. These scores are accessed using index addressing modes 4(R0), 8(R0), and 12(R0). The index registers then incremented by 16 to point to the ID location of the second student. Register R4, which is initialized to the value n is decremented by 1 at the end of each pass through the loop. When the contents of R4 reached to 0, all the student records have been accessed, and the loop terminates. Until then, the conditional branch instruction transfer control back to the start of the loop to process the next record. The last three instructions transfer the accumulated sums from R1, R2, and R3 into memory locations SUM1, SUM2, and SUM3.

2. **Base with Index Mode**: In this mode, effective address is generated by adding the contents of base register with the contents of index register. It is represented as shown below.

$$\textbf{Add} \quad (R_i, R_j)$$

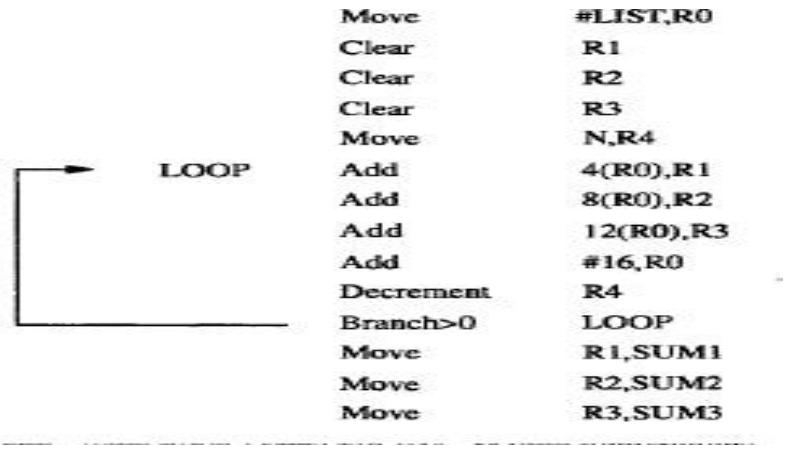
The effective address is the sum of the contents of registers R_i, and R_j. The second register is usually called the base register.

$$EA = [R_i] + [R_j]$$

3. **Base with Index and Offset:** In this mode, effective address is the sum of the constant X and the contents of registers R_i and R_j. It is represented as sown below.

$$X (R_i, R_j)$$

$$EA = X + [R_i] + [R_j]$$



3. **Relative Addressing:** In Relative Addressing, the Program counter is used instead of a general purpose register.'
   1. **Relative Mode:** In this mode, the effective address is determined by adding the contents of program counter to offset value. It is represented symbolically as

      X (PC)

      The effective address of the operand is given by

      EA = X + [PC]

      This mode can be used to access data operands. It's most common use is to specify the target address in branch instructions. An instruction such as

      Branch > 0 LOOP

      Causes program to go to branch location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as n offset from the current value of the program counter. For example, suppose that the relative mode is used to generate the branch target address LOOP in the branch instruction of the program using indirect addressing. Assume that the four instructions of the loop body, starting at LOOP, are located at memory locations 1000, 1004, 1008, and 1012. Hence, the updated contents of the PC at the time the branch target address is generated will be 1016. To branch to location LOOP (1000), the offset value needed is X = -16.

4. **Additional Modes:** The additional modes like auto increment and auto decrement are useful for accessing data items in successive locations in memory.
   1. **Auto increment Mode:** The effective address of the operand is the contents of a register specified in the instruction. **After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.** It is represented as

      (R_i) +

      It normally increments 1, but in byte sized operands or byte addressable memory. Thus, the increment is 1 for 8 bit operands, 2 for 16 bit and 4 for 32 bit operands. The effective address of the operand is

      EA = [R_i]; Increment R_i
   2. **Auto decrement Mode:** The contents of a register specified in the instruction **are first automatically decremented and are then used as effective address of the operand**. It is represented as

      - (R_i)

      The effective address of the operand is

      Decrenebt R_i;

      EA = [R_i]

The following program describes how to use auto increment mode

```
                Move        N,R1          }
                Move        #NUM1,R2      } Initialization
                Clear       R0            }
        LOOP    Add         (R2)+,R0
                Decrement   R1
                Branch>0    LOOP
                Move        R0,SUM
```

Basic Input / Output Operations: Input / Output operations are essential which has a significant effect on performance of a computer**. An I/O device is connected to the interconnection network by using a circuit, called the *device interface*,** which provides the means for data transfer and for the exchange of status and control information needed to facilitate the data transfers and govern the operation of the device. The interface includes some registers that can be accessed by the processor.

Consider a task that reads in **character input from a keyboard** and **produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as Program controlled I/O.** A solution to this problem is as follows: On output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character, and so on. Input is sent from the keyboard in a similar way; the processor waits for a signal indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard. Then the processor proceeds to read that code.

The keyboard and the display are separate devices as shown below.

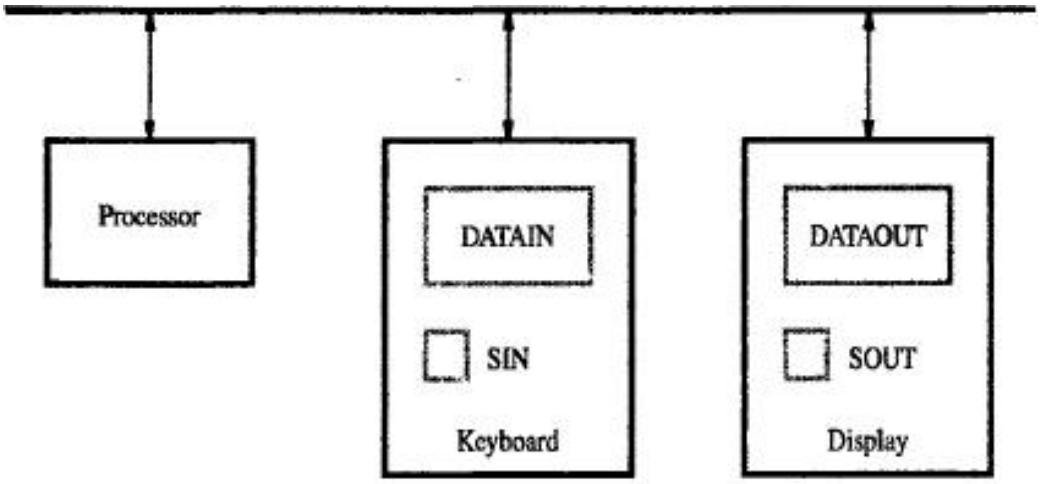


Figure 2.19 Bus connection for processor, keyboard, and display.

Consider the problem of moving a character from the keyboard to the processor. Striking a key stores the corresponding character code in an 8-bit buffer register DATAIN which is associated with the keyboard. To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1 (initially SIN=0).

A program monitors SIN, and when **SIN is set to 1, the processor reads the contents of DATAIN**.

When the character is transferred to the processor, SIN is again set to 0 and the process repeats.

To transfer the characters from processor to display, a buffer register, DATAOUT, and a status control flag, SOUT, are used.

When SOUT equals 1, the display is ready to receive a character. Under program control, the processor monitors SOUT, and when SOUT is set to 1, the processor transfers a character code to DATAOUT clears SOUT to 0; when the display device is ready to receive a second character, SOUT is again set to 1.

The processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to register R1 by the **following sequence of operations**.

READWAIT Branch to READWAIT if SIN = 0
MOVE DATAIN, R1

The contents of the keyboard character buffer DATAIN can be transferred to processor register instruction is

MoveByte DATAIN ,R1

An analogous sequence of operations is used for transferring output to display as shown below.

WRITEWAIT Branch to WRITEWAIT if SOUT = 0
MOVE R1, DATAOUT

The contents of the processor register R1 transferred to display buffer DATAOUT instruction is

MoveByte R1,OUTDATA

The following program explains how to read sequence of characters and display it.

	Move	#LOC,R0	Initialize pointer register R0 to point to the address of the first location in memory where the characters are to be stored.
READ	TestBit	#3,INSTATUS	Wait for a character to be entered
	Branch=0	READ	in the keyboard buffer DATAIN.
	MoveByte	DATAIN,(R0)	Transfer the character from DATAIN into the memory (this clears SIN to 0).
ECHO	TestBit	#3,OUTSTATUS	Wait for the display to become ready.
	Branch=0	ECHO	
	MoveByte	(R0),DATAOUT	Move the character just read to the display buffer register (this clears SOUT to 0).
	Compare	#CR,(R0)+	Check if the character just read is CR (carriage return). If it is not CR, then
	Branch≠0	READ	branch back and read another character. Also, increment the pointer to store the next character.

Here assume that 3rd bit in registers INSTATUS and OUTSTATUS corresponds to SIN and SOUT respectively.

M.JOSHNA

The role of STACKS and QUEUES in computer programming equation: In order to organize the control and information linkage between the main program and the subroutine, a data structure called a stack is used. A stack is the list of data elements, usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called as top of the stack, and another end is called as bottom. The structure is sometimes referred as pushdown stack. Stack follows Last-In-First-Out (LIFO) strategy, where elements inserted last will be the element removed first. Two basic operations that can be performed on stack are PUSH and POP, which add and remove elements from top of the stack respectively.

Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations. Assume that first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successive lower address locations. The following diagram shows a stack of word data items in the memory of a computer.

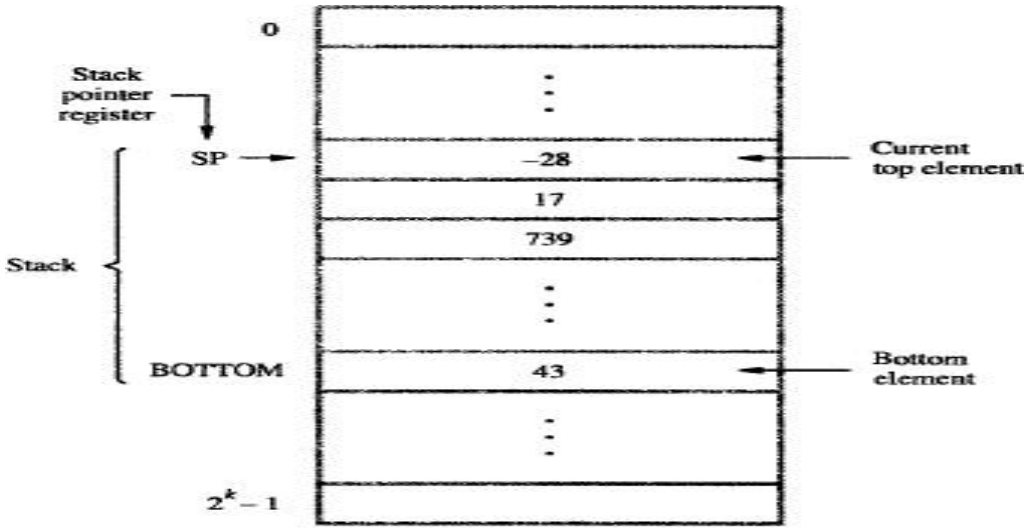


Figure 2.21 A stack of words in the memory.

It contains numerical values, with 43 at bottom and -28 at the top. A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the Stack Pointer (SP). It could be one of the general purpose registers or a register dedicated to this function. Assume a byte addressable memory with 32-bit word length, the PUSH operation can be implemented as

```
Subtract   #4,SP
Move       NEWITEM,(SP)
```

Where the Subtract instruction subtracts the source operand 4 from the destination operand contained in SP and places the result in SP. These two instructions move the word from location NEWITEM onto the top of the stack, decrementing the stack pointer by 4 before the move.

```
Move   (SP),ITEM
Add    #4,SP
```

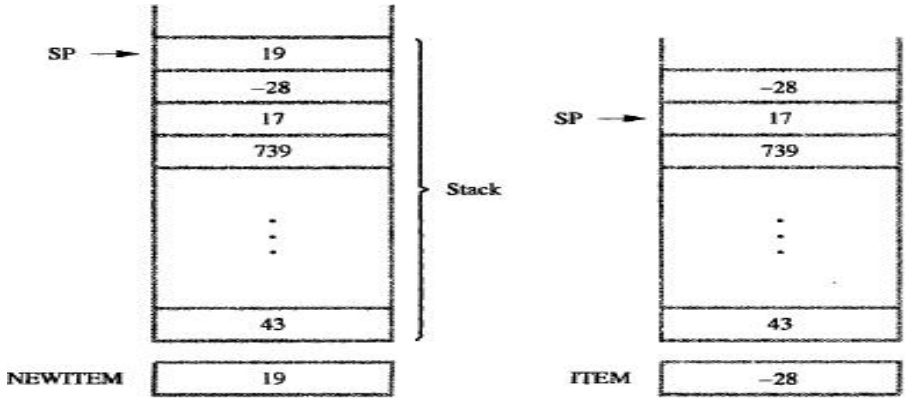
if the processor has the auto increment and auto decrement address modes the PUSH operation can be performed by the instruction:

```
Move   NEWITEM,–(SP)
```

The POP operation can be implemented as:

```
Move   (SP)+,ITEM
```

These two instructions move the top value from the stack into location ITEM and then increment the stack pointer by 4, so that it points to the new top element as shown below.



(a) After push from NEWITEM (b) After pop into ITEM

Suppose that a stack runs from location 2000 (BOTTOM) down no further than 1500. The stack pointer is loaded initially with the address value 2004. SP is decremented by 4 before new data are stored on the stack. Hence, an initial value of 2004 means that the first item pushed on to the stack will be at location 2000. To prevent either pushing an item on a full stack or popping an item off an empty stack, the single instruction PUSH and POP operations can be replaced by the following instruction sequences.

SAFEPOP	Compare	#2000,SP	Check to see if the stack pointer contains an address value greater than 2000. If it does, the stack is empty. Branch to the routine EMPTYERROR for appropriate action.
	Branch>0	EMPTYERROR	
	Move	(SP)+,ITEM	Otherwise, pop the top of the stack into memory location ITEM.

a safe pop operation

SAFEPUSH	Compare	#1500,SP	Check to see if the stack pointer contains an address value equal to or less than 1500. If it does, the stack is full. Branch to the routine FULLERROR for appropriate action.
	Branch≤0	FULLERROR	
	Move	NEWITEM,−(SP)	Otherwise, push the element in memory location NEWITEM onto the stack.

a safe push operation

Another useful data structure that is similar to the stack is called a Queue. Data are stored in and retrieved from a queue on a First-In-First-Out (FIFO) basis. Thus, if queue grows in the direction of increasing addresses in memory, new data are added at the back (high address end) and retrieved from the front (low address end) of the queue.

There are two important differences between stack and queue. One end of the stack is fixed (the bottom), while the other end raises and falls as data are pushed and popped. A single pointer is needed to point to the top of the stack at any given time. On the other hand, both ends of a queue move to higher addresses as data are added at the back and removed from the front. So, two pointers are needed to keep track of two ends of the queue.

Another difference between stack and queue is that, a queue would continuously move through memory of a computer in the direction of higher addresses. This can be avoided by using Circular Queue.

M.JOSHNA

Additional Instructions (Or) Component of Instructions:

Additional Instructions (Or) Component of Instructions are two types

1.Logic Instructions

2.Shift and Rotate Instructions

Logic Instructions: Logic operations such as AND, OR, and NOT applied to individual bits, are the basic building blocks of digital circuits. It is also useful to be able to perform logic operations in software. **For example**,

1's complement:

NOT dst

NOT dst complements all bits contained in the destination operand, changing 0's to 1 and 1's to 0.

2's complement:

The following two instructions calculates 2's complement of a number

NOT R0
ADD #1, R0

Many computers have a single instruction for 2's complement..i.e. NEGATE R0
Logic Instructions AND, OR, and NOT represented as bits in a table as follows:

AND table :

Operand 1	Operand 2	AND
0	0	0
0	1	0
1	0	0
1	1	1

OR table

Operand 1	Operand 2	OR
0	0	0
0	1	1
1	0	1
1	1	1

NOT table

Operand	NOT
0	1
1	0

Shift and Rotate Instructions: There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. There are two types of shift instructions.

1. **Logical Shift Instructions(**(LShiftL, LShiftR)
2. **Arithmetic Shift Instructions**

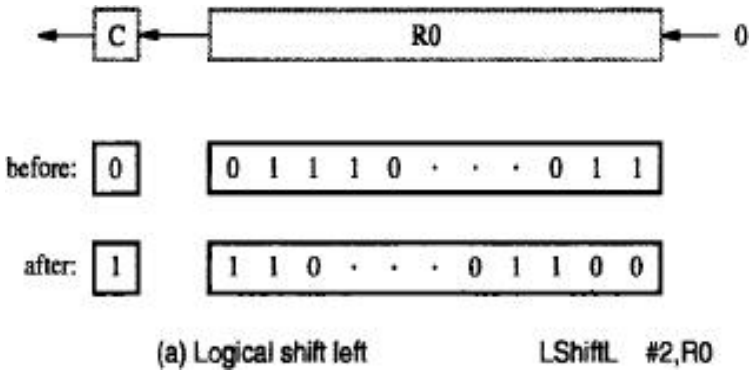
1. **Logical Shift Instructions**: These instructions shift an operand over a number of bit positions specified in a count operand in the instruction.There are two types of Logical Shift Instructions: Logical left shift (LShiftL) and Logical right shift ((LShiftR)

Logical left shift (LShiftL): The general form of Logical Shift Left Instruction is

LShiftL count, DST

The count operand may be an immediate operand or it may be contained in a processor register. For example,

LShiftL #2, R0, this is represented as



(a) Logical shift left LShiftL #2,R0

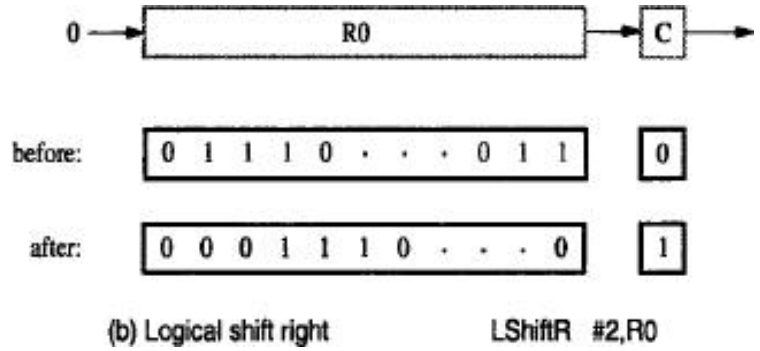
It shifts the contents of register R0 left by two positions. Vacated positions are filled with 0's.

1. **Logical right shift (LShiftR)**: The general form of Logical Shift Right Instruction is

LShiftR count, DST

The count operand may be an immediate operand or it may be contained in a processor register. For example,

LShiftR #2, R0, this is represented as



(b) Logical shift right LShiftR #2,R0

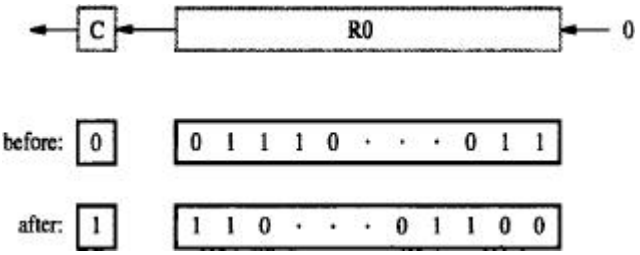
It shifts the contents of register R0 right by two positions. Vacated positions are filled with 0's.

2. Arithmetic Shift Instructions: There are two types of Arithmetic Shift Instructions:

1. **Arithmetic Shift Left (AShiftL):** A Left Arithmetic shifts a binary number by specified number of positions towards left. The vacated positions are filled with 0's. It can be represented as

AShiftL count, DST

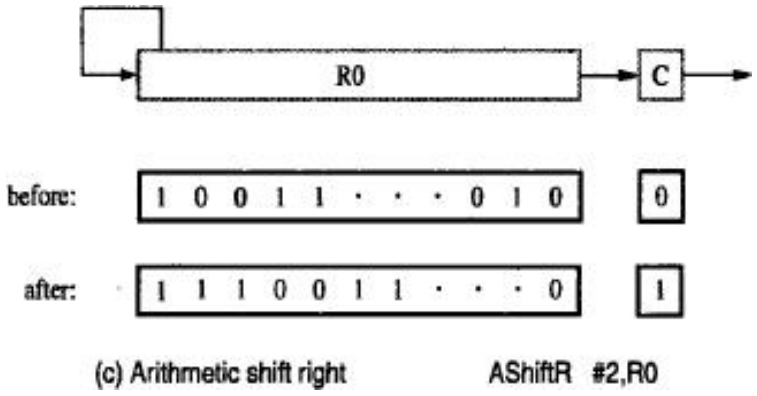
For example, AShiftL #2, R0 shifts contents of R0 by 2 positions and vacated positions are filled with zeros as shown below.



2. **Arithmetic Shift Right (AShiftR):** A Right Arithmetic shifts a binary number by specified number of positions towards right. The vacated positions are filled with copies of the original MSB bit. It can be represented as

AShiftR count, DST

For example, AShiftR #2, R0 shifts contents of R0 by 2 positions and vacated positions are filled with copies of the original MSB bit as shown below.

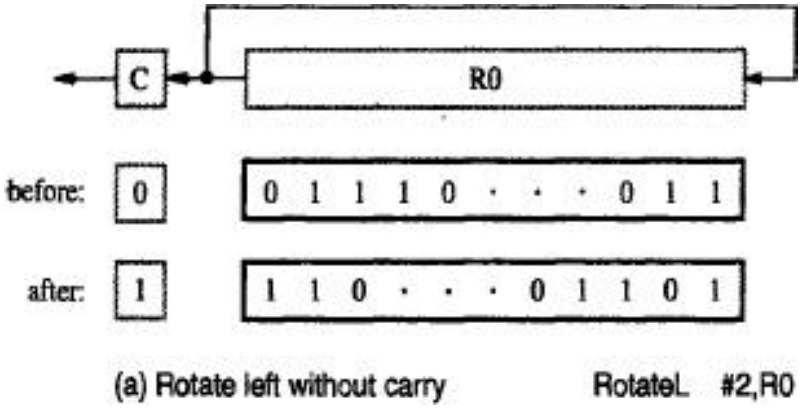


(c) Arithmetic shift right AShiftR #2,R0

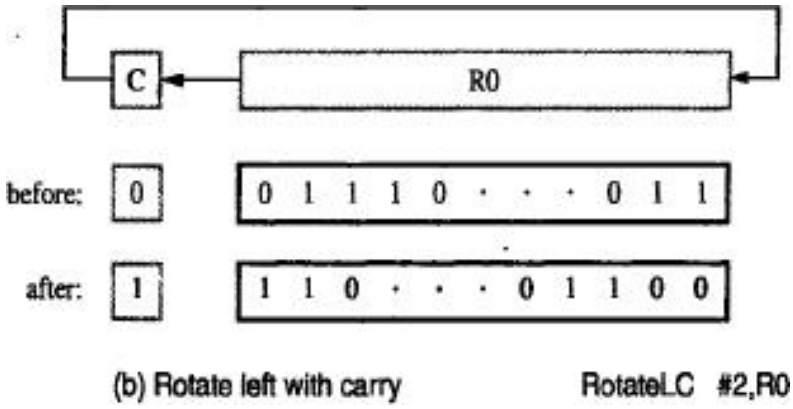
Rotate Instructions: In shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is in the carry flag C. To preserve all bits, a set of rotate instructions can be used. They move the bits that are shifted out of one end of the operand back into the other end. Different rotate instructions are

1. **Rotate Left without Carry**
2. **Rotate Left with Carry**
3. **Rotate Right without Carry**
4. **Rotate Right with Carry**

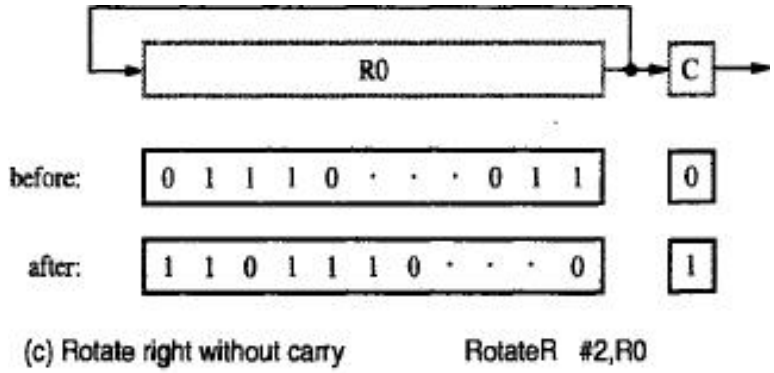
They are represented as follows.



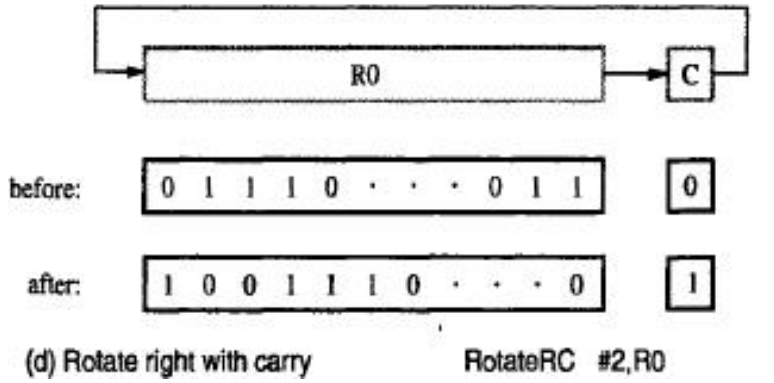
(a) Rotate left without carry RotateL #2,R0



(b) Rotate left with carry RotateLC #2,R0



(c) Rotate right without carry RotateR #2,R0



(d) Rotate right with carry RotateRC #2,R0

Digit-Packing Example:

Consider a task which uses both shift and logic operations. Suppose two decimal digits represented in ASCII code are located in memory at byte locations LOC and LOC + 1. Our goal is to represent each of these digits in the 4-bit BCD code and store both of them in a single byte location PACKED. The result is said to be in packed-BCD format.

To do this, extract the lower order 4 bits in LOC and LOC + 1 and concatenate them into the single byte at PACKED as shown below.

```
Move        #LOC,R0      R0 points to data.
MoveByte    (R0)+,R1     Load first byte into R1.
LShiftL     #4,R1        Shift left by 4 bit positions.
MoveByte    (R0),R2      Load second byte into R2.
And         #$F,R2       Eliminate high-order bits.
Or          R1,R2        Concatenate the BCD digits.
MoveByte    R2,PACKED    Store the result.
```

Here, R0 is a pointer to the ASCII characters in memory, and use the registers R1 and R2 to develop the BCD codes. MoveByte instruction transfers a byte between memory and a 32-bit processor register. The And instruction is used to mask out all except the four rightmost bits in R2. Immediate operand $F, interpreted as a 32-bit pattern, has 28 zero's in most significant bit positions.

Vector Dot Product Program: Let A and B be two vectors of length n. Their dot product can be defined as

$$\text{Dot Product} = \sum_{i=0}^{n-1} A(i) \times B(i)$$

The following program computes dot product and storing it in memory location DOTPROD. The first elements of each vector, A(0), and B(0), are stored at memory locations AVEC and BVEC.

```
            Move        #AVEC,R1        R1 points to vector A.
            Move        #BVEC,R2        R2 points to vector B.
            Move        N,R3            R3 serves as a counter.
            Clear       R0              R0 accumulates the dot product.
LOOP        Move        (R1)+,R4        Compute the product of
            Multiply    (R2)+,R4          next components.
            Add         R4,R0           Add to previous sum.
            Decrement   R3              Decrement the counter.
            Branch>0    LOOP            Loop again if not done.
            Move        R0,DOTPROD      Store dot product in memory.
```

Figure 2.33 A program for computing the dot product of two vectors.

Byte Sorting Program: Consider a program for sorting a list of bytes stored in memory into ascending alphabetic order. Assume that the list consists of n bytes. Let the list stored in memory locations LIST through LIST + n – 1, and let n be a 32-bit value stored at address N. To do this, use straight selection sort algorithm. First, the largest number is found and placed at the end of the list in location LIST + n – 1. Then the largest number in the remaining sub list of n – 1 numbers is placed at the end of the sub list in location LIST + n – 2. The procedure is repeated until the list is sorted.

```
            Move        #LIST,R0        Load LIST into base register R0.
            Move        N,R1            Initialize outer loop index
            Subtract    #1,R1             register R1 to j = n − 1.
OUTER       Move        R1,R2           Initialize inner loop index
            Subtract    #1,R1             register R2 to k = j − 1.
            MoveByte    (R0,R1),R3      Load LIST(j) into R3, which holds
                                          current maximum in sublist.
INNER       CompareByte R3,(R0,R2)      If LIST(k) ≤ [R3],
            Branch≤0    NEXT              do not exchange.
            MoveByte    (R0,R2),R4      Otherwise, exchange LIST(k)
            MoveByte    R3,(R0,R2)        with LIST(j) and load
            MoveByte    R4,(R0,R1)        new maximum into R3.
            MoveByte    R4,R3             Register R4 serves as TEMP.
NEXT        Decrement   R2              Decrement index registers R2 and
            Branch≥0    INNER             R1, which also serve as
            Decrement   R1                as loop counters, and branch
            Branch>0    OUTER             back if loops not finished.
```

(b) Assembly language program for sorting

Linked Lists: Suppose to maintain list of student records in consecutive memory locations in increasing order of student ID numbers, use a data structure called Linked List as shown below. To insert a record between i and i+1, the link address in record i is coped into the link field in the new record and then the address of the new record is written into the link field of record i. To delete record i, the address in its link field is copied into the link field of record i-1.

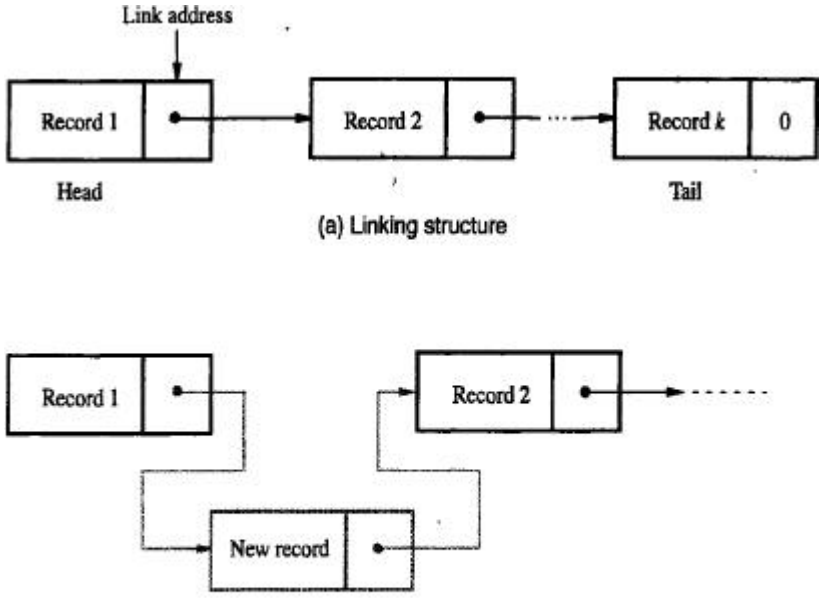


Figure 2.35 Linked-list data structure.

A subroutine for performing insertion and deletion are shown below.

```
          INSERTION   Compare     #0, RHEAD
                      Branch>0    HEAD
                      Move        RNEWREC, RHEAD      new record
not empty                                             becomes a
                      Return                          one-entry lis
          HEAD        Compare     (RHEAD), (RNEWREC)
                      Branch>0    SEARCH
                      Move        RHEAD, 4(RNEWREC)   new record
insert new record                                     becomes
somewhere after       Move        RNEWREC, RHEAD      new head
current head
                      Return
          SEARCH      Move        RHEAD, RCURRENT
          LOOP        Move        4(RCURRENT), RNEXT
                      Compare     #0, RNEXT
                      Branch=0    TAIL
new record becomes new tail
                      Compare     (RNEXT), (RNEWREC)
                      Branch<0    INSERT
insert new record in  Move        RNEXT, RCURRENT
an interior position  Branch      LOOP
          INSERT      Move        RNEXT, 4(RNEWREC)
          TAIL        Move        RNEWREC, 4(RCURRENT)
                      Return
```

Figure 2.37 A subroutine for inserting a new record into a linked list.

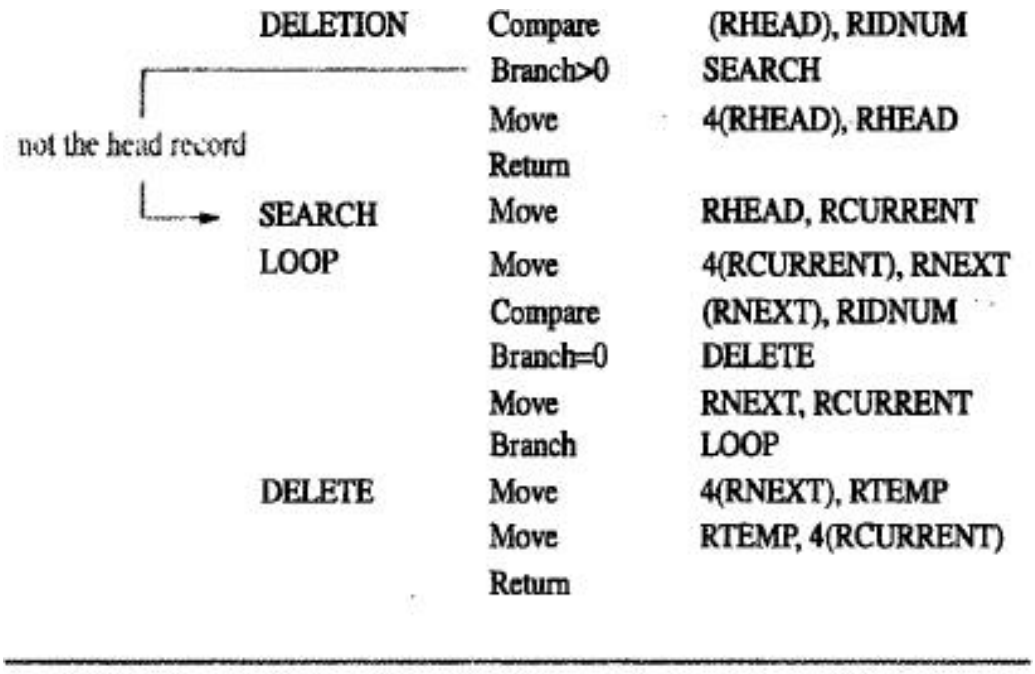


Figure 2.38 A subroutine for deleting a record from a linked list.

FREQUENTLY ASKED QUESTIONS

1. What is a register transfer language?

2. Differentiate the instruction execution for adding 'n' numbers using Straight line sequencing and branching.

3. Write short notes on shift and rotate instructions.

4. Write about various means by which data are transferred between memory of a computer and outside world.

5. What is register transfer notation? Write and explain these notations to three-address, two-address, single address and zero-address instruction types.

6. In how many ways the location of an operand is specified in an instruction? Explain each mode with suitable examples.

7. What are different addressing modes? Explain.

8. Give example for left and right shift operations.

9. List basic input and output operations.

10. Write short notes on additional addressing modes

11. With an example write about relative addressing.

12. Illustrate the concept of assembly directives with an assembly language program

13. Illustrate with examples rotate instruction?

14. Differentiate between shift and rotate instructions.

UNIT – 3

Types of Instructions: Arithmetic and Logic Instructions, Branch Instructions, Addressing modes, Input / Output Instructions.

Arithmetic and Logic Instructions: The ARM instruction set has a number of instructions for **arithmetic operations** on operands that are either contained in the general-purpose registers or given as immediate operands in the instruction itself. The basic assembly-language format for arithmetic and logic instructions is shown below.

opcode R*d*, R*n*, R*m*

where the operation specified by the opcode is performed on the source operands in general-purpose registers R*n* and R*m*. The result is placed in destination register R*d*.

Arithmetic Instructions: Different types of Arithmetic instructions are **ADD, SUB, MUL and MLA**. For example, the instruction

ADD R0, R2, R4

performs the operation

R0←[R2] + [R4]

The instruction

SUB R0, R6, R5

performs the operation

R0←[R6] − [R5]

The second source operand can be specified in the immediate mode. Thus,

ADD R0, R3, #17

performs the operation

R0←[R3] + 17

The immediate operand is an 8-bit value contained in bits *b*7−0 of the encoded machine instruction. It is an unsigned number in the range 0 to 255. The assembly language allows negative values to be used as immediate operands. If the instruction

ADD R0, R3, #−17

is used in a program, the assembler replaces it with the instruction

SUB R0, R3, #17

When the second source operand is specified as the contents of a register, they can be shifted or rotated before being used in the operation. Logical shift left (LSL), logical shift right (LSR), arithmetic shift right (ASR), and rotate right (ROR). For example, the instruction

ADD R0, R1, R5, LSL #4

is executed as follows. The second source operand, which is contained in register R5, **is shifted left 4 bit positions (equivalent to [R5] × 16),** then added to the contents of register R1. The sum is placed in register R0. The carry bit, C, is not involved in these operations. The shift or rotation amount can also be specified as the contents of a fourth register.

Two basic versions of a multiply instruction are provided. The first version multiplies the contents of two registers and places the low-order 32-bits of the product in a third register. The high-order bits of the product are discarded. If the operands are 2's-complement numbers, and if their product can be represented in 32 bits, then the retained low-order 32 bits of the product represent the correct result. For example, the instruction

MUL R0, R1, R2

Performs the operation

R0←[R1] × [R2]

The second version of the basic Multiply instruction specifies a fourth register whose contents are added to the product before the result is stored in the destination register. Hence, the instruction

MLA R0, R1, R2, R3

Performs the operation

R0←([R1] × [R2]) + [R3]

This is called a Multiply-Accumulate operation. It is often used in signal-processing applications.

Logic Instructions: Different types of Logic Instructions are **AND, OR, XOR, BIC and MVC.** For example, **The AND instruction**

AND R*d*, R*n*, R*m*

performs a bitwise logical AND of the operands in registers R*n* and R*m* and places the result in register R*d*. For example, if register **R0 contains the hexadecimal pattern 02FA62CA** and **R1 contains the pattern 0000FFFF,** then the instruction

AND R0, R0, R1 (R0 ← [R0] ^ [R1])

will **result in the pattern 000062CA** being placed in register R0.

The Bit Clear instruction, BIC, is closely related to the AND instruction. It complements each bit in operand R*m* before ANDing them with the bits in register R*n*. Using the same R0 and R1 bit patterns as in the above example, the instruction

BIC R0, R0, R1 (R0 ← [R0] ^ [NOT R1])

results in the pattern 02FA0000 being placed in R0.

The OR instruction

OR R0,R0,R1 (R0 ← [R0] V [R1])

Performs OR operation between contents of R0, R1 registers.

The XOR instruction

XOR R0,R0,R1 (R0 ← [R0]◯[R1])

Performs XOR operation between contents of R0, R1 registers.

The Move Negation instruction, with the opcode Mnemonic MVN, complements the bits of the source operand and places the result in Rd. for example,

MVN R0, R3.

If the contents of R3 are the hexadecimal pattern 0F0F0F0F, then it places the result F0F0F0F0 in the register R0.

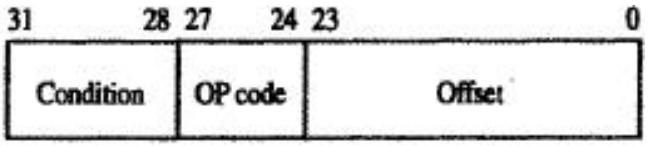
The following ARM program merges two BCD digits into a byte.

```
LDR     R0,POINTER      Load address LOC into R0.
LDRB    R1,[R0]         Load ASCII characters
LDRB    R2,[R0,#1]        into R1 and R2.
AND     R2,R2,#&F       Clear high-order 28 bits of R2.
ORR     R2,R2,R1,LSL #4 Or [R1] shifted left into [R2].
STRB    R2,PACKED       Store packed BCD digits
                          into PACKED.
```

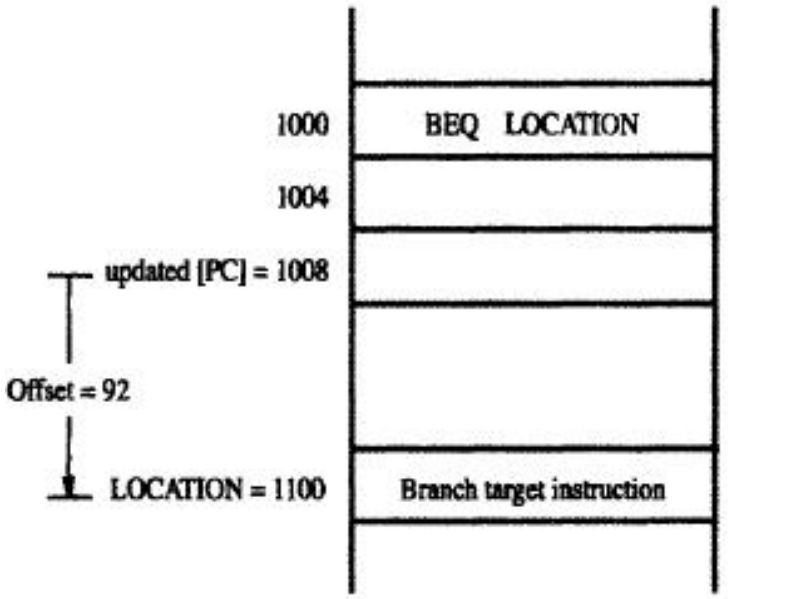
Figure 3.5 An ARM program for packing two 4-bit decimal digits into a byte.

The first instruction in the program loads the address LOC into register R0. The two ASCII characters containing the BCD digits in their low-order four bits are loaded into the low-order byte positions of registers R1 and R2 by the next two Load instructions. The AND instruction clears the high-order 2 bits of R2 to zero, leaving the second BCD digit in the four low-order bit positions. The '&' character in this instruction signifies hexadecimal notation for the immediate value. The ORR instruction then shifts the first BCD digit in R1 to the left four positions and places it to the left of the second BCD digit in R2. The two digits packed into the low-order byte of R2 are then stored into location PACKED.

Branch Instructions: Conditional branch instructions contain a signed, 2's complement, 24-bit offset that is added to the updated contents of the program counter to generate the branch target address. The format for the branch instruction is shown below.



(a) Instruction format



(b) Determination of a branch target address

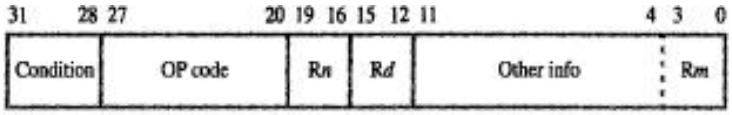


Figure 3.2 ARM instruction format.

The BEQ instruction (Branch if Equal to 0) causes a branch if the Z flag is set to 1. The higher order 4 bits b_{31-28}, of the instruction word determines whether or not branching should takes place. At the time the branch target address is computed, the contents of the PC will have been updated to contain the address of the instruction that is two words beyond the Branch instruction itself. This is due to pipelined instruction execution. If the Branch instruction is at address location 1000 and the branch target address is 1100, as shown in Figure D.6, then the offset is 92, because the contents of the updated PC will be 1000 + 8 = 1008 when the branch target address 1100 is computed.

Setting condition codes: The Compare and Test instructions always update the condition code flags. Some instructions, such as compare, given by CMP R*n*, R*m* performs the operation [R*n*] − [R*m*]. The arithmetic, logic, and Move instructions affect the condition code flags only if explicitly specified to do so by a bit in the OP-code field. This is indicated by appending the suffix S to the assembly language OP-code mnemonic. For example, the instruction ADDS R0, R1, R2 sets the condition code flags, but ADD R0, R1, R2 does not.

The following is an ARM program for adding n numbers. Here, Location N contains the number of entries in the list, and location SUM is used to store the sum. The Load and Store operations performed by

the first and last instructions use the Relative addressing mode. This assumes that the memory locations N and SUM are within the range reachable by offsets relative to the PC. The address NUM1 of the first of the numbers to be added is loaded into register R2 by the second instruction. The Post-indexed addressing mode, which includes writeback, is used in the first instruction of the loop.

```
        LDR   R1,N          Load count into R1.
        LDR   R2,POINTER    Load address NUM1 into R2.
        MOV   R0,#0         Clear accumulator R0.
LOOP    LDR   R3,[R2],#4    Load next number into R3.
        ADD   R0,R0,R3      Add number into R0.
        SUBS  R1,R1,#1      Decrement loop counter R1.
        BGT   LOOP          Branch back if not done.
        STR   R0,SUM        Store sum.
```

Figure 3.7 An ARM program for adding numbers.

Input / Output Instructions:

Input / Output Instructions: The ARM architecture uses memory mapped I/O. Input / Output operations are essential which has a significant effect on performance of a computer. An I/O device is connected to the interconnection network by using a circuit, called the *device interface*, which provides the means for data transfer and for the exchange of status and control information needed to facilitate the data transfers and govern the operation of the device. The interface includes some registers that can be accessed by the processor. One register may serve as a buffer for data transfers, another may hold information about the current status of the device, and yet another may store the information that controls the operational behavior of the device. These *data*, *status*, and *control* registers are accessed by program instructions as if they were memory locations.

Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as Program controlled I/O. A solution to this problem is as follows: On output, the processor sends the first character and then waits for a signal from the display that the character has been received. It then sends the second character, and so on. Input is sent from the keyboard in a similar way; the processor waits for a signal indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard. Then the processor proceeds to read that code.

The keyboard and the display are separate devices as shown below.

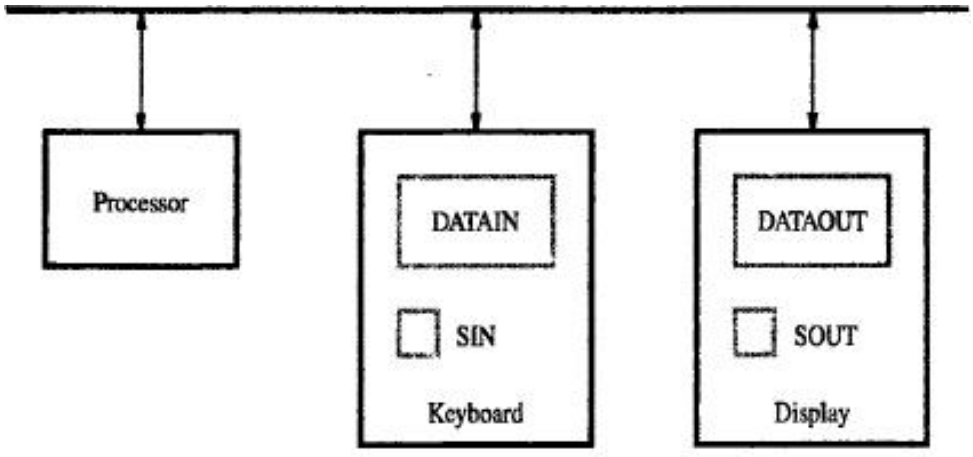


Figure 2.19 Bus connection for processor, keyboard, and display.

Consider the problem of moving a character from the keyboard to the processor. Striking a key stores the corresponding character code in an 8-bit buffer register DATAIN which is associated with the keyboard. To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1 (initially SIN=0). A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN. When the character is transferred to the processor, SIN is again set to 1 and the process repeats.

To transfer the characters from processor to display, a buffer register, DATAOUT, and a status control flag, SOUT, are used. When SOUT equals 1, the display is ready to receive a character. Under program control, the processor monitors SOUT, and when SOUT is set to 1, the processor transfers a character code to DATAOUT clears SOUT to 0; when the display device is ready to receive a second character, SOUT is again set to 1.

Suppose that bit 3 in each of the device status registers INSTTUS and OUTSTATUS contains the respective control flags SIN and SOUT. Also assume that the keyboard DATAIN and display DATAOUT registers are located at addresses INSTATUS + 4 and OUTSTATUS + 4. The READ and WRITE wait loops can be implemented as follows.

```
READWAIT   LDR    R3,[R1]
           TST    R3,#8
           BEQ    READWAIT
           LDRB   R3,[R1,#4]
WRITEWAIT  LDR    R4,[R2]
           TST    R4,#8
           BEQ    WRITEWAIT
           STRB   R3,[R2,#4]
```

Assume that the address INSTATUS has been loaded into register R1. The instruction sequence reads a character into register R3 when a key has been pressed on the keyboard. The test (TST) instruction performs the bitwise logical AND operation on its two operands and sets the condition code flags based on the result. The immediate operand 8 (0000 1000) has a single 1 in the bit 3 position. Therefore, the result of the TST operation will be zero if bit 3 of INSTATUS is zero and will be non zero if bit 3 is one, signifying that a character is available in DATAIN. The BEQ instruction branches back to READWAIT if the result is zero.

Assuming that the address OUTSTATUS has been loaded into register R2, the instruction sequence sends the character in register R3 to the DATAOUT register when display is ready to receive it.

The following ARM program reads line of characters.

```
READ   LDR    R3,[R1]      Load [INSTATUS] and
       TST    R3,#8          wait for character.
       BEQ    READ
       LDRB   R3,[R1,#4]   Read the character and
       STRB   R3,[R0],#1     store it in memory.
ECHO   LDR    R4,[R2]      Load [OUTSTATUS] and
       TST    R4,#8          wait for display
       BEQ    ECHO           to be ready.
       STRB   R3,[R2,#4]   Send character to display.
       TEQ    R3,#CR       If not carriage return,
       BNE    READ           read more characters.
```

Figure 3.9 An ARM program that reads a line of characters and displays it.

M.JOSHNA

ARM Addressing Modes:

Table D.1 ARM indexed addressing modes.

Name	Assembler syntax	Addressing function
With immediate offset:		
Pre-indexed	[R*n*, #offset]	EA = [R*n*] + offset
Pre-indexed with writeback	[R*n*, #offset]!	EA = [R*n*] + offset; R*n* ← [R*n*] + offset
Post-indexed	[R*n*], #offset	EA = [R*n*]; R*n* ← [R*n*] + offset
With offset magnitude in R*m*:		
Pre-indexed	[R*n*, ± R*m*, shift]	EA = [R*n*] ± [R*m*] shifted
Pre-indexed with writeback	[R*n*, ± R*m*, shift]!	EA = [R*n*] ± [R*m*] shifted; R*n* ← [R*n*] ± [R*m*] shifted
Post-indexed	[R*n*], ± R*m*, shift	EA = [R*n*]; R*n* ← [R*n*] ± [R*m*] shifted
Relative (Pre-indexed with immediate offset)	Location	EA = Location = [PC] + offset

EA = effective address
offset = a signed number contained in the instruction
shift = direction #integer
where direction is LSL for left shift or LSR for right shift; and
integer is a 5-bit unsigned number specifying the shift amount
±R*m* = the offset magnitude in register R*m* can be added to or subtracted from the contents of base register R*n*

The basic method for addressing memory operands is an indexed addressing mode, defined as

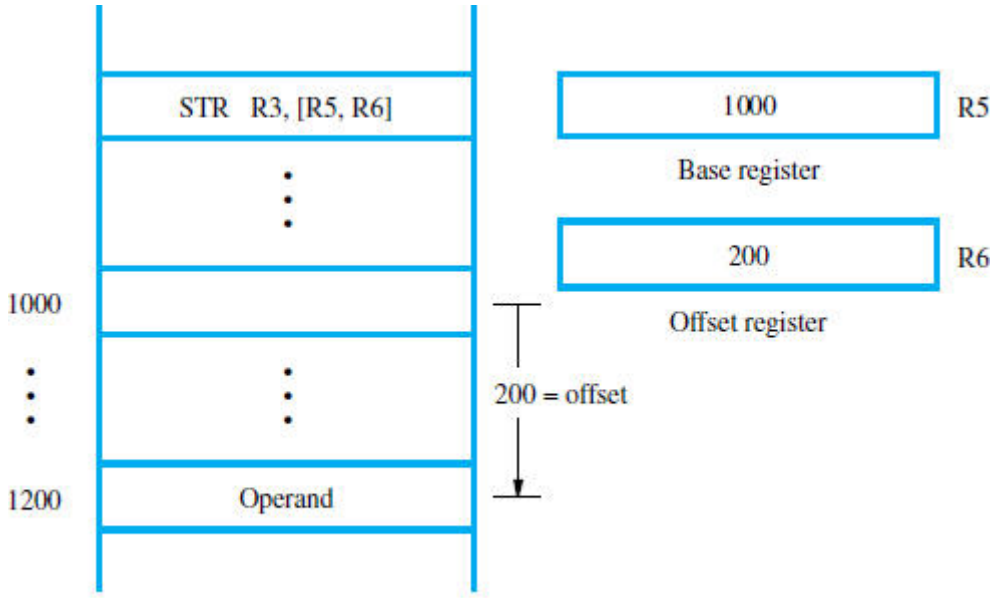
1. *Pre-indexed mode:* The effective address of the operand is the **sum of the contents of a base register, R*n*, and a signed offset.** For example, the Load instruction

   LDR R*d*, [R*n*, #offset]

   specifies the offset (expressed as a signed number) in the immediate mode and performs the operation

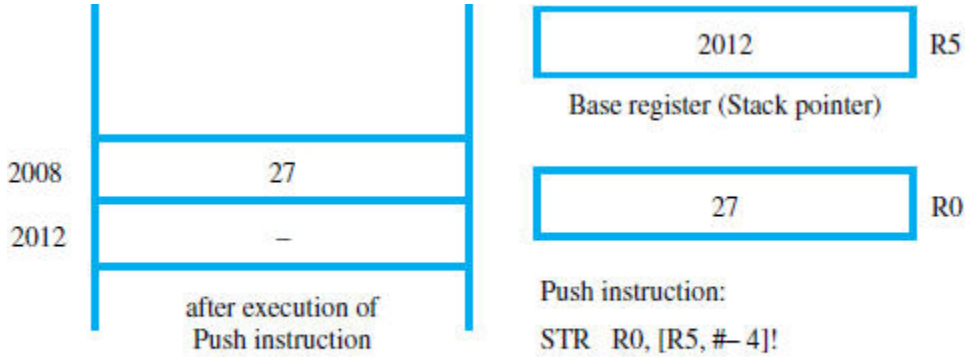
   R*d* ← [R*n*] + offset

   The following shows an example of the Pre-indexed mode with the offset contained in register R6 and the base value contained in R5. The Store instruction (STR) stores the contents of R3 into the word at memory location 1200.



(b) Pre-indexed addressing mode

2. *Pre-indexed with write back mode:* The effective address of the operand is generated in the same way as in the Pre-indexed mode, then the effective address is written back into R*n*. The exclamation mark signifies write-back in pre-indexed addressing mode.



(b) Pre-indexed addressing with writeback

Above shows an example of pushing the contents of register R0, which are 27, onto a programmer-defined stack. Register R5 is used as the stack pointer. Initially, it contains the address 2012 of the current TOS (top-of-stack) element. The Pre-indexed addressing mode with write back can be used to perform the Push operation with the instruction

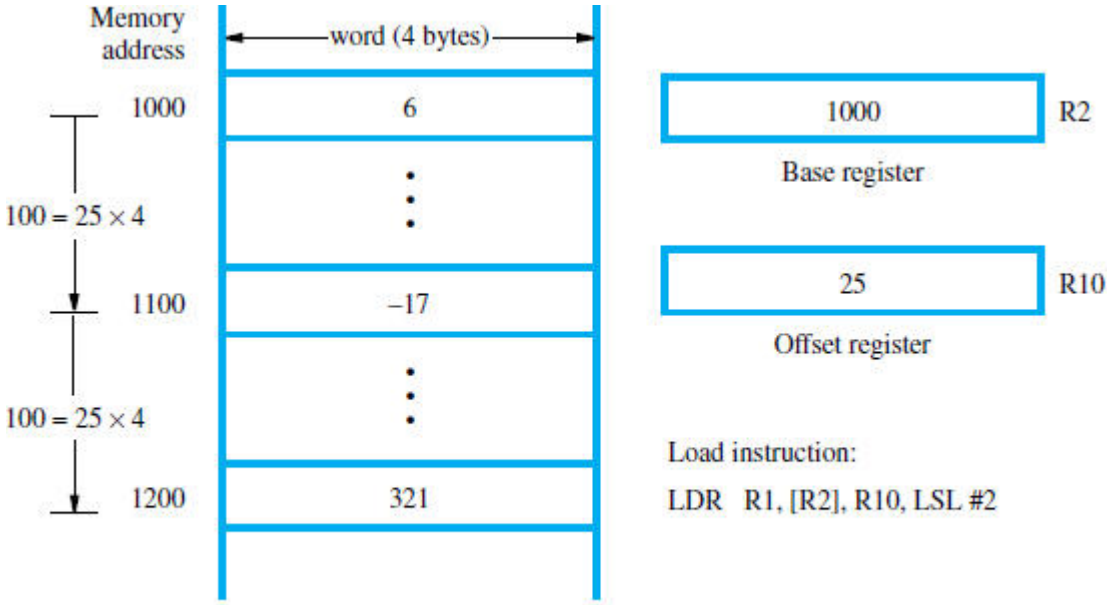
STR R0, [R5, #–4]!

The immediate offset −4 is added to the contents of R5 and the new value is written back into R5. Then, this address value of the new top of the stack, 2008, is used as the effective address for the Store operation. The contents of register R0 are then stored at this location.

3. ***Post-indexed mode*—The effective address of the operand is the contents of R*n*. The offset is then added to this address and the result is written back into R*n*. The post-indexed mode always involves write back, so the exclamation is not needed.**

pre- and post-indexing are distinguished by the way the square brackets are used. When only the base register is enclosed in square brackets, its contents are used as the effective address. The offset is added to the register contents after the operand is accessed. In other words, post-indexing is specified. When both the base register and the offset are placed inside the square brackets, their sum

is used as the effective address of the operand, that is, pre-indexing is used. If writeback is to be performed, it must be indicated by the exclamation character.



(a) Post-indexed addressing

The first time that the Load instruction is executed, the effective address is [R2] = 1000. Therefore, the number 6 at this address is loaded into R1. Then, the write back operation changes the contents of R2 from 1000 to 1100 so that it points to the second number, −17. It does this by shifting the contents, 25, of the offset register R10 left by two bit positions and then adding the shifted value to the contents of R2. The contents of R10 are not changed in this process. The left shift is equivalent to multiplying 25 by 4, generating the required offset of 100. When the Load instruction is executed on the second pass through the loop, the second number, −17, is loaded into R1. The third number, 321, is loaded into R1 on the third pass, and so on.

In all three indexed addressing modes, the offset may be given as an immediate value in the Range ±4095. Alternatively, the magnitude of the offset may be specified as the contents of the R*m* register, with the sign (direction) of the offset specified by a ± prefix on the register name. For example, the instruction

LDR R0, [R1, −R2]!

performs the operation

R0←[[R1] − [R2]]

The effective address of the operand, [R1]−[R2], is then loaded into R1 because write back is specified.

When the offset is given in a register, it may be scaled by a power of 2 before it is used by shifting it to the right or to the left. This is indicated in assembly language by placing the shift direction (LSL for left shift or LSR for right shift) and the shift amount after the register name, R*m*. For example, the contents of R2 in the example above may be multiplied by 16 before being used as an offset by modifying the instruction as follows:

LDR R0, [R1, −R2, LSL #4]!

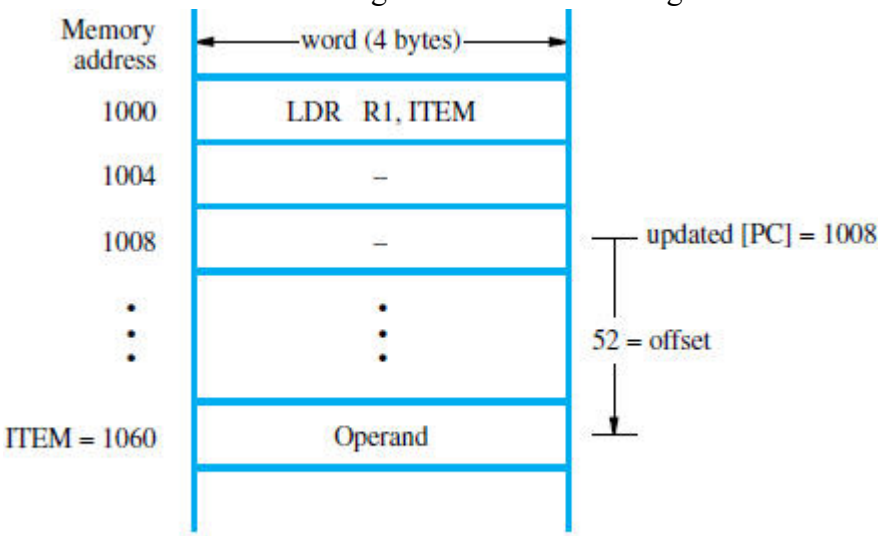
This instruction performs the operation

R0 ← [[R1] −16 × [R2]]

and then loads the effective address into R1 because write back is specified.

4.*Relative Addressing:* The program counter PC, may be used as the base register R_n. In this case, the relative addressing mode is used. The assembler determines the immediate offset as the signed distance

between the address of the operand and the contents of the updated PC. When the effective address is calculated at instruction execution time, the contents of PC will have been updated to the address two words (8 bytes) forward from the instruction containing the relative addressing mode.



(a) Relative addressing mode

The address of the operand, given symbolically in the instruction as ITEM, is 1060. There is no Absolute addressing mode available in the ARM architecture. Therefore, when the address of a memory location is specified by placing an address label in the operand field, the assembler uses the Relative addressing mode. This is implemented by the Pre-indexed mode with an immediate offset, using PC as the base register. As shown in the figure, the offset calculated by the assembler is 52, because the updated PC will contain 1008 when the offset is added to it during program execution. The effective address generated by this instruction is 1060 = 1008 + 52. The operand must be within a distance of 4095 bytes forward or backward from the updated PC. If the operand address is outside this range, an error is indicated by the assembler and a different addressing mode must be used to access the operand.

68000 Addressing Modes: The 68000 has several addressing modes which are defined as follows.

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Absolute Short	Value	EA = Sign Extended WValue
Absolute Long	Value	EA = Value
Register	Rn	EA = R_n that is, Operand = [R_n]
Register Indirect	(An)	EA = [A_n]
Autoincrement	(An)+	EA = [A_n]; Increment A_n
Autodecrement	–(An)	Decrement A_n; EA = [A_n]
Basic index	WValue(An)	EA = WValue + [A_n]
Full index	BValue(An, Rk)	EA = BValue + [A_n] +[R_k]
Basic relative	WValue(PC)	EA = WValue + [PC]
Full relative	BValue(PC, Rk)	EA = BValue + [PC] + [R_k]

M.JOSHNA

1. *Immediate mode*: The operand is a constant value that is contained within the instruction. Four sizes of immediate operands can be specified. Small 3-bit numbers can be included in the OP-code word of certain instructions. Byte, word, and long word operands are found in one or two extension words that follow the OP-code word.
2. *Absolute mode*: The memory address of an operand is given in the instruction immediately after the OP-code word. There are two versions of this mode—long and short. In the long mode, a full 24-bit address is specified in two extension words. In the short mode, a 16-bit value is given in one extension word.
3. *Register mode:* The operand is in a processor register, A*n* or D*n*, that is specified in the instruction.
4. *Register indirect mode:* The effective address of the operand is in an address register, A*n*, that is specified in the instruction.
5. *Auto increment mode:* The effective address of the operand is in an address register, A*n*, that is specified in the instruction. After the operand is accessed, the contents of A*n* are incremented by 1, 2, or 4, depending on whether the operand is a byte, a word, or a long word.
6. *Auto decrement mode:* The contents of an address register, A*n*, that is specified in the instruction are first decremented by 1, 2, or 4, depending on whether the operand is a byte, a word, or a long word. The effective address of the operand is then given by the decremented contents of A*n*.
7. *Basic index mode:* A 16-bit signed offset and an address register, A*n*, are specified in the instruction. The offset is sign-extended to 32 bits, and the sum of the sign-extended offset and the 32-bit contents of A*n* is the effective address of the operand.
8. *Full index mode:* An 8-bit signed offset, an address register A*n*, and an index register R*k* (either an address or a data register) are given in the instruction. The effective address of the operand is the sum of the sign-extended offset, the contents of register A*n*, and the signed contents of register R*k*.
9. *Basic relative mode:* This mode is the same as the Basic index mode, except that the program counter (PC) is used instead of an address register, A*n*.
10. *Full relative mode:* This mode is the same as the Full index mode, except that the program counter (PC) is used instead of an address register, A*n*.

IA-32 Addressing Modes: The IA-32 architecture has a large and flexible set of addressing modes which are defined as follows.

Name	Assembler syntax	Addressing function
Immediate	Value	Operand = Value
Direct	Location	EA = Location
Register	Reg	EA = Reg that is, Operand = [Reg]
Register indirect	[Reg]	EA = [Reg]
Base with displacement	[Reg + Disp]	EA = [Reg] + Disp
Index with displacement	[Reg * S + Disp]	EA = [Reg] × S + Disp
Base with index	[Reg1 + Reg2 * S]	EA = [Reg1] + [Reg2] × S
Base with index and displacement	[Reg1 + Reg2 * S + Disp]	EA = [Reg1] + [Reg2] × S + Disp

Value = an 8- or 32-bit signed number
Location = a 32-bit address
Reg, Reg1, Reg2 = one of the general purpose registers EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, with the exception that ESP cannot be used as an index register.
Disp = an 8- or 32-bit signed number, except that in the Index with displacement mode it can only be 32 bits.
S = a scale factor of 1, 2, 4, or 8

1. *Immediate mode:* The operand is contained in the instruction. It is a signed 8-bit or 32-bit number, with the length being specified by a bit in the opcode of the instruction. This bit is 0 for the short version and 1 for the long version.
2. *Direct mode:* The memory address of the operand is given by a 32-bit value in the instruction.
3. *Register mode:* The operand is contained in one of the eight general-purpose registers specified in the instruction.
4. *Register indirect mode:* The memory address of the operand is contained in one of the eight general- purpose registers specified in the instruction.
5. *Base with displacement mode:* An 8-bit or 32-bit signed displacement and one of the eight general- purpose registers to be used as a base register are specified in the instruction. The effective address of the operand is the sum of the contents of the base register and the displacement.
6. *Index with displacement mode:* A 32-bit signed displacement, one of the eight general purpose registers to be used as an index register, and a scale factor of 1, 2, 4, or 8, are specified in the instruction. To obtain the effective address of the operand, the contents of the index register are multiplied by the scale factor and then added to the displacement.
7. *Base with index mode:* Two of the eight general-purpose registers and a scale factor of 1, 2, 4, or 8, are specified in the instruction. The registers are used as base and index registers. The effective address of the operand is determined by first multiplying the contents of the index register by the scale factor and then adding the result to the contents of the base register.
8. *Base with index and displacement mode:* An 8-bit or 32-bit signed displacement, two of the eight general-purpose registers, and a scale factor of 1, 2, 4, or 8, are specified in the instruction. The registers are used as base and index registers. The effective address of the operand is determined by first multiplying the contents of the index register by the scale factor and then adding the result to the contents of the base register and the displacement.

M.JOSHNA

UNIT-2

Concept of Boolean Algebra

Introduction:

→ In 1854 George Boolean introduced a systematic treatment of logic and developed for this purpose of an algebric system Now called Boolean Algebra.

→ In 1938 C.E Shanon introduced a two-valued Boolean Algebra called a switching Algebra

Fundamental Postulateds of Boolean Algebra

S.NO	Postulates	Comments
1.	→ Result of each operator is either 0 or 1	$1, 0 \in B$
2.	a) $0+0=0$, $0+1=1+0=1$ b) $1\cdot1=1$; $0\cdot1=1\cdot0=0$	Identity element "0" for "+" and "1" for "·"
3.	a) $(A+B)=(B+A)$ b) $(A\cdot B)=(B\cdot A)$	Commulative law
4.	a) $A\cdot(B+C)=(A\cdot B)+(A\cdot C)$ b) $A+(B\cdot C)=(A+B)\cdot(A+C)$	Distributive law
5.	a) $A+\bar{A}=1$, $0+\bar{0}=1$, $1+\bar{1}=1$ b) $A\cdot\bar{A}=0$, if $\bar{0}\cdot\bar{0}=0\cdot1=0$ $1\cdot\bar{1}=1\cdot0=0$	Complement

$\bar{0}=1$
$\bar{1}=0$

M.JOSHNA

Basic theorems and properties

Duality :-

→ The principle of Duality theorem says that starting with a boolean relation, we can derive another boolean relation by

1. Changing each "OR" sign to an "AND" sign
2. Changing each "AND" sign to an "OR" sign
3. Any 0 or 1 operating in the Expression of boolean function.

Ex :- Dual of relation $A+\bar{A}=1$ is $A\cdot\bar{A}=0$

$A+\bar{A}=1$ $\quad$ $A\cdot\bar{A}=0$

$A\cdot\bar{A}=0$

principal of Duality theorem says that starting with a

~~Theorems~~ :-

Laws of Boolean algebra :-

Boolean Expression	B.A law (or) Rule
=> $A+1=1$ $A\cdot 0=0$	Annulment
=> $A+0=A$ $A\cdot 1=A$	Identity
=> $A+A=A$ $A\cdot A=A$	Idempotent
=> $\bar{\bar{A}}=A$ Not $\bar{\bar{A}}=A$	Double Negation

Boolean Expression	BA law (or) Rule
$\Rightarrow A+\bar{A}=1$ $A\cdot\bar{A}=0$	Complement
$\Rightarrow A+B=B+A$ $A\cdot B=B\cdot A$	Commitative law
$\Rightarrow \overline{A+B}=\bar{A}\cdot\bar{B}$ $\overline{AB}=\bar{A}+\bar{B}$	De-morgan

Boolean Algebra functions :-

Function	Description	Expression
1.	NULL	0
2.	Identity	1
3.	INput A	A
4.	INput B	B
5.	NOT A	$\bar{A}$
6.	NOT B	$\bar{B}$
7.	A AND B (AND)	$A\cdot B$
8.	A AND NOT B	$A\cdot\bar{B}$
9.	NOTA ANDB	$\bar{A}\cdot B$
10.	NOT AND (NAND)	$\overline{A\cdot B}$
11.	A OR B (OR)	$A+B$
12.	A OR NOT B	$A+\bar{B}$

M.JOSHNA

Function	Description	Expression
13.	NOT A OR B	$\bar{A}+B$
14.	NOT OR (NOR)	$\overline{A+B}$
15.	Exclusive OR (x-OR)	$A\bar{B}+\bar{A}B$
16.	Exclusive NOR (x-NOR)	$AB+\bar{A}\bar{B}$

Theorems:-

(1) a) $A+A=A$

b) $A \cdot A = A$

(2) a) $A+1=1$

b) $A \cdot 0 = 0$

(3) a) $\overline{\bar{A}} = A$

(4) a) $A+AB=A$

b) $A(A+B)=A$

(5) a) $A+\bar{A}B = A+B$

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

(1)(a) Proof:-

L.H.S $A+A = (A+A)\cdot(1)$

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

$= AA + A\bar{A} + AA + A\bar{A}$

$= AA + A\bar{A}$

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

$= A$ //

$A+A=A$

if $A=0$; $0+0=0$

if $A=1$; $1+1=1$

$[1 = A+\bar{A}]$

(1)(b) Proof:-

L.H.S $A \cdot A = A \cdot A + (0)$

$= A \cdot A + A \cdot \bar{A}$

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

$= A$ //

$A \cdot A = A$

$A=0 \Rightarrow 0 \cdot 0 = 0$

$A=1 \Rightarrow 1 \cdot 1 = 1$

$[A \cdot \bar{A} = 0]$

(2) (a) $A+1=1$

Proof :- $A+1 = (1)A+1$

$= (A+\bar{A})\cdot(A+1)$

$= A\cdot A + A + \bar{A}\cdot A + \bar{A}$

$= A + A + 0 + \bar{A}$

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

$= A + \bar{A}$

$= 1$

$[A\cdot\bar{A}=0$
$A+\bar{A}=1]$

$A+1=1$

if $A=0$; $0+1=1$
$A=1$; $1+1=1$

(b) $A\cdot 0 = 0$

Proof :- $A\cdot 0 = A\cdot(A\cdot\bar{A})$

$= A(0)$

$= 0$

$A\cdot 0=0$
$A=0$; $0\cdot 0=0$
$A=1$; $1\cdot 0=0$

(3) $\bar{\bar{A}} = A$

Proof :- if $A=0$; $\bar{\bar{0}} = \bar{1} = 0$

$A=1$; $\bar{1} = \bar{0} = 1$

$[\bar{0}=1$
$\bar{1}=0]$

(4) (a) $A+AB = A$

Proof :- $A+AB = A(1+B)$

$= A(1)$

$= A$

$[\because 1+B=1]$

(b) $A(A+B) = A$

Proof :- $A(A+B) = (A+AB)(A+B)$

$= AA + AB + ABA + AB\cdot B$

$= A + AB + A\cdot AB + AB$

$= A + AB + AB + AB$

$= A(1+B) + AB$

$= A + AB$

$= A(1+B)$

$= A$ //

(5)(a) $A + \bar{A}B = A + B$

proof: $A + \bar{A}B = (A + \bar{A})(A + B)$

$= A + B$ $\quad [\because A + \bar{A} = 1]$

(b) $A(\bar{A} + B) = AB$ $\quad [A \cdot \bar{A} = 0]$

proof: $A(\bar{A} + B) = A \cdot \bar{A} + AB$

$= 0 + AB$

$= AB$ //

**** De-Morgan's theorem:

(i) $\overline{AB} = \bar{A} + \bar{B}$

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

(i) $\overline{AB} = \bar{A} + \bar{B}$

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

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

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

→ Consensus theoram :-

→ In simplification of Boolean Expresion, an Expresion of the form $AB+\bar{A}C+BC$. The term BC is redented and can be elemin-ated to form the Equilent Expresion $AB+\bar{A}C$.

→ The theoram is used for this simplification is known as Consensus theoram. and it is stated as

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

Proof :- L.H.S $\Rightarrow AB+\bar{A}C+BC = AB+\bar{A}C+BC(1)$ [$\because A+\bar{A}=1$]

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

$= AB+\bar{A}C+BCA+BC\bar{A}$

[$\because 1+C=1$, $1+B=1$]

$= ABC(1+C)+\bar{A}C(1+B)$

$= AB+\bar{A}C$

* Solve the given Expresion using Consensus theorem

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

$\Rightarrow \bar{A}\bar{B}+AC+B\bar{C}+\bar{B}C(1)+AB$ [$A+\bar{A}=1$]

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

$\Rightarrow \bar{A}\bar{B}+AC+B\bar{C}+\bar{B}CA+\bar{B}C\bar{A}+AB$

$\Rightarrow \bar{A}\bar{B}(1+C)+AC(1+\bar{B})+B\bar{C}+AB$ [$1+C=1$, $1+\bar{B}=1$]

$\Rightarrow \bar{A}\bar{B}+AC+B\bar{C}+AB$

$\Rightarrow \bar{A}\bar{B}+AC+B\bar{C}+AB(1)$

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

$\Rightarrow \bar{A}\bar{B}+AC+B\bar{C}+ABC+AB\bar{C}$ [$1+B=1$, $1+A=1$]

$\Rightarrow \bar{A}\bar{B}+AC(1+B)+B\bar{C}(1+A)$

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

Dual of consensus theorem:

→ The Dual form of consensus theorem is stated as

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

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

$(0 + AC + \bar{A}B + BC)(B+C) = (0 + AC + \bar{A}B + BC)$ $[A \cdot \bar{A} = 0]$

$ABC + \bar{A}B \cdot B + BC \cdot B + AC \cdot C + \bar{A}BC + BC \cdot C = AC + \bar{A}B + BC$

$[BC + BC = BC]$

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

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

$\bar{A}B(1+C) + BC(1+A) + AC = AC + \bar{A}B + BC$

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

Boolean function (or) Switching function:

→ Boolean equations are constructed by connecting the boolean Constants and variables with the boolean operation.

→ This Boolean Expressions are known as boolean formulas we use boolean Expression to describe boolean functions.

for Example: If Boolean Expresion $(A+\bar{B})C$ is used to describe the function of f, then boolean function is written as $f(A,B,C) = (A+\bar{B})C$

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

→ Let us consider the whole four variable boolean function.

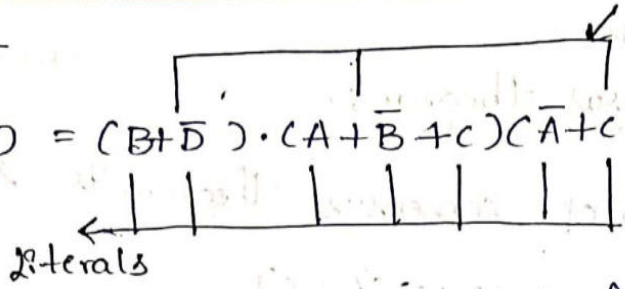
Product-terms:

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

product terms

literals

Sum - Terms :-



→ The literals and terms are arranged in the forms to in two

1. Sum of product (SOP)
2. product of sum (POS)

1. Sum of products :-

→ The sum of product is also called Disjunctional normal form (or) Disjunctional normal formula

→ the words sum & product are discribed from the symbolic representation "OR" and "AND" function (+ and *)

Ex :-1 $f(A,B,C) = A\bar{B}C + A\bar{B}\bar{C}$

sum terms

products

Ex :-2 $f(P,Q,R) = \bar{P}Q + QR + RS$

sum-terms

Products

2. Product of sum :-

→ The product of sum is also called conjunctional normal form (or) conjunctional normal formula.

→ A product of sum is any group of sum terms ANDed to gether.

Ex :- $f(A,B,C) = (A+B+C)\cdot(\bar{A}+\bar{B}+\bar{C})$

Product

Sum

Ex:- $f(P,Q,r) = (P+Q)\cdot(r+\bar{P})\cdot Q$

(+ : Sum, · : product)

*Canonical form :- (Standard sop and pos forms)

→ Basically Canonical forms are two typs

1. Standard Sop (or) minterm canonical form
2. standard pos (or) maxterm canonical form

1. standard sop :- (minterm Canonical form)

→ The ssop is given by $f(A,B,C) = (A\bar{B}C) + (ABC) + (\bar{A}BC)$

Each product term is consistent all literals in either completed form or uncompleted form

2. standard pos :- (maxterm)

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

Each sum term consists of all literals in the completed form

*Converting Expressions in standard sop (or) pos form

steps to convert sop to standard sop form :-

Steps:-

1. find the missing literal in each product term if any
2. "AND" each product term having missing literals which term 1 formed by "OR"ing the literals and its complement
3. Expand the terms by applying Distributive law and recorded the literals in the product terms
4. Reduce the expression omitting (or) remove repeated product terms if any. because $(A+A) = A$

Ex:- convert the given Expresion in standard sop form

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

Step:-1 ~~$f(A,B,C)$~~ = find the missing literal in each product

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

- A literal is missing
- c literal missing
- B literal missing

Step:-2 AND product terms with (missing literal + complement)

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

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

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

Step 3:- Expand the term and recorded the terms

Expand:-

$f(A,B,C) = ACB + AC\bar{B} + \bar{A}BC + \bar{A}B\bar{C} + BCA + BC\bar{A}$

Recorded:-

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

Step 4:- omitting the repeted product term

$f(A,B,C) = \underline{ABC} + A\bar{B}C + \underline{\bar{A}BC} + \bar{A}B\bar{C} + \underline{ABC} + \bar{A}BC$

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

Ex:- Convert the given Expresion in standard sop form

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

Step 1:- find

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

- B literal missing
- C literal missing

Step 2:- AND product term with (missing literal + complement)

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

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

M.JOSHNA

Function	Description	Expression
13.	NOT A OR B	$\bar{A}+B$
14.	NOT OR (NOR)	$\overline{A+B}$
15.	Exclusive OR (x-OR)	$A\bar{B}+\bar{A}B$
16.	Exclusive NOR (x-NOR)	$AB+\bar{A}\bar{B}$

Theorems:-

(1) a) $A+A=A$

b) $A\cdot A=A$

(2) a) $A+1=1$

b) $A\cdot 0=0$

(3) a) $\overline{\bar{A}}=A$

(4) a) $A+AB=A$

b) $A(A+B)=A$

(5) a) $A+\bar{A}B=A+B$

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

(1)(a) Proof:-

L.H.S $A+A$

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

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

$= AA+A\bar{A}+AA+A\bar{A}$

$= AA+A\bar{A}$

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

$= A$ //

$A+A=A$

if $A=0$; $0+0=0$

if $A=1$; $1+1=1$

$[1=A+\bar{A}]$

(1)(b) Proof:-

L.H.S $A\cdot A$

$= A\cdot A+(0)$

$= A\cdot A+A\cdot\bar{A}$

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

$= A$ //

$A\cdot A=A$

$A=0 \Rightarrow 0\cdot 0=0$

$A=1$; $1\cdot 1=1$

$[A\cdot\bar{A}=0]$

Recorded :-

$f(A,B,C) = [(A+B+C)(A+B+\bar{C})] \cdot [(A+B+C)\cdot(\bar{A}+B+C)] \cdot [(A+B+C)(A+\bar{B}+C)]$

Step 4 :- omitting the repeated product term

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

Ex :- Convert the given expression in standard pos forms

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

Step 1 :- find the missing literal in each product

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

B, C literals missing

step 2 :- OR product term with (missing literal + complement)

$f(A,B,C) = [A+(B\cdot\bar{B})+(C\cdot\bar{C})]\cdot(A+B+C)$

Step 3 :- Expand the term and recorded the terms

Expand :- $[A+(B+C)\cdot(B+\bar{C})\cdot(\bar{B}+C)\cdot(\bar{B}+\bar{C})]\cdot(A+B+C)$

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

Step 4 :- omitting the repeated product term

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

M-Notations :- (minterms and maxterms)

Decimal NO	Binary Numbers (421)	Minterm (m) (SOP)	Maxterm (M) (POS)
0	000	$\bar{A}\bar{B}\bar{C}$ (m_0)	$A+B+C$ (M_0)
1	001	$\bar{A}\bar{B}C$ (m_1)	$A+B+\bar{C}$ (M_1)
2	010	$\bar{A}B\bar{C}$ (m_2)	$A+\bar{B}+C$ (M_2)
3	011	$\bar{A}BC$ (m_3)	$A+\bar{B}+\bar{C}$ (M_3)
4	100	$A\bar{B}\bar{C}$ (m_4)	$\bar{A}+B+C$ (M_4)
5	101	$A\bar{B}C$ (m_5)	$\bar{A}+B+\bar{C}$ (M_5)
6	110	$A B\bar{C}$ (m_6)	$\bar{A}+\bar{B}+C$ (M_6)
7	111	ABC (m_7)	$\bar{A}+\bar{B}+\bar{C}$ (M_7)

Onlinebit

Minterm = Complement of maxterm

Example :- $f(A,B,C) = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}BC + AB\bar{C}$

$= m_0 + m_1 + m_3 + m_6$

$= \sum m(0,1,3,6)$

Ex :- $f(A,B,C) = (A+B+\bar{C}) \cdot (A+\bar{B}+\bar{C}) \cdot (\bar{A}+\bar{B}+C)$

$= M_1 \cdot M_3 \cdot M_6$

$= \pi M(1,3,6)$

Ex :- To find sum of product form to the given table

A B C	Y
0 0 0	0
0 0 1	0
0 1 0	1 ←
0 1 1	1 ←

[ABC as a Input variables]

[SOP is 1's]

A B C	y
1 0 1	0
1 1 0	1 ←
1 1 1	0

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

$= m_2 + m_3 + m_6$

$= \Sigma m(2,3,6)$

To find the product of sum from the given table

[Product of sum is zero's]

A+B+C	y
0 0 0	1
0 0 1	1
0 1 0	0 ←
0 1 1	1
1 0 0	1
1 0 1	0 ←
1 1 0	1
1 1 1	1

$f(A,B,C) = (A+\bar{B}+C)\cdot(\bar{A}+B+\bar{C})$.

$= M_2 \cdot M_5$

$= \pi M(2,5)$

M.JOSHNA

Algebraic Simplifications:

1. $A \cdot \bar{A} C = 0$
2. $ABCD + ABD = ABD$
3. $A(A+B) = A$
4. $AB + ABC + AB(D+E) = AB$
5. $xy + xyz + xy\bar{z} + \bar{x}yz$
6. $\bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}BC$
7. $ABC + A\bar{B}C + AB\bar{C}$
8. $A + \bar{A}B + A\bar{B} = A+B$

Ex:- $\overline{\bar{A}\bar{B} + \bar{A} + AB}$ $\quad [\overline{\bar{A}\bar{B}} = \bar{A} + \bar{B}]$

$\Rightarrow \overline{\bar{A} + \bar{B} + \bar{A} + AB}$ $\quad [\bar{B} + (A\bar{B}) = \bar{B}A) + (B \cdot \bar{B})$
$= (\bar{B}+A) \cdot (\bar{B}+B)]$

$\Rightarrow \overline{\bar{A} + \bar{B} + A\bar{B}}$ $\quad [\bar{B} + B = 1]$

$\Rightarrow \overline{\bar{A} + (\bar{B}+A) \cdot (\bar{B}+B)}$

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

$\Rightarrow \overline{(A+\bar{A}) + \bar{B}}$ $\quad [\bar{1} = 0]$

$\Rightarrow \overline{1 + \bar{B}}$

$\Rightarrow \bar{1}$

$\Rightarrow 0$

(1) $A \cdot \bar{A} C = 0$

$(A \cdot \bar{A}) C = 0$

$(0) C = 0$

$\overline{\bar{A}B + \bar{A} + AB}$

$\bar{A} + \bar{B} + \bar{A} + AB$

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

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

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

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

② $ABCD + ABD = ABD$ $[1+C=1]$

$\Rightarrow ABD(1+C)$

$\Rightarrow ABD$

$\Rightarrow ABCD + ABD = ABD$

③ $A(A+B) = A$ $[A \cdot A = 1]$

$\Rightarrow A \cdot A + AB$

$\Rightarrow A + AB$

$\Rightarrow A(1+B)$

$\Rightarrow A$

$A \cdot A + AB = A$

$A + AB = 0$

$(A+A)(A+B)$

$(A+AB)$

$A(1+B)$

$= A$

④ $AB + ABC + AB(D+E) = AB$

$\Rightarrow AB + ABC + AB(D+E)$

$\Rightarrow AB(1+C) + ABD + ABE$

$\Rightarrow AB + AB(D+E)$

$\Rightarrow AB(1+D+E)$

$\Rightarrow AB(1+D)+E)$

$\Rightarrow AB[1+E]$

$\Rightarrow AB(1)$

$\Rightarrow AB$

$xy + xyz + x\bar{y}z + \bar{x}yz = y(x+z)$

$= xy + yz + \bar{x}yz$

$xy + xy(z+\bar{z}) + \bar{x}yz$

$xy + y(x+\bar{x}z)$

$xy + y(x+\bar{x})(x+z)$

$xy + y(1)(x+z)$

$xy + yx + yz$

$xy + xy + yz$

$xy + yz$

$y(x+z)$

5. $xy + xyz + x\bar{y}z + \bar{x}yz = y(x+z)$

$\Rightarrow xy(1) + xyz + x\bar{y}z + \bar{x}yz$

$\Rightarrow xy(z+\bar{z}) + xyz + x\bar{y}z + \bar{x}yz$

$\Rightarrow xyz + xy\bar{z} + xyz + x\bar{y}z + \bar{x}yz$

$\Rightarrow xyz + xy(\bar{z}+z) + \bar{x}yz$

(or)

$xy(1+z) + (\bar{x}+x)yz$

$xy + yz$

$y(x+z)$ //

$\Rightarrow xy + x\bar{y} + \bar{x}yz$

$\Rightarrow xy + \bar{x}yz$

$\Rightarrow y(x + \bar{x}z) \Rightarrow y((x+\bar{x})\cdot(x+z)) \Rightarrow y(1)\cdot(x+z)$

$\Rightarrow y(x+z)$ //

[$(x)+(\bar{x}z)$
$(x+\bar{x})\cdot(xz)$]

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

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

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

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

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

$\Rightarrow \bar{A}((\bar{B}+B)\cdot(\bar{C}+B))$

$\Rightarrow \bar{A}((1)\cdot(\bar{C}+B))$

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

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

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

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

(7) $ABC + A\bar{B}C + AB\bar{C}$

$ABC + A\bar{B}C + AB\bar{C}$

$AC(B+\bar{B}) + AB\bar{C}$

$AC + AB\bar{C}$

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

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

$A(C+B)$

$AC + AB$ //

$ABC + A\bar{B}C + AB\bar{C}$

$AC(B+\bar{B}) + AB\bar{C}$

$AC + AB\bar{C}$

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

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

$A(C+B)$

$AC + AB$

M.JOSHNA

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

$\Rightarrow A+\bar{A}B+A\bar{B}$

$\Rightarrow A(1+\bar{B})+\bar{A}B$

$\Rightarrow A(1)+\bar{A}B$

$\Rightarrow A+\bar{A}B$

$\Rightarrow (A+\bar{A})\cdot(A+B)$

$\Rightarrow (1)\cdot(A+B)$

$\Rightarrow A+B$ //

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

Ex: Simplify the following three variable Expression by using Boolean algebra $Y = \Sigma m(1,3,5,7)$

Given Data $Y = \Sigma m(1,3,5,7)$

$= m_1+m_3+m_5+m_7$

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

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

$= \bar{A}C(1)+AC(1)$

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

$= C$ //

$Y = \Sigma m(1,3,5,7)$
$= \bar{A}\bar{B}C+\bar{A}BC+$
$A\bar{B}C+ABC$
$= \bar{A}C(B+\bar{B})+AC(B+$
$= \bar{A}C+AC$
$= C(A+\bar{A})$
$= C(1)$

Ex: Simplify the following three variable Expression by using Boolean algebra $Y = \Pi M(1,3,5,7)$

Given that $Y = \Pi M(1,3,5,7)$

$= M_1\cdot M_3\cdot M_5\cdot M_7$

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

$= M_1\cdot M_3\cdot M_5\cdot M_7$
$= (A+B+\bar{C})\cdot(A+\bar{B}+C)$

$C \ldots + (A\cdot\bar{B})+(A\cdot\bar{C})+(B\cdot A)+(B\cdot\bar{B})+(B\cdot\bar{C})+(\bar{C}\cdot A)+(\bar{C}\cdot\bar{B})+(\bar{C}\cdot\bar{C})]\cdot$

$\ldots (B\cdot\bar{A})+(B\cdot\bar{B})+(B\cdot\bar{C})+(\bar{C}\cdot\bar{A})+$

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

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

$= B\bar{C}(A+\bar{A}) + \bar{B}\bar{C}(1+A) + \bar{C}(A+\bar{A}) + \bar{A}BC(1+C) + \bar{C}(1+B) + \bar{A}\bar{B}\bar{C}$

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

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

Ex (2) Simplify the falowing three Variable Expresion. Convert the Exprsion into minterm complementary property. and s

$Y = \pi_M(1,3,5,7)$

Sol:- The given Expresion is maxterm $Y = \pi_M(1,3,5,7)$.

To convert given sequences to minterm Complementary form

minterm = Complementary of Maxterm

The minterm is given by $= \Sigma m(0,2,4,6)$

$Y = \Sigma m(0,2,4,6) = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + A\bar{B}\bar{C} + AB\bar{C}$

$= m_0 + m_2 + m_4 + m_6 = \bar{A}\bar{C}(B+\bar{B}) + A\bar{C}(B+\bar{B})$

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

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

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

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

(14m) ** * transform Each of the following Canonical Expresion into its other Canonical form its decimal notations.

(i) $f(x,y,z) = \Sigma m(1,3,5)$

(ii) $f(w,x,y,z) = \pi_M(0,2,5,6,7,8,9,11,12)$

(i) $f(x,y,z) = \Sigma m(1,3,5)$ $f(x,y,z) = \pi_M(0,2,4,6,7)$

$f(x,y,z) = \pi_M(0,2,4,6,7)$

$= M_0 \cdot M_2 \cdot M_4 \cdot M_6 \cdot M_7 \; (x+y+z)\cdot(x+\bar{y}+z)\cdot(\bar{x}+y+z)\cdot(\bar{x}+\bar{y}+z)\cdot(\bar{x}+\bar{y}+\bar{z})$

$= (x+x\bar{y}+xz+xy+yz+xz+\bar{y}z+z)(\bar{x}+\bar{x}\bar{y}+\bar{x}z+\bar{x}y+yz+\bar{x}z+z\bar{y}+z).$
$(\bar{x}+y+\bar{z})$

$= xyz+x\bar{y}z+xz+x\bar{y}z+xyz+x\bar{y}z+xz+xyz+xyz+\bar{x}yz+\bar{x}yz+\bar{x}yz+$
$yz+\bar{x}yz+yz+xyz+x\bar{y}z+xz+\bar{x}\bar{y}z+\bar{x}\bar{y}z+\bar{x}\bar{y}z+\bar{x}\bar{y}z+$
$\bar{y}z+\bar{y}z+\bar{x}z+\bar{x}\bar{y}z+\bar{x}z+\bar{x}yz+yz+\bar{x}z+z\bar{y}+z)$

$= xyz+x\bar{y}z+xz+\bar{x}yz+yz+\bar{x}\bar{y}z+\bar{y}z+\bar{x}z+z$

$= xz(y+\bar{y})+yz(\bar{x}+1)+\bar{y}z(1+\bar{x})+z(x+\bar{x})+z$

$= xz+yz+\bar{y}z+z \Rightarrow x(x+y)+z(1+\bar{y})$

$= xz+xy+z$ //

(ii) $f(w,x,y,z) = \Pi M(0,2,5,6,7,8,9,11,12)$

$f(w,x,y,z) = \Pi_M(0,2,5,6,7,8,9,11,12)$

$= \Sigma m(1,3,4,10,14,15)$

$= m_1+m_3+m_4+m_{10}+m_{13}+m_{14}+m_{15}$

$= \bar{w}\bar{x}\bar{y}z+\bar{w}\bar{x}yz+\bar{w}x\bar{y}\bar{z}+w\bar{x}y\bar{z}+wx\bar{y}z+$
$+wxy\bar{z}+wxyz$

$= \bar{w}\bar{x}z(\bar{y}+y)+wxz(y+\bar{y})+wy\bar{z}(x+\bar{x})+\bar{w}x\bar{y}\bar{z}$

$= \bar{w}\bar{x}z+wxz+wy\bar{z}+\bar{w}x\bar{y}\bar{z}$

M.JOSHNA

Decimal NO	Binary number (8421)	minterm	Maxterm
8	1000	$A\bar{B}\bar{C}\bar{D}$ (m8)	(M) $\bar{A}+B+C+D$
9	1001	$A\bar{B}\bar{C}D$ (m9)	$\bar{A}+B+C+\bar{D}$
10	1010	$A\bar{B}C\bar{D}$ (m10)	$\bar{A}+B+\bar{C}+D$
11	1011	$A\bar{B}CD$	$\bar{A}+B+\bar{C}+\bar{D}$
12	1100	$AB\bar{C}\bar{D}$	$\bar{A}+\bar{B}+C+D$
13	1101	$AB\bar{C}D$	$\bar{A}+\bar{B}+C+\bar{D}$
14	1110	$ABC\bar{D}$	$\bar{A}+\bar{B}+\bar{C}+D$
15	1111	$ABCD$	$\bar{A}+\bar{B}+\bar{C}+\bar{D}$

UNIT-III

Gate-Level Minimization

Map :-

→ The map method gives us a systamatic approch for simplyfying a boolean expresion. The map method first proposed by veitch and modified by karnaugh, Hence it is known as the veitch diagram or the karnaugh map (k-map)

One-Variable, two Variable, three Varible and four variable Maps :-

1 Variable $= 2^1 = 2$ cells

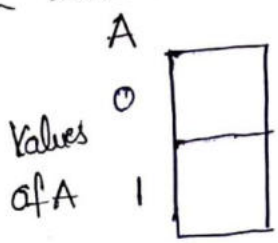
2 Variable $= 2^2 = 4$ cells

3 Variable $= 2^3 = 8$ cells

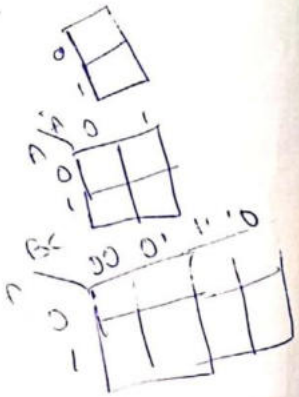
4 Variable $= 2^4 = 16$ cells

1 Var $= 2^1 = 2$ cells
2 Var $= 2^2 = 4$ cells
3 Var $= 2^3 = 8$ cells
4 Var $= 2^4 = 16$ cells

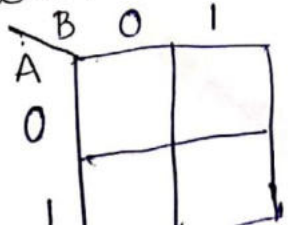
1 Variable k-Map :-



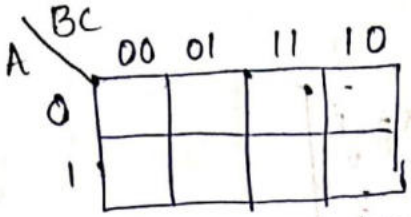
1-Variable Map (2cells)



2-Variable k-Map :-



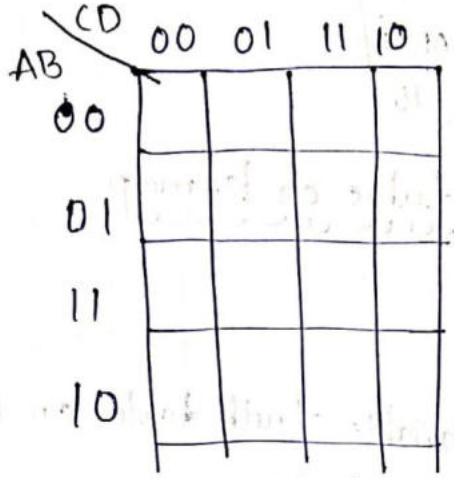
3-Variable k-map :-



3-Variable k-map (8 cells)

4- Variable k-map :-

Values of CD in gray code



4- Variable k-map (16 cells)

1- Variable :-

0	$\bar{A}$	$\bar{A}$ 0
1	A	A 1

2- Variable :-

A \ B	$\bar{B}$	B
$\bar{A}$	$\bar{A}\bar{B}$ 0	$\bar{A}B$ 1
A	$A\bar{B}$ 2	AB 3

3- Variable :-

A \ BC	$\bar{B}\bar{C}$	$\bar{B}C$	BC	$B\bar{C}$
$\bar{A}$	$\bar{A}\bar{B}\bar{C}$ 0	$\bar{A}\bar{B}C$ 1	$\bar{A}BC$ 3	$\bar{A}B\bar{C}$ 2
A	$A\bar{B}\bar{C}$ 4	$A\bar{B}C$ 5	ABC 7	$AB\bar{C}$ 6

4-Variable

AB \ CD	$\bar{C}\bar{D}$	$\bar{C}D$	CD	$C\bar{D}$
$\bar{A}\bar{B}$	$\bar{A}\bar{B}\bar{C}\bar{D}$ 0	$\bar{A}\bar{B}\bar{C}D$ 1	$\bar{A}\bar{B}CD$ 3	$\bar{A}\bar{B}C\bar{D}$ 2
$\bar{A}B$	$\bar{A}B\bar{C}\bar{D}$ 4	$\bar{A}B\bar{C}D$ 5	$\bar{A}BCD$ 7	$\bar{A}BC\bar{D}$ 6
AB	$A\bar{B}\bar{C}\bar{D}$ 12	$A\bar{B}\bar{C}D$ 13	$ABCD$ 15	$ABC\bar{D}$ 14
$A\bar{B}$	$A\bar{B}\bar{C}\bar{D}$ 8	$A\bar{B}\bar{C}D$ 9	$A\bar{B}CD$ 11	$A\bar{B}C\bar{D}$ 10

Representation of Truth table on k-map

two-Variable k-map

→ The representation of two variable truth table on k-map is given below

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

A \ B	0	1
0	0	1
1	1	0

OR

A \ B	$\bar{B}$	B
$\bar{A}$	0	1
A	1	0

three-Variable k-map :-

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1

A \ BC	00	01	11	10
0	0 (0)	0 (1)	0 (3)	1 (2)
1	1 (4)	1 (5)	0 (7)	0 (6)

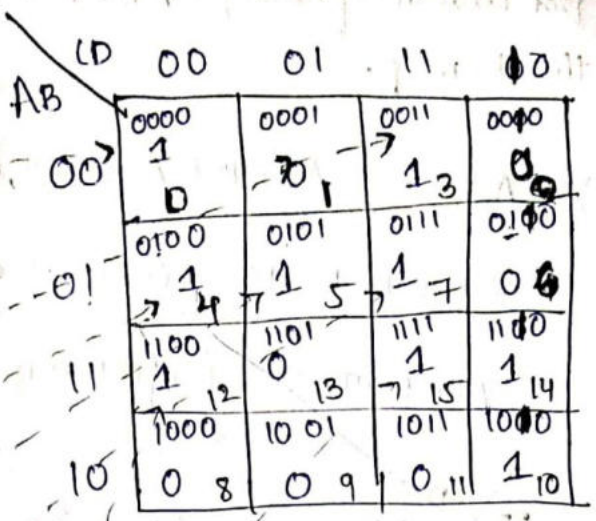
OR

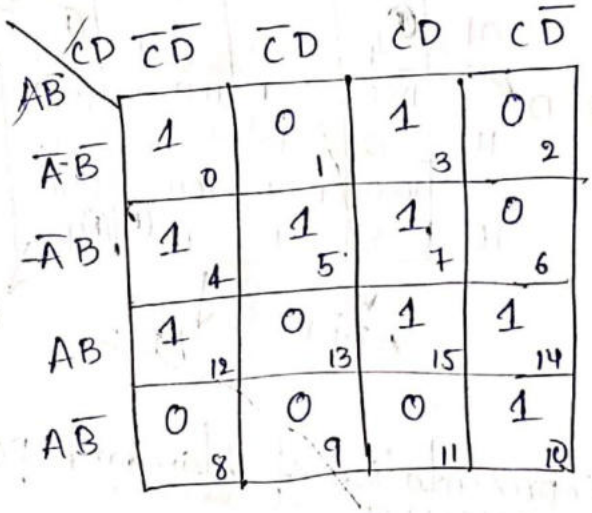
AB \ BC	$\bar{B}\bar{C}$	$\bar{B}C$	BC	$B\bar{C}$
$\bar{A}$	0 (0)	0 (1)	0 (3)	1 (2)
A	1 (4)	1 (5)	1 (7)	0 (6)

M.JOSHNA

Four-Variable K-map:

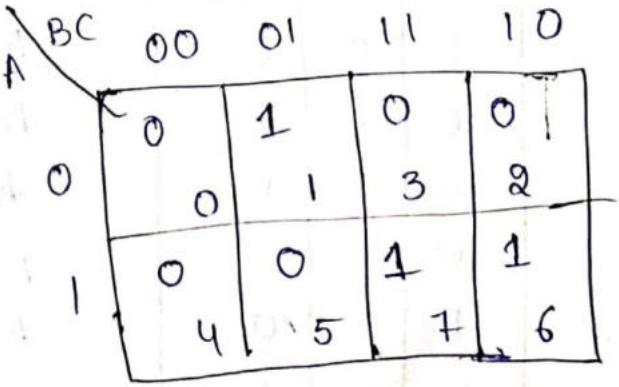
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1





Ex: Plot Boolean Expression $Y = AB\bar{C} + ABC + \bar{A}\bar{B}C$ on the k-map

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

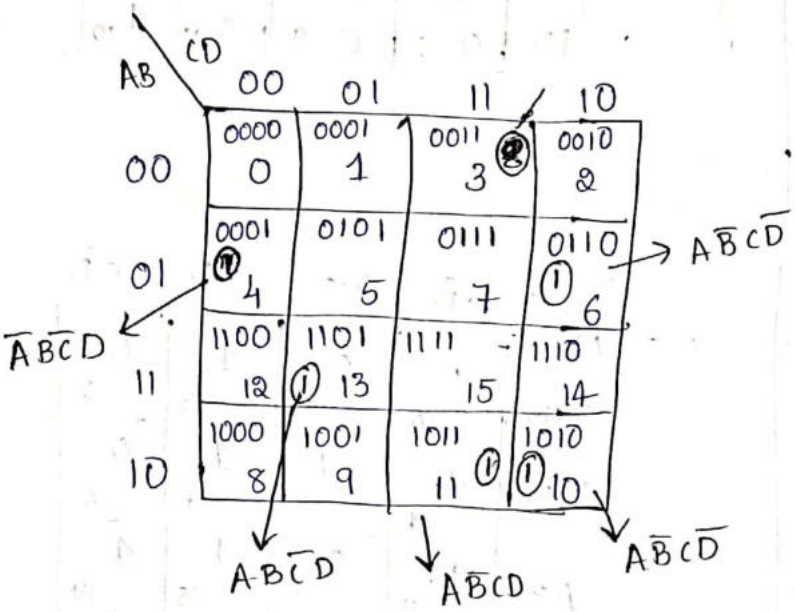


Ex:- plot Boolean expression $y = \bar{A}B\bar{C}\bar{D} + A\bar{B}C\bar{D} + \bar{A}BC\bar{D} + A\bar{B}CD + AB\bar{C}D$ on the k-map.

$$y = \bar{A}B\bar{C}\bar{D} + A\bar{B}C\bar{D} + \bar{A}BC\bar{D} + A\bar{B}CD + AB\bar{C}D$$

$$= 0100 + 1010 + 0110 + 1011 + 1101$$

$$= m_4 + m_{10} + m_6 + m_{11} + m_{13}$$



0100 + 1010 +

Representation of standard POS on k-map

Three variable k-map:-

A	B	C	Maxterm (M)
0	0	0	$A+B+C$ (M_0)
0	0	1	$A+B+\bar{C}$ (M_1)
0	1	0	$A+\bar{B}+C$ (M_2)
0	1	1	$A+\bar{B}+\bar{C}$ (M_3)
1	0	0	$\bar{A}+B+C$ (M_4)
1	0	1	$\bar{A}+B+\bar{C}$ (M_5)
1	1	0	$\bar{A}+\bar{B}+C$ (M_6)
1	1	1	$\bar{A}+\bar{B}+\bar{C}$ (M_7)

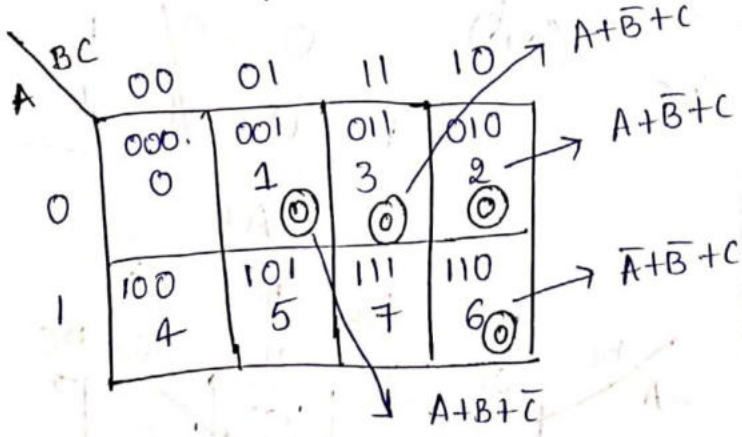
foure Variable k-map :-

A	B	C	D	Maxterm (M)
0	0	0	0	$A+B+C+D$ (M_0)
0	0	0	1	$A+B+C+\bar{D}$ (M_1)
0	0	1	0	$A+B+\bar{C}+D$ (M_2)
0	0	1	1	$A+B+\bar{C}+\bar{D}$ (M_3)
0	1	0	0	$A+\bar{B}+C+D$ (M_4)
0	1	0	1	$A+\bar{B}+C+\bar{D}$ (M_5)
0	1	1	0	$A+\bar{B}+\bar{C}+D$ (M_6)
0	1	1	1	$A+\bar{B}+\bar{C}+\bar{D}$ (M_7)
1	0	0	0	$\bar{A}+B+C+D$ (M_8)
1	0	0	1	$\bar{A}+B+C+\bar{D}$ (M_9)
1	0	1	0	$\bar{A}+B+\bar{C}+D$ (M_{10})
1	0	1	1	$\bar{A}+B+\bar{C}+\bar{D}$ (M_{11})
1	1	0	0	$\bar{A}+\bar{B}+C+D$ (M_{12})
1	1	0	1	$\bar{A}+\bar{B}+C+\bar{D}$ (M_{13})
1	1	1	0	$\bar{A}+\bar{B}+\bar{C}+D$ (M_{14})
1	1	1	1	$\bar{A}+\bar{B}+\bar{C}+\bar{D}$ (M_{15})

Ex:- plot Boolean expression $Y = (A+\bar{B}+C)(A+\bar{B}+\bar{C})(\bar{A}+\bar{B}+C)(A+B+\bar{C})$ on the k-map :

$Y = M_2 \cdot M_3 \cdot M_6 \cdot M_1$

[Maxterm ல் "1" represent
marter "0" represent]

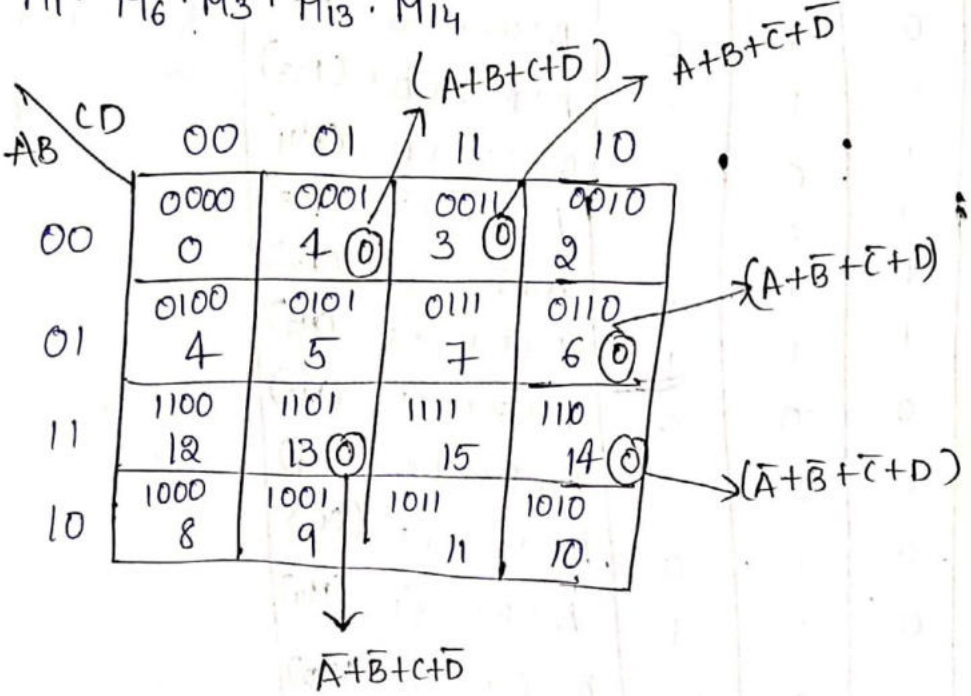


Ex-① plot Boolean Expression $y = (A+B+C+\bar{D})(A+\bar{B}+\bar{C}+D)(A+B+\bar{C}+\bar{D})$

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

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

$y = M_1 \cdot M_6 \cdot M_3 \cdot M_{13} \cdot M_{14}$



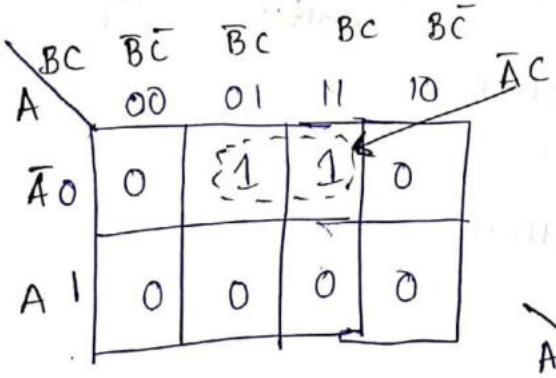
Grouping cells for simplifaction:-

Grouping two adjcent ones (pair)

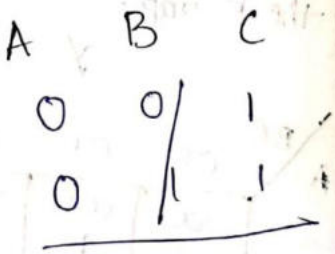
Ex:- $y = \bar{A}\bar{B}C + \bar{A}B\bar{C}$

$= \bar{A}C(B+\bar{B})$ $\quad [B+\bar{B}=1]$

$= \bar{A}C$



(or)



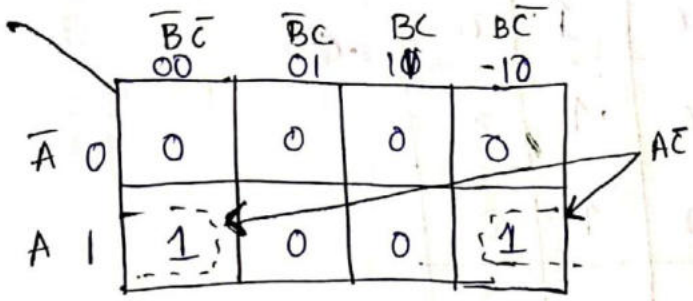
Ex-2 $Y = \bar{A}BC + ABC = BC$

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

Ex÷ ③ $Y = A\bar{B}\bar{C} + AB\bar{C}$

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

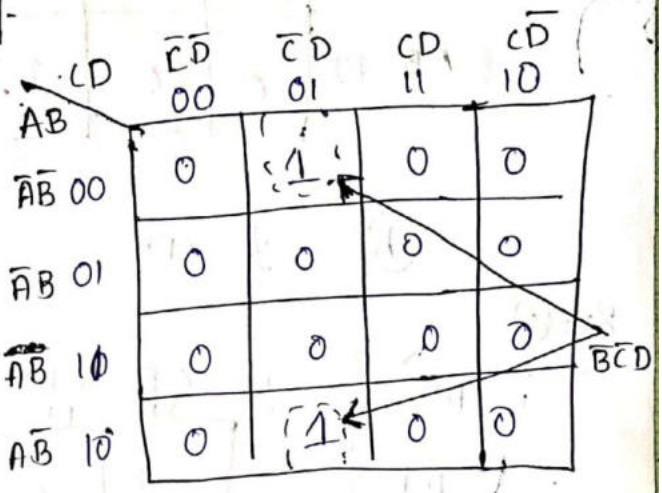
$= A\bar{C}$



Ex÷ ④ $Y = \bar{A}\bar{B}\bar{C}D + A\bar{B}\bar{C}D$

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

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



Ex÷ ⑤ $Y = \bar{A}\bar{B}C + \bar{A}BC + ABC$

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

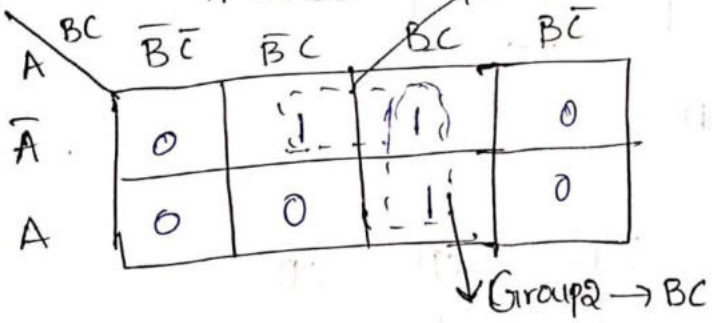
$= \bar{A}C(\bar{B}+B) + BC(\bar{A}+A)$

$= \bar{A}C + BC$

A B C

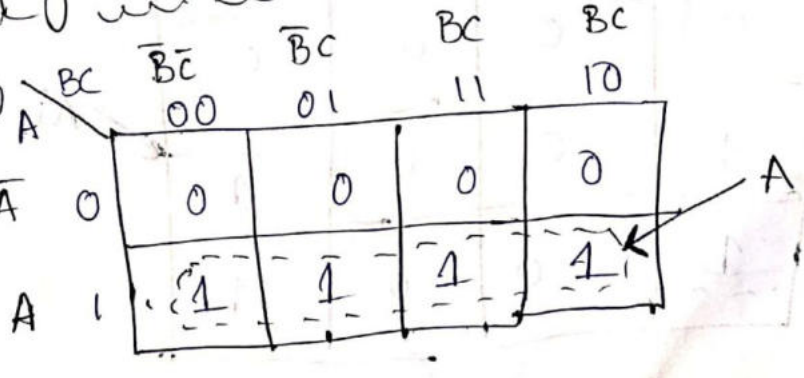
1

(OR)



Grouping four adjcent ones (Quad)

Ex÷ ①



Ex÷2

AB \ CD	$\bar{C}\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
$\bar{A}\bar{B}$ 00	0	0	1	0
$\bar{A}B$ 01	0	0	1	0
AB 11	0	0	1	0
$A\bar{B}$ 10	0	0	1	0

CD

(b) $y = CD$

Ex÷(3)

AB \ CD	$\bar{C}\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
$\bar{A}\bar{B}$	0	0	0	0
$\bar{A}B$	0	1	1	0
AB	0	1	1	0
$A\bar{B}$	0	0	0	0

BD

(3) $y = BD$

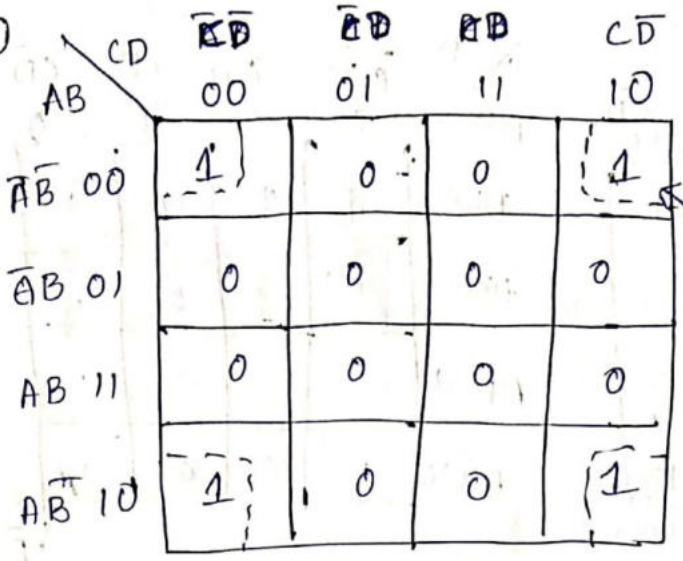
Ex÷(4)

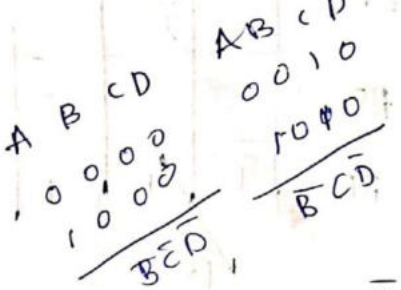
AB \ CD	$\bar{C}\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
$\bar{A}\bar{B}$ 00	0	0	0	0
$\bar{A}B$ 01	0	0	0	0
AB 11	1	0	0	1
$A\bar{B}$ 10	1	0	0	1

$A\bar{D}$

$y = A\bar{D}$

Ex:- 5



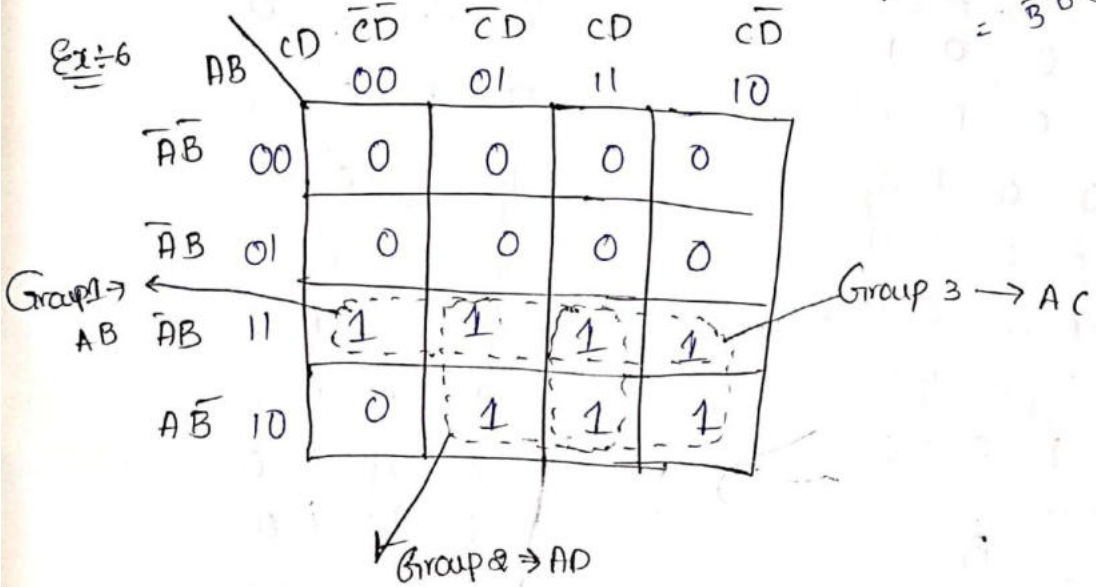


Ans:- $Y = \bar{B}\bar{D}$

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

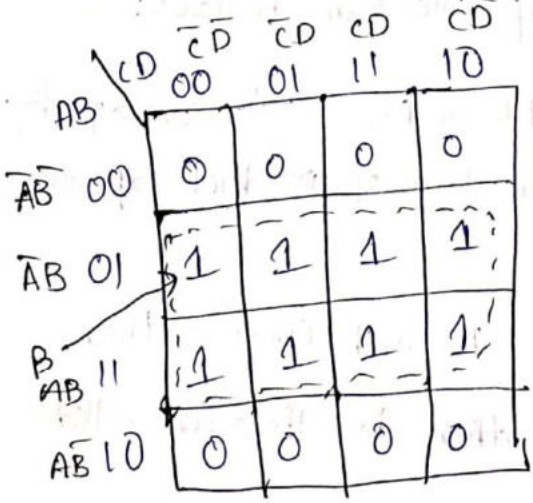
$= \bar{B}\bar{D}(C + \bar{C})$

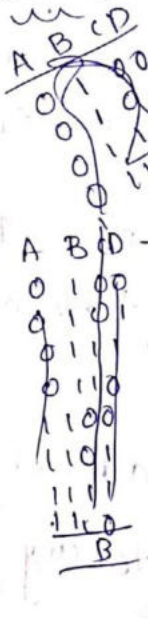
Ex:-6

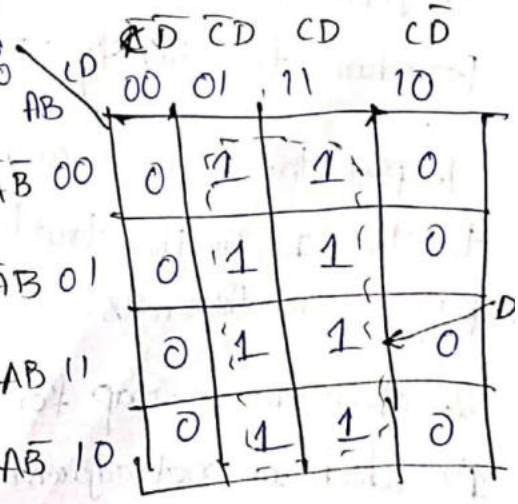


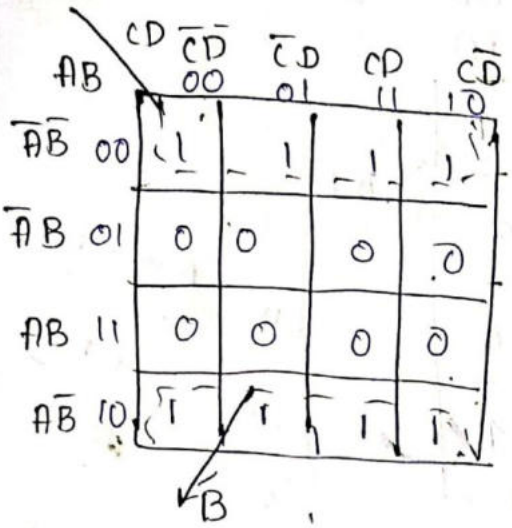
(A) $Y = AB + AD + AC$

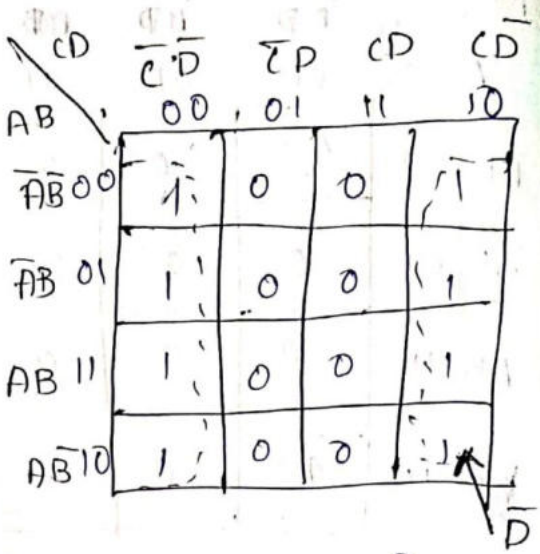
Grouping Eight Adjacent ones (octet)

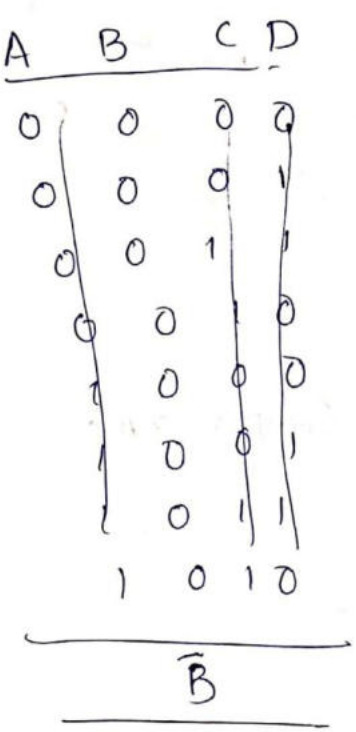


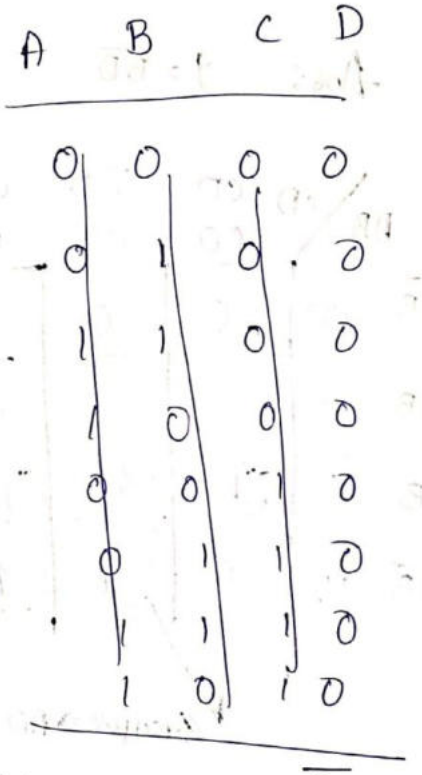












Simplifications of SOP Expressions

→ from the above discussion we can outline generalized procedure to simplify Boolean expressions as follows:

1. plot the k-map and place 1s in those cells corresponding to the 1s in the truth table. or sum of product expression. place 0s in other cells.

2. check the k-map for adjacent 1s and encircle those 1's which are not adjacent to any other 1s. these are called

3. check for those 1's which are adjacent to only one other 1 and encircle such pairs.

4. check for quads and octets of adjacent 1s even if it contains some 1s that have already bee encircled. while doing this make sure that there are minimum no. of groups.

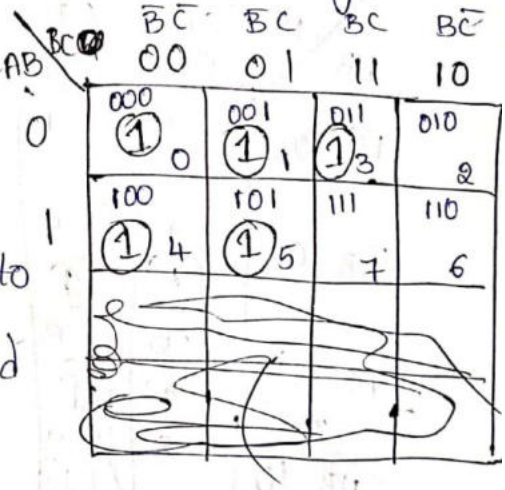
5. combaine any pairs necessary to include any 1s that have not yet been grouped.

6. From the simplified expresion by summing product terms of all groups.

Ex:- minimize the expresion $Y = A\bar{B}C + \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}\bar{C} + \bar{A}\bar{B}\bar{C}$

Solution:
$$Y = 101 + 001 + 011 + 100 + 000$$
$$= m_5 + m_1 + m_3 + m_4 + m_0$$
$$= \Sigma m(5, 3, 4, 0)$$

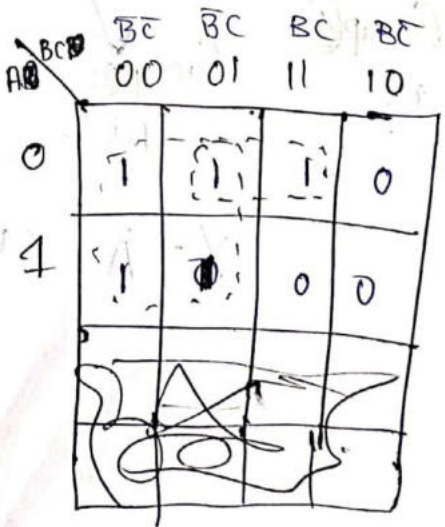
Step 1:- The k-map for three variables and it is ploted according to the given expression.



Step 2:- There are no isolated 1s

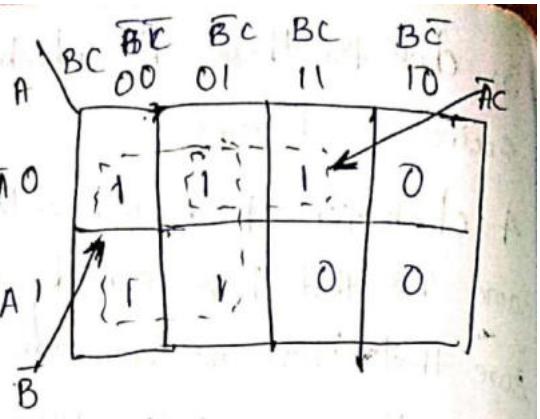
Step 3:- 1 in the cell 3 adjacent only to 1 in the cell 1. This pair is combained and refered to as group 1

Step 4:- There is no octet, but there is a quad cells 0, 1, 4 and 5 from a quard. This quad is combained and reffered to as a group.



Step 5:- All 1's have already been

Step 6: Each group genrates a term in expresion for y. In group 1 B variable is eliminated and in group 2 variables A and care eliminated and we get



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

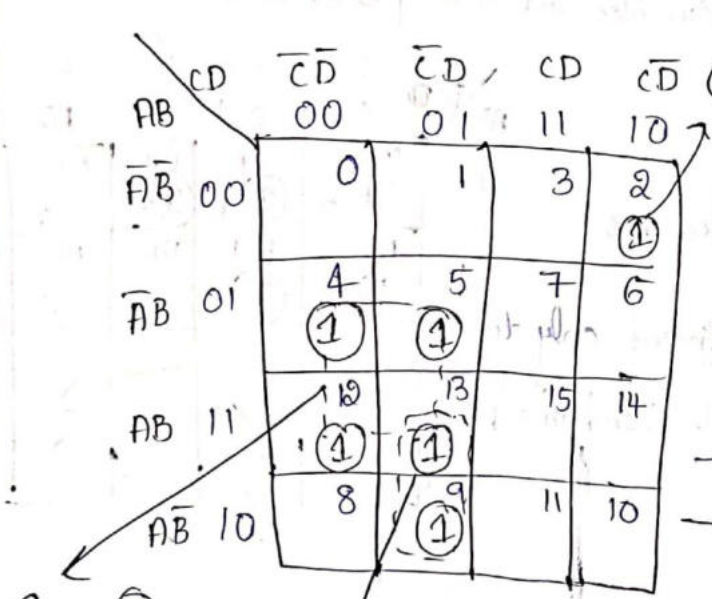
Ex:- Minimize the Expression

$$y = \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + AB\bar{C}\bar{D} + AB\bar{C}D + A\bar{B}\bar{C}D + \bar{A}\bar{B}C\bar{D}$$

$$= 0100 + 0101 + 1100 + 1101 + 1001 + 0010$$

$$= m_4 + m_5 + m_{12} + m_{13} + m_9 + m_2$$

$$= \Sigma m(4, 5, 12, 13, 9, 2)$$



G_1

A	B	C	D
0	1	0	0
0	1	0	1
1	1	0	0
1	1	0	1

$B\bar{C}$

G_2

A	B	C	D
1	1	0	1
1	0	0	1

$A\bar{C}D$

G_3

A	B	C	D
0	0	1	0

$\bar{A}\bar{B}C\bar{D}$

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

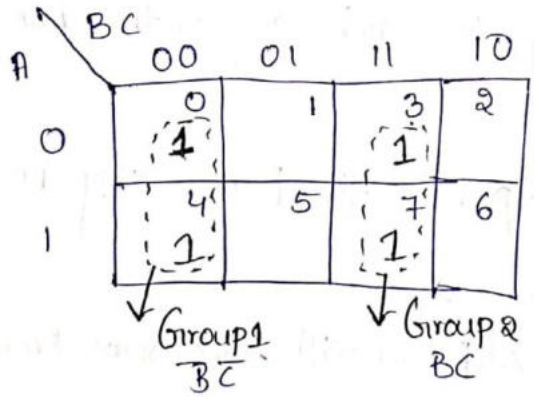
* Simplify the logic function specified by the truth table by using the kanaugh map method. Y is the output variable and A, B and C are the input variables

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

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

$= m_0 + m_3 + m_4 + m_7$

$= \Sigma m(0,3,4,7)$



Group 1

A	B	C
0	0	0
1	0	0

$\bar{B}\bar{C}$

Group 2

A	B	C
0	1	1
1	1	1

BC

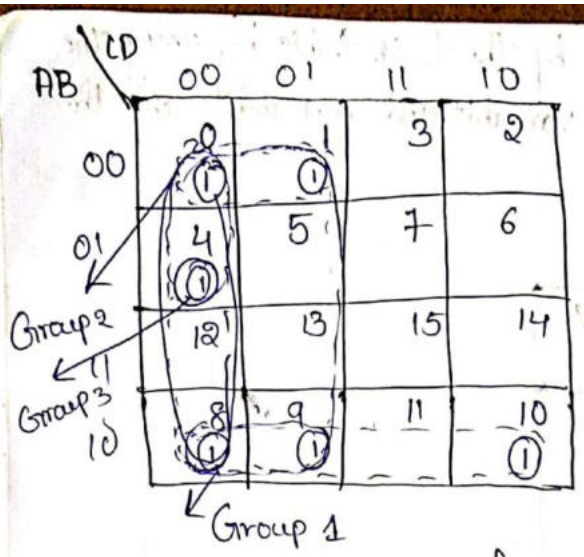
$\therefore Y = \bar{B}\bar{C} + BC$

Ex:- Reduce the following function using k-map technique. $\bar{A}\bar{C}\bar{D} + A\bar{B}\bar{D} + \bar{C}$

$f(A,B,C,D) = \Sigma m(0,1,4,8,9,10)$

$= m_0 + m_1 + m_4 + m_8 + m_9 + m_{10}$

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



Group 1

A	B	C	D
1	0	0	0
1	0	1	0

$A\bar{B}\bar{D}$

Group 2

A	B	C	D
0	0	0	0
0	0	0	1
1	0	0	0
1	0	0	1

$\bar{B}\bar{C}$

Group 3

A	B	C	D
0	1	0	0

$\bar{A}B\bar{C}\bar{D}$

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

Product of Sum simplification:-

1. plot the k-map and place 0s in those cells corresponding to the 0s in the truth table or maxterms in the product of sum expression
2. check the k-map for adjacent 0s and encircle those 0s which are not adjacent to any other 0s. These are called isolated 0s.
3. Check for those 0s which are adjacent to only one other 0 and encircle such pairs.
4. check for quads and octets of adjacent 0s even if it contains some 0s that have already been encircled. while doing this make sure that there are minimum no. of groups.
5. Combaine any pairs necessary to include any 0s that have not yet been grouped.
6. From the simplified pos expression for F by taking product of sum terms of all the groups

To get familiar with these steps we will solve some examples.

Ex:- minimize the Expression

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

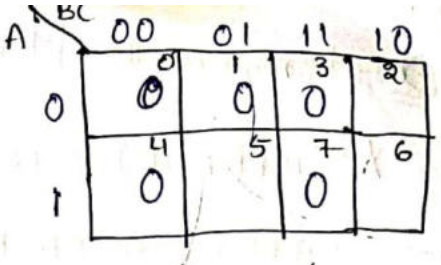
$= (001)(011)$

$= (A+B+\bar{C}) = M_1$, $(A+\bar{B}+\bar{C}) = M_3$, $(\bar{A}+\bar{B}+\bar{C}) = M_7$

$(\bar{A}+B+C) = M_4$

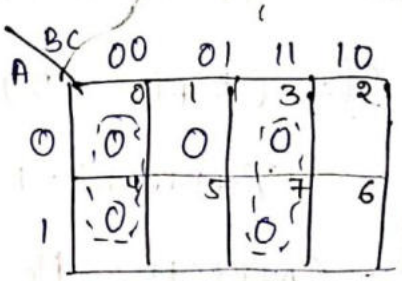
M.JOSHNA

Step 1 :- (a) shows the k-map for three variable and it is plotted according to given maxterms



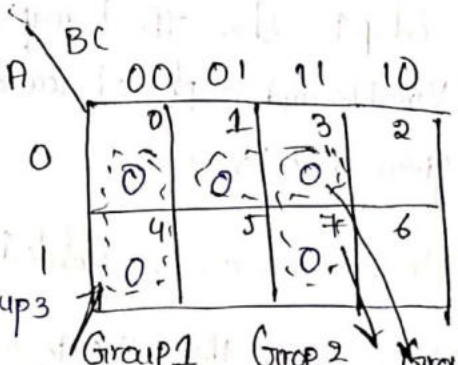
Step 2 :- three are no isolated 0s

Step 3 :- 0 in the cell 4 is adjacent only to 0 in the cell 0 and 0 in cell 7 is adjacent only to 0 in the cell 3. These two pairs are combained and refered to as group 1 and group 2 respectively.



Step 4 :- There are no quads and octets

Step 5 : The 0 in the cell 1 can be combained with 0 in the cell 3 to be from a pair. This pair is referred to as group 3



Step 6 : In group 1 and in group 2, A is eliminated, where as in group 3 variable B is eliminated and we get

Group 1	Group 2	Group 3
A B C	A B C	A B C
0 0 0	0 1 1	0 0 1
1 0 0	1 1 1	0 1 1
$\bar{B}\bar{C}$	BC	$\bar{A}C$

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

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

[According to Demorgans]

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

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

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

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

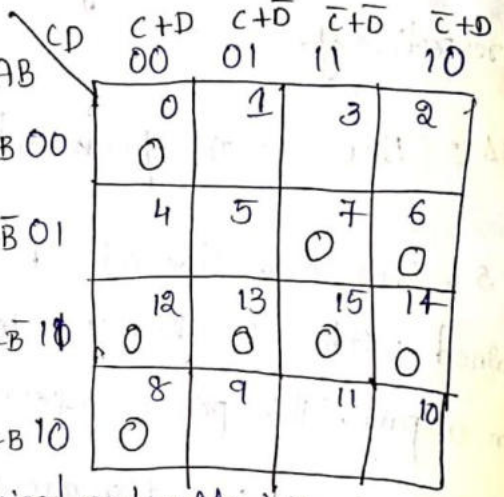
Ex:- Minimize the following expression in the pos form

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

Sol:- $(\bar{A}+\bar{B}+C+D) = M_{12}$, $(\bar{A}+\bar{B}+\bar{C}+D) = M_{14}$, $(\bar{A}+\bar{B}+\bar{C}+\bar{D}) = M_{15}$

$(\bar{A}+B+C+D) = M_8$, $(A+\bar{B}+\bar{C}+D) = M_6$, $(A+\bar{B}+\bar{C}+\bar{D}) = M_7$

$(A+B+C+D) = M_0$ and $(\bar{A}+\bar{B}+C+\bar{D}) = M_{13}$



Step1: shows the k-map for four Variable and, it plotted according to given max terms

Step2: There are no isolated 1's

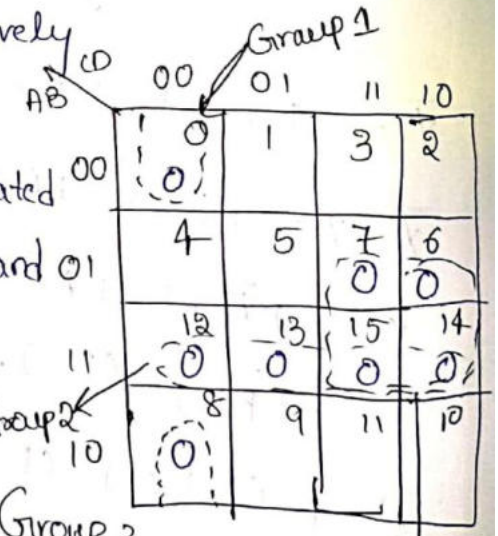
Step3: 0 in the cell 0 is adjacent only to in the cell this pair is combained and refferd toas group1

Step4: There are two quads cells 12, 13, 14 and 15 form a quad 1 and cells 6, 7, 14, 15 forms a quad 2. These two quads are reffered to as group2 and group3, respectively

Step5: All os have already been grouped

Step6:- In group 1, Variable A is eliminated in group2, Variable C and D are eliminated and 01 in group 3 variable A and D are eliminated.

∴ we get simplified POS Expression



Group1

A	B	C	D
0	0	0	0
1	0	0	0

$\bar{B}\,\bar{C}\,\bar{D}$

Group2

A	B	C	D
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0

Group 3

A	B	C	D
0	1	1	1
0	1	1	0
0	1	1	1
1	1	1	0

$$\bar{Y} = \bar{B}\bar{C}\bar{D} + AB + BC$$

$$Y = \bar{\bar{Y}} = \overline{\bar{B}\bar{C}\bar{D} + AB + BC}$$

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

Incompletely specified functions (Don't Care terms or conditions)

Ex:

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	X
1	1	1	X

Sol:

A \ BC	00	01	11	10
0	0	1	3 1	2
1	4	5 1	7 X	6 X

Don't Care Conditions

Group 1

A \ BC	00	01	11	10
0	0	1	3 1	2
1	4	5 1	7 1	6 0

Group 1

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

C

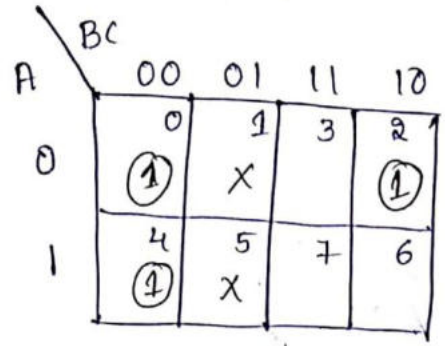
$\therefore Y = C$

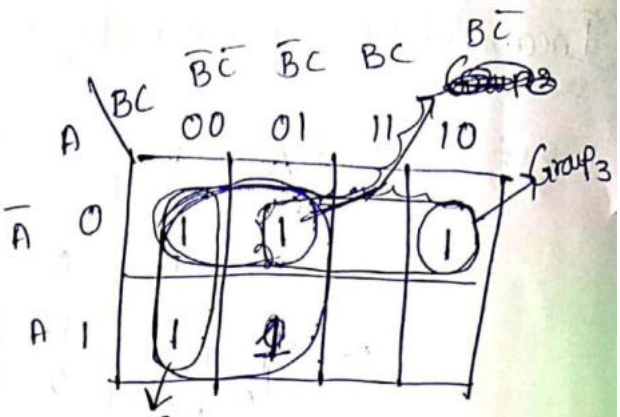
Describing Incomplete Boolean function

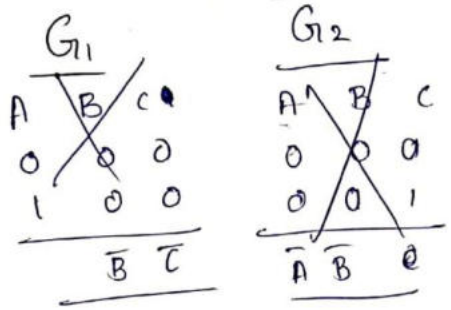
Ex:- (i) $f(A,B,C) = \Sigma m(0,2,4) + d(1,5)$

(ii) $f(A,B,C) = \Pi M(2,5,7) + d(1,3)$

(i) $f(A,B,C) = \Sigma m(0,2,4) + d(1,5)$







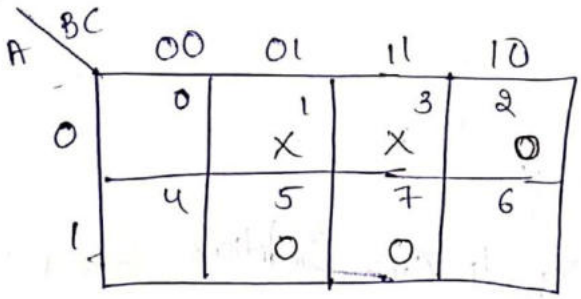
$Y = \bar{B}\bar{C} + A\bar{B} + B\bar{C}$ //

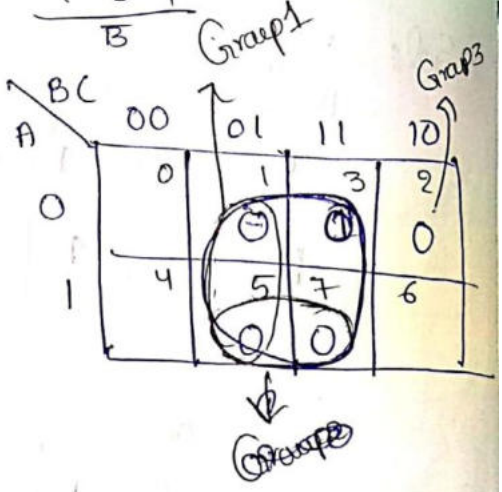
Group1

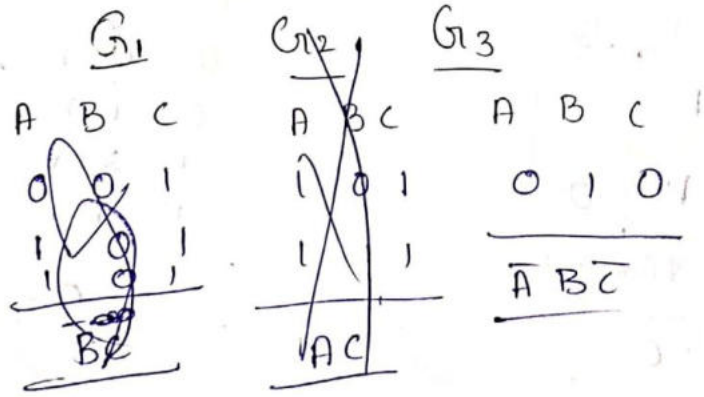
A	B	C	$Y = \bar{B} + B\bar{C}$
0	0	0	
0	0	1	
1	0	0	
1	0	1	

B̄

(ii) $f(A,B,C) = \Pi M(2,5,7) + d(1,3)$







G1

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

C

$Y = \bar{A}B\bar{C} + [illegible]C$ //

Don't care conditions in logic design

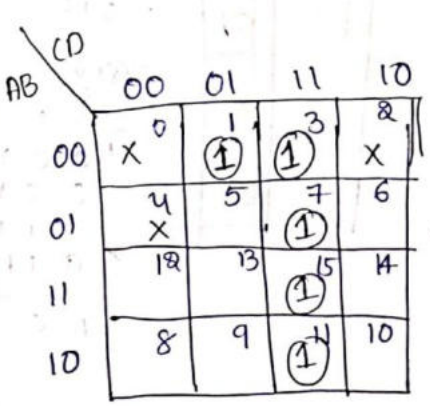
	A	B	C	D	P
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	D
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	–
11	1	0	1	1	–
12	1	1	0	0	–
13	1	1	0	1	–
14	1	1	1	0	–
15	1	1	1	1	–

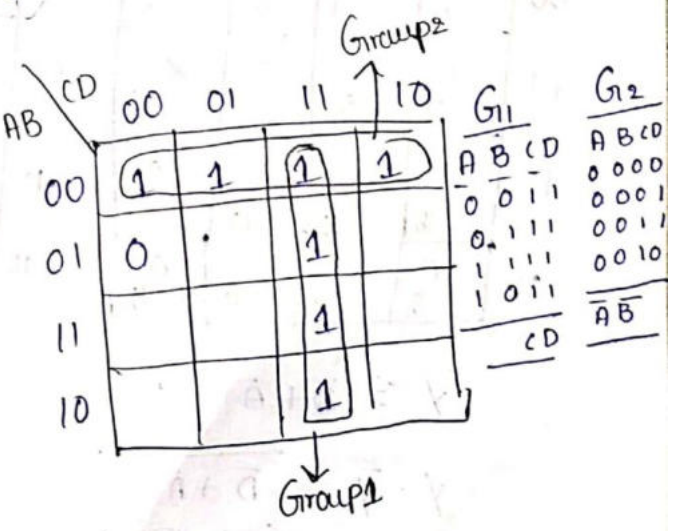
$P = \Sigma m(1,2,4,7,8) + d(10,11,12,13,14,15)$

Ex: find the reduced SOP form of the following function.

$f(A,B,C,D) = \Sigma m(1,3,7,11,15) + \Sigma d(0,2,4)$

$f(A,B,C,D) = \Sigma m(1,3,7,11,15) + \Sigma d(0,2,4)$

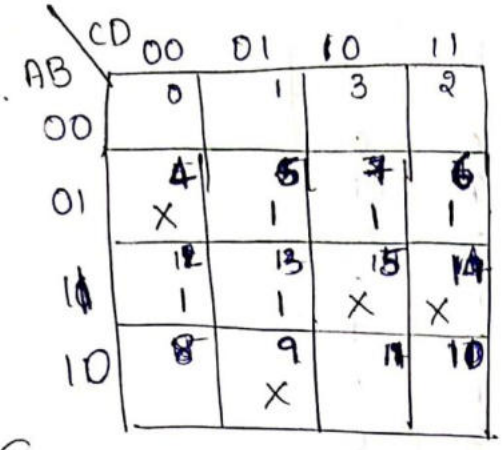


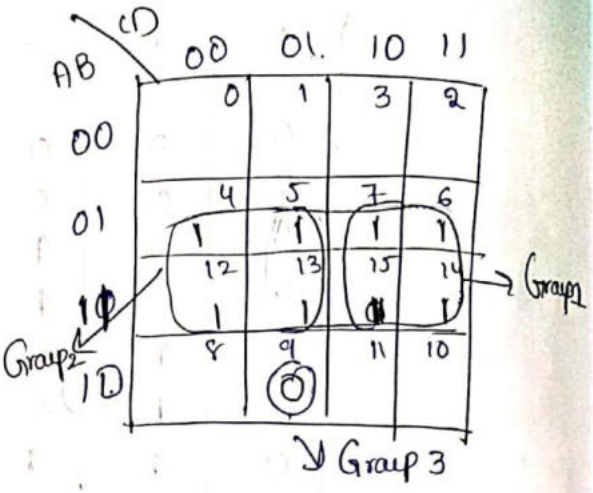


$\therefore Y = \bar{A}\bar{B} + CD$

Ex:- Reduce the following function using k-map technique

$f(A,B,C,D) = \Sigma m(5,6,7,12,13) + \Sigma d(4,9,14,15)$





G_1

A	B	C	D
0	1	0	0
0	1	0	1
1	1	0	0
1	1	0	1

$B\bar{C}$

G_{12}

A	B	C	D
0	1	1	0
0	1	1	1
1	1	1	0
1	1	1	1

BC

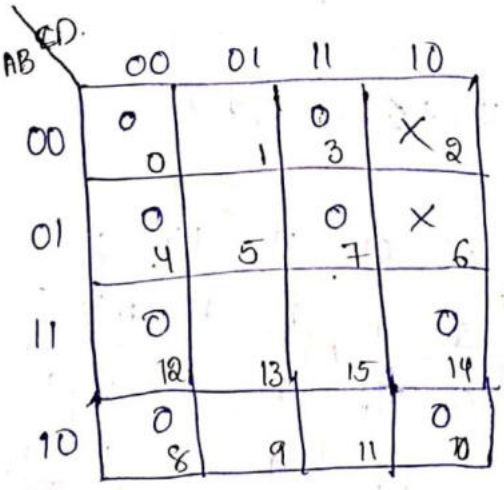
A	B	C	D
1	0	0	1

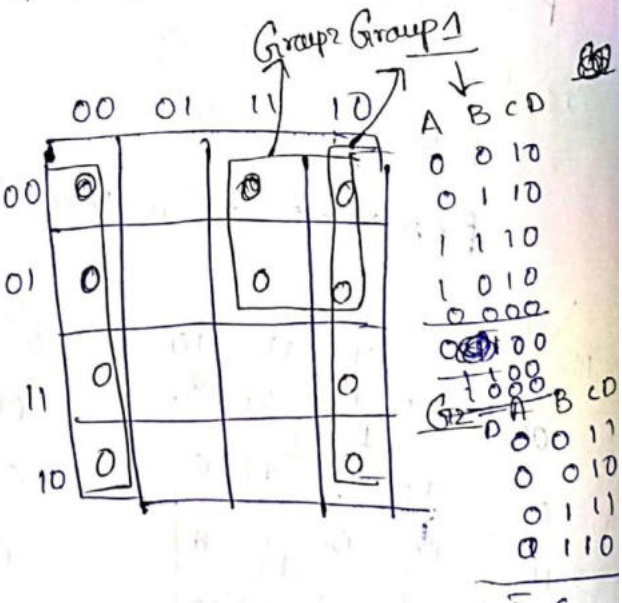
$A\bar{B}\bar{C}D$

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

Ex:- Reduce the following function by using k-map technique

$f(A,B,C,D) = \Pi M(0,3,4,7,8,10,12,14) + d(2,6)$





A	B	C	D
0	0	1	0
0	1	1	0
1	1	1	0
1	0	1	0

G_2

A	B	C	D
0	0	1	1
0	1	1	0
1	1	1	0

$\bar{A}C$

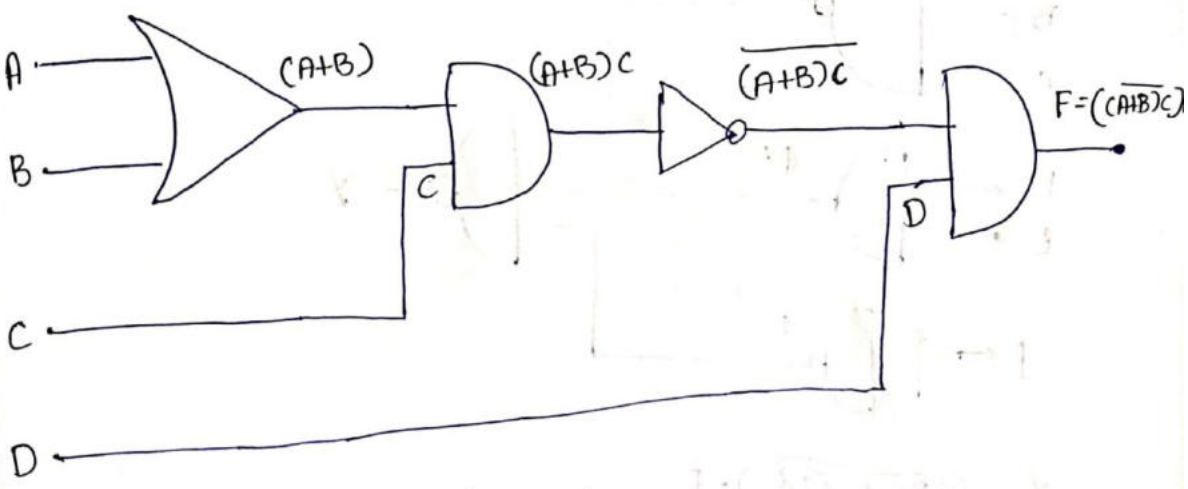
$\bar{Y} = \bar{D} + \bar{A}C$

$Y = \bar{\bar{Y}} = \overline{\bar{D} + \bar{A}C}$

$= (\bar{\bar{D}}) \cdot (\bar{\bar{A}}C)$

* ~~F(A+B, C+D)~~ =

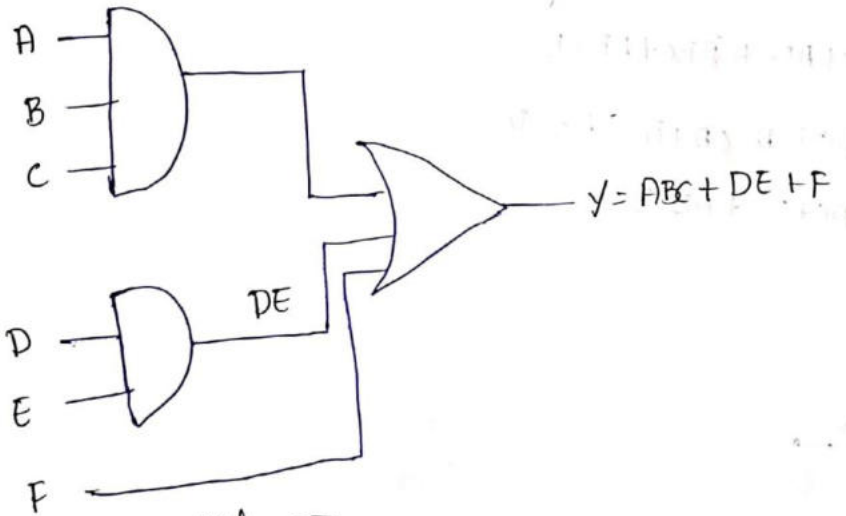
Ex:- $F = (\overline{(A+B)C})D$ to Implement logic Design?



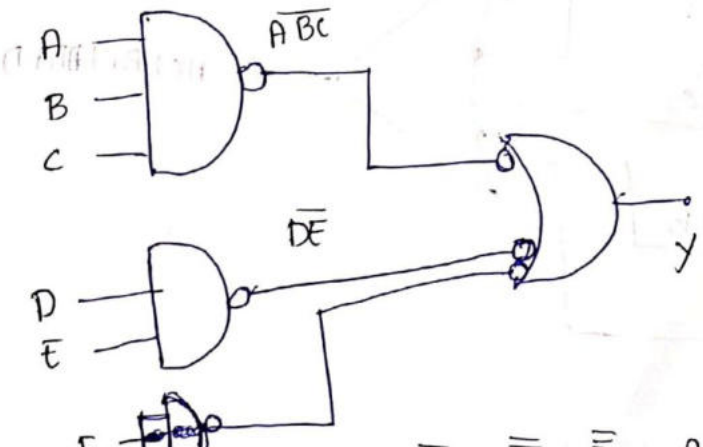
NAND to NAND Implementation :-

$y = ABC + DE + F$

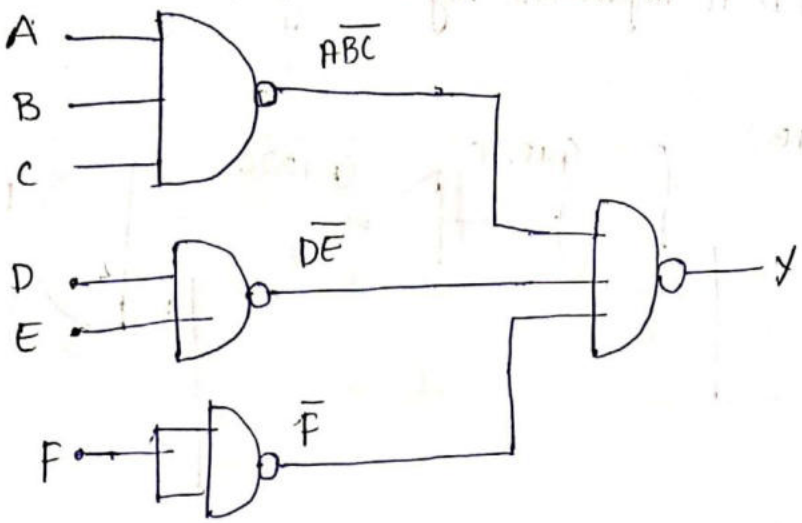
(i) AND - OR



(ii) NAND - Bubble OR



(iii) NAND - NAND



$$Y = \overline{(\overline{ABC}) \cdot (\overline{DE}) \cdot \overline{F}}$$

$$= (\overline{\overline{ABC}}) + (\overline{\overline{DE}}) + \overline{F}$$

$$= ABC + DE + F$$

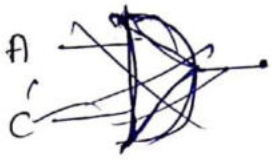
Ex: $Y = AC + ABC + \bar{A}BC + AB + D$ Implement the following Boolean function with NANA to NAND logic.

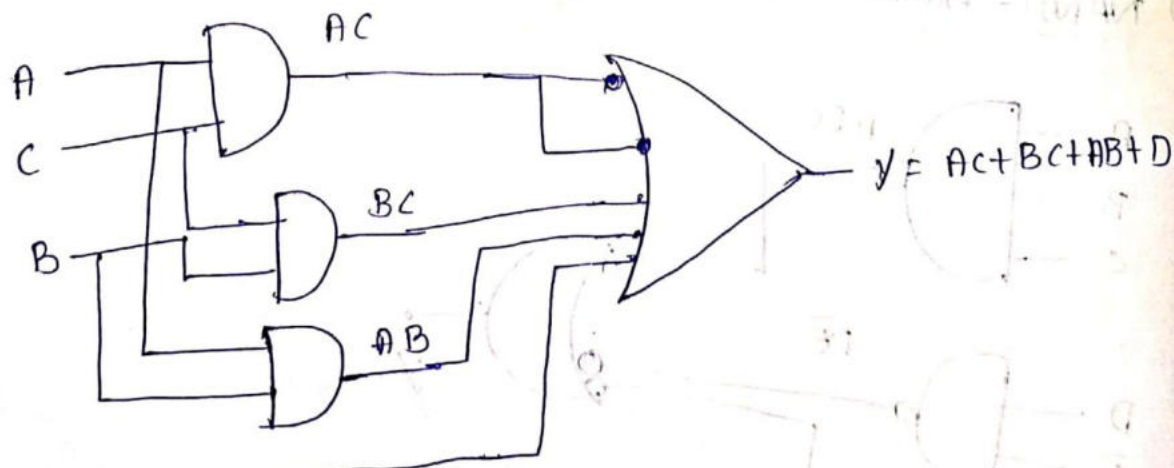
$$Y = AC + ABC + \bar{A}BC + AB + D$$

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

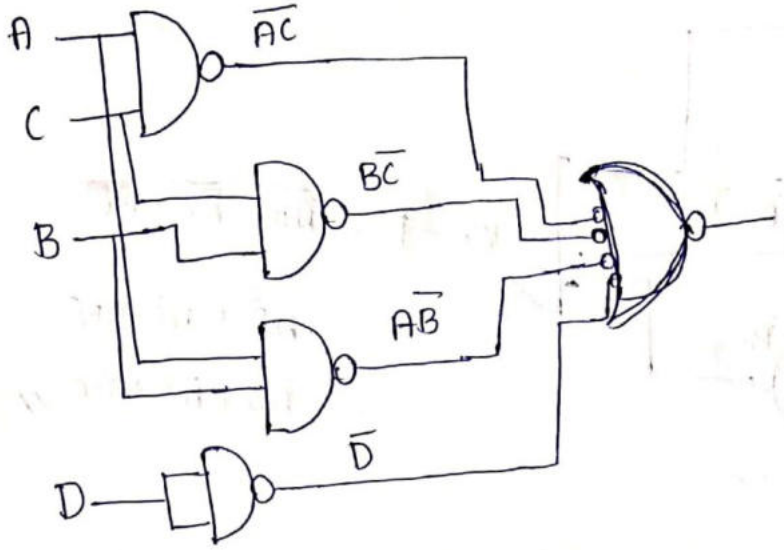
$$Y = AC + BC + AB + D$$

(i) AND - OR





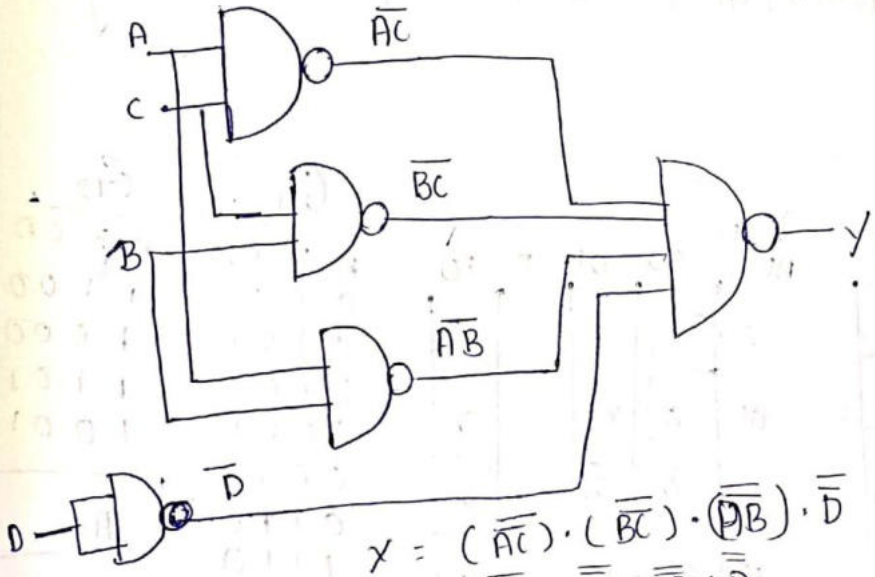
(ii) NAND - Bubble OR



$Y = \overline{\overline{AC}} + \overline{B\overline{C}} + \overline{A\overline{B}} + \overline{\overline{D}}$

$= AC + BC + AB + D$

(iii) NAND - NAND

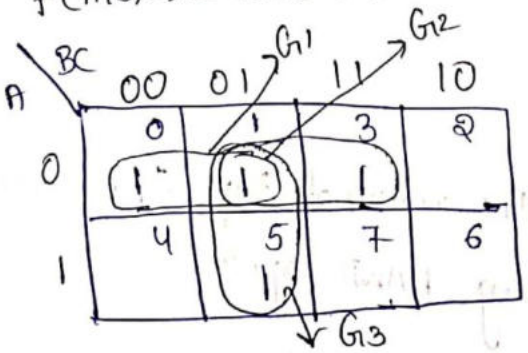


$X = (\overline{\overline{AC}}) \cdot (\overline{\overline{BC}}) \cdot (\overline{\overline{AB}}) \cdot \overline{\overline{D}}$

$= \overline{\overline{AC}} + \overline{\overline{BC}} + \overline{\overline{AB}} + \overline{\overline{D}}$

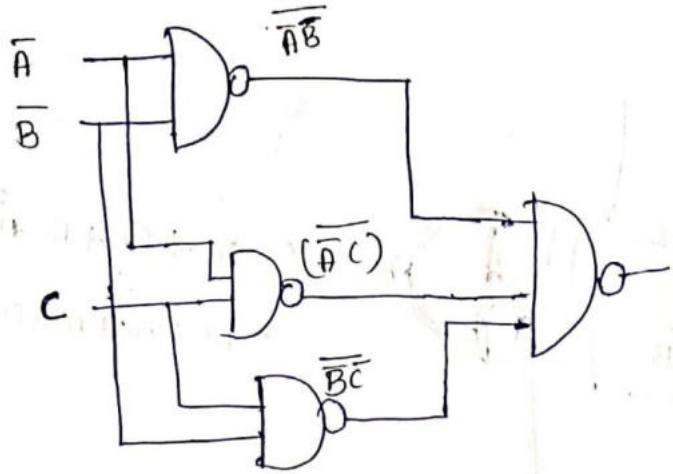
$= AC + BC + AB + D$

Ex: Implementation the following Boolean Expresion with NAND-NAND logic. $F(A,B,C) = \Sigma m(0,1,3,5)$



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

NAND - NAND



$F = \overline{\bar{A}\bar{B} \cdot \bar{A}C \cdot \bar{B}C}$

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

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

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

Ex:① find the reduced pos form of the following Equation

$F(A,B,C,D) = \Sigma m(1,3,7,11,15) + d(0,2,5)$ Implement and using NAND logic?

AB \ CD	00	01	11	10
00	X 0	1 1	1 3	X 2
01	0 4	X 5	1 7	0 6
11	0 12	0 13	1 15	0 14
10	0 8	0 9	1 11	0 10

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

Group2 Group1

G_1

A	B	C	D
0	0	0	0
0	1	0	0
1	1	0	0
1	0	0	0
0	0	1	0
0	1	1	0
1	1	1	0
1	0	1	0

$\bar{D}$

G_2

A	B	C	D
1	1	0	0
1	0	0	0
1	1	0	1
1	0	0	1

$A\bar{C}$

$\bar{Y} = \bar{D} + A\bar{C}$

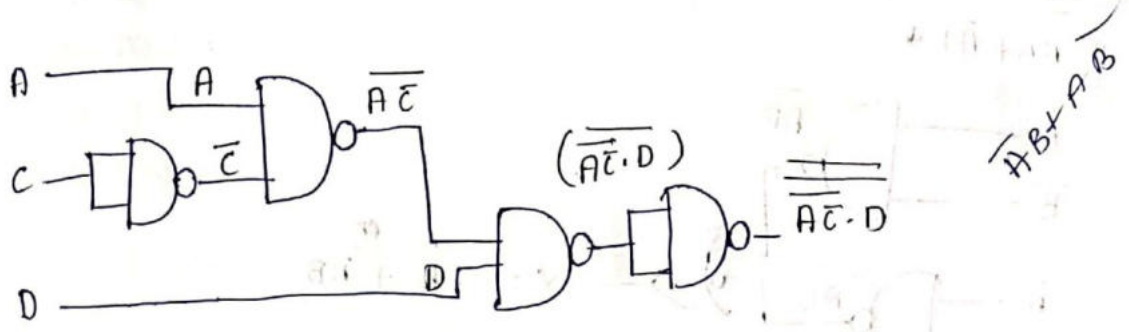
$Y = \overline{\bar{Y}} = (\overline{\bar{D}})(\overline{A\bar{C}})$

$\therefore Y = D \cdot (\bar{A} + C)$

Ex:② ~~using k-map, Determine the minimal sop Expression and resize the simplified Expression using NAND logic?~~

~~$f(w,x,y,z) = \Pi M(0,2,3,7,8,9,10)$~~

$\therefore Y = D \cdot (\bar{A} + C)$



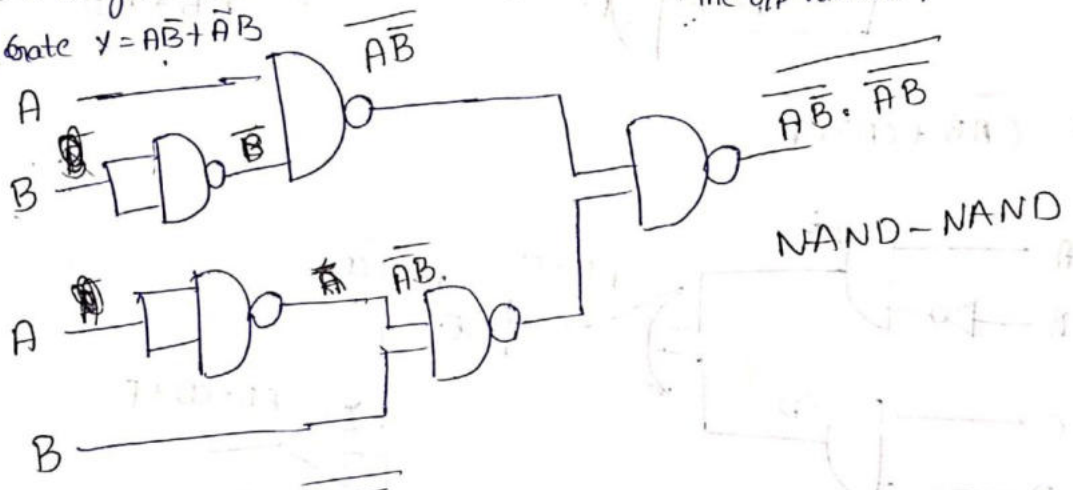
$$Y = (\overline{A\bar{C}} \cdot D)$$
$$= (\bar{A} + \bar{\bar{C}}) \cdot D$$
$$Y = (\bar{A} + C) \cdot D \;//$$

Ex:- To Implement X-OR gate by using NAND-NAND logic gate

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

The X-OR gate function is two Inputs and one output. The i/p variables are A and B. The o/p variable Y

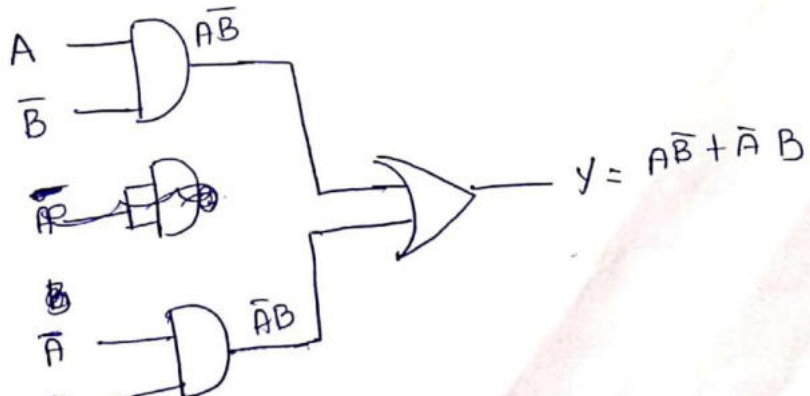
X-OR Gate $Y = A\bar{B} + \bar{A}B$



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

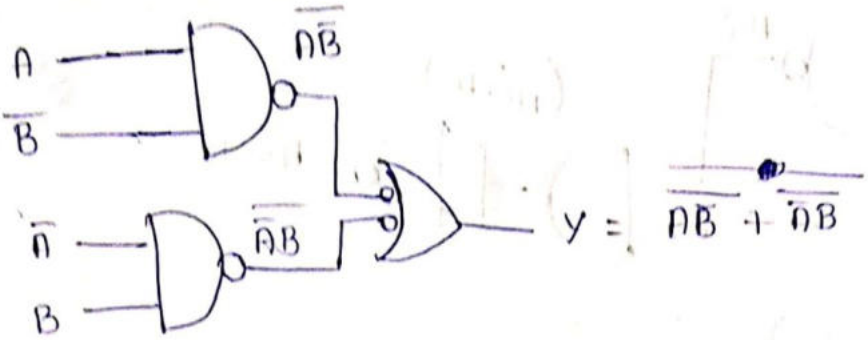
$$Y = A\bar{B} + \bar{A}B \;//$$

(9) AND-OR



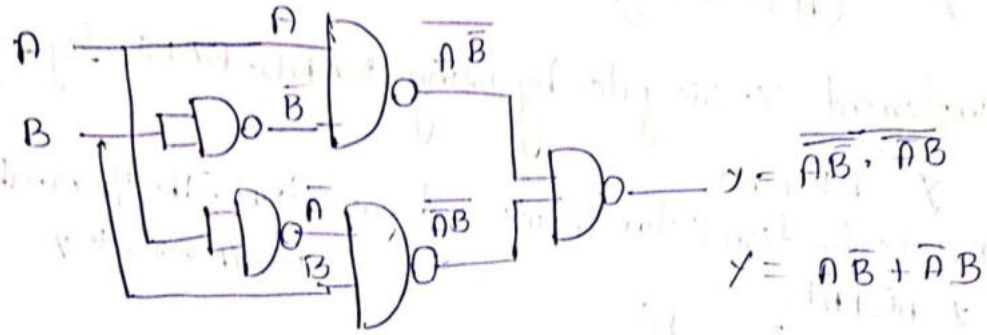
(ii) NAND - Bubbled OR

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

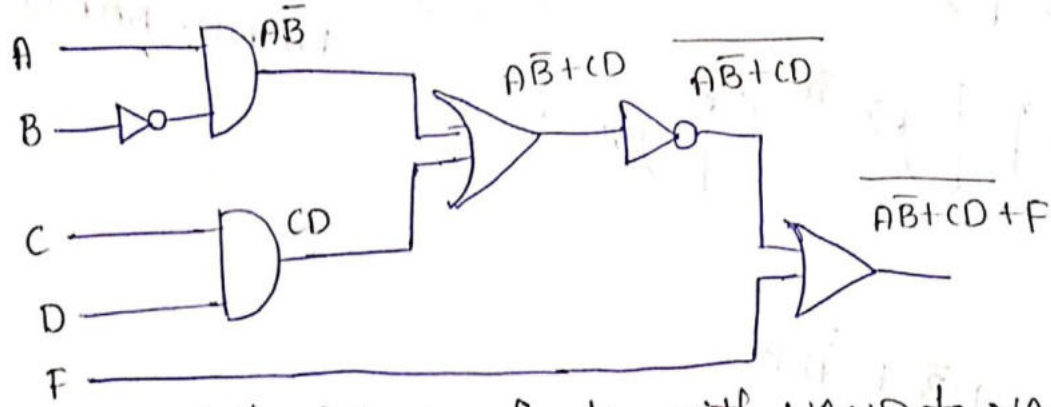


$Y = A\bar{B} + \bar{A}B$

(iii) NAND - NAND



Ex: $\overline{(A\bar{B} + CD) + F}$



Ex: Implement the following function with NAND-to NAND logic

$f(A, B, C, D) = \Sigma m(0, 1, 3, 5)$

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

NOR- NOR Implementation :-

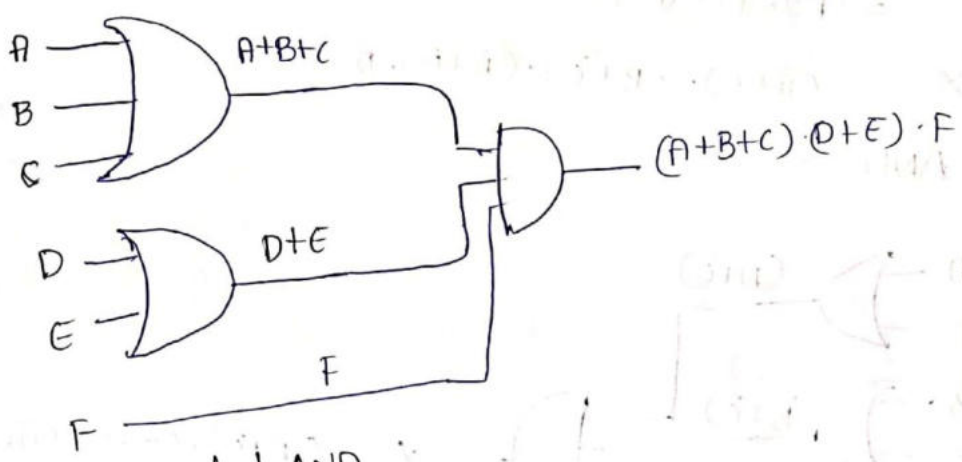
(i) OR - AND

(ii) NOR - Bubbled AND

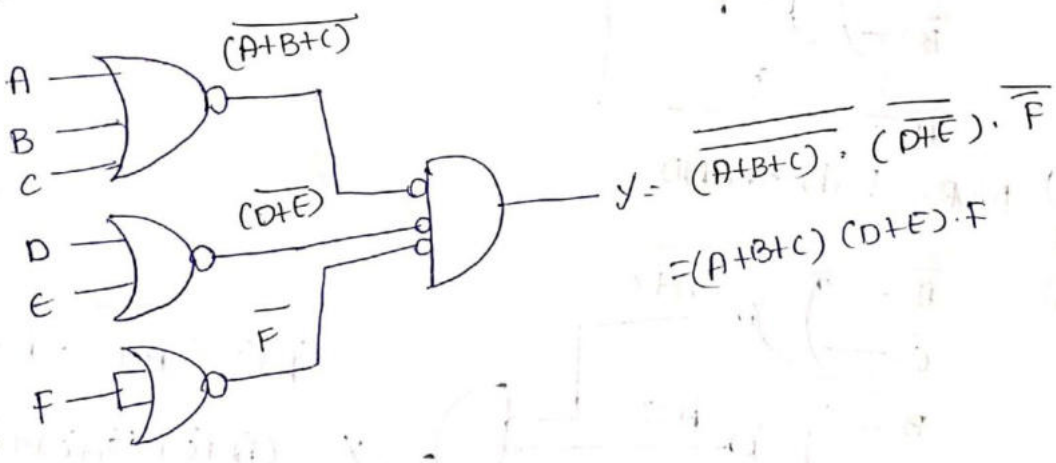
(iii) NOR - NOR

Ex :- $Y = (A+B+C) \cdot (D+E) \cdot F$

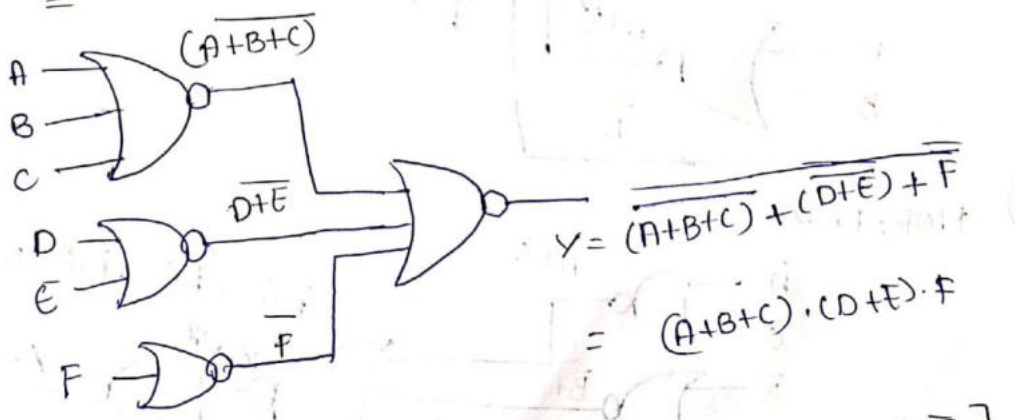
(i) OR - AND



(ii) NOR - Bubbled AND



(iii) NOR - NOR



$[\because \overline{A+B} = \overline{A} \cdot \overline{B}]$

$\overline{\overline{A}} = A]$

Ex:- Implement the following Boolean function with NOR-NOR logic

$Y = AC + BC + AB + D$

The given Boolean Expression

$y = AC + BC + AB + D$

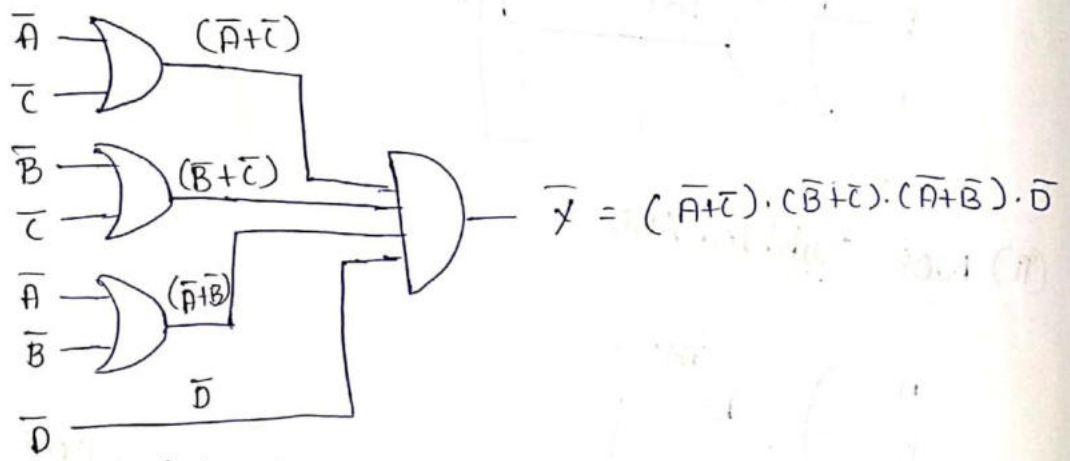
Assumption. By using Duality theorem

$$\overline{Y} = \overline{AC + BC + AB + D}$$

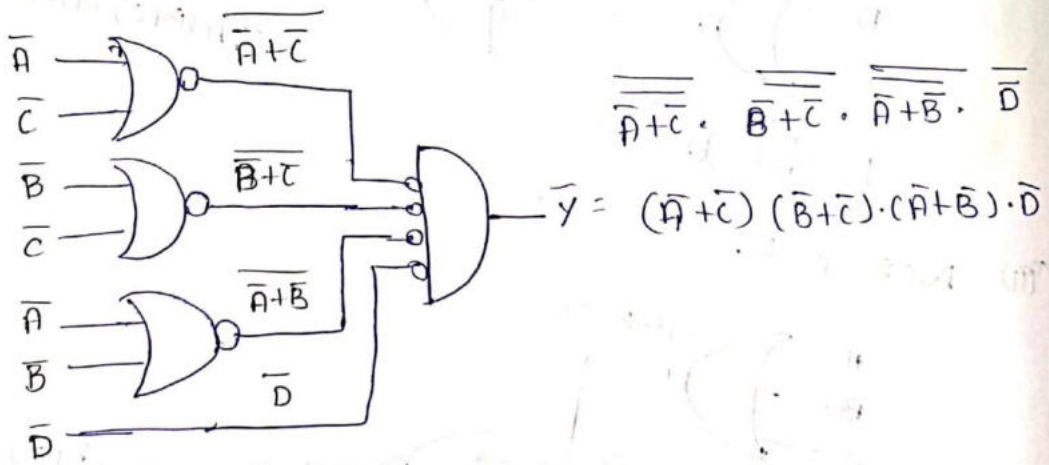
$$= (\overline{AC}) \cdot (\overline{BC}) \cdot (\overline{AB})\, \overline{D}$$

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

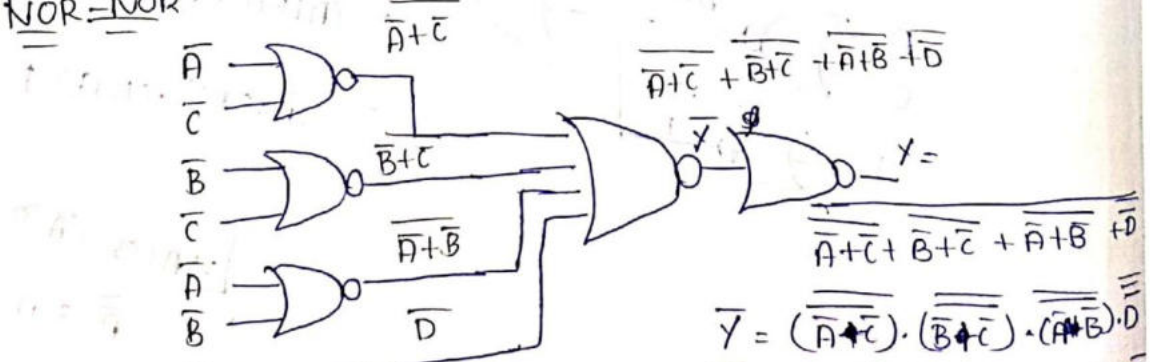
(i) OR-AND



(ii) NOR- Bubbled-AND



(iii) NOR-NOR



$\overline{Y} = (\bar{A}+\bar{C}).(\bar{B}+\bar{C}).(\bar{A}+B).\bar{D}$ $\quad [\because \overline{\bar{A}+\bar{B}} = \bar{\bar{A}}.\bar{\bar{B}} = AB]$

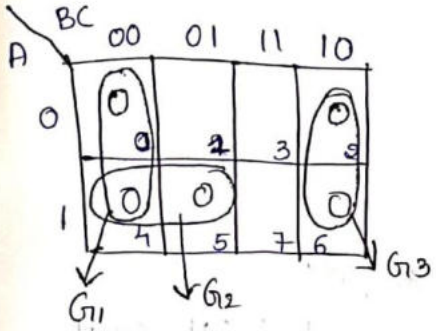
$Y = AC+BC+AB+D$

Ex:- Implement the following Boolean function with NOR-NOR logic

$f(A,B,C) = \Pi_M(0,2,4,5,6)$

Ex:- Implement the X-NOR gate by using only NOR gate $(AB+\bar{A}\bar{B})$

Ex:① $f(A,B,C) = \Pi_M(0,2,4,5,6)$



G1

A	B	C
0	0	0
1	0	0

$\bar{B}\bar{C}$

G2

A	B	C
1	0	0
1	0	1

$A\bar{B}$

G3

A	B	C
0	1	0
1	1	0

$B\bar{C}$

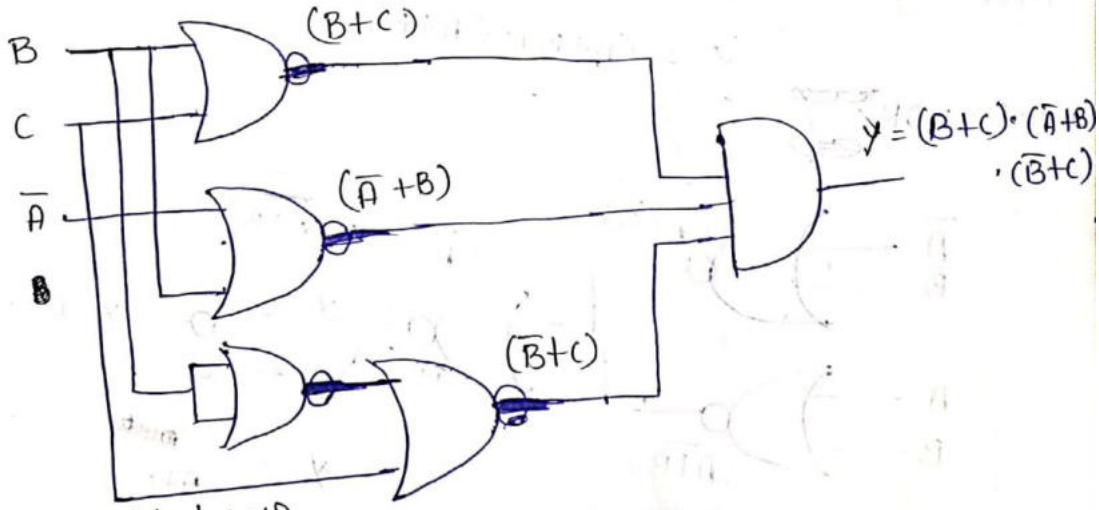
$\overline{Y} = \bar{B}\bar{C} + A\bar{B} + B\bar{C}$

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

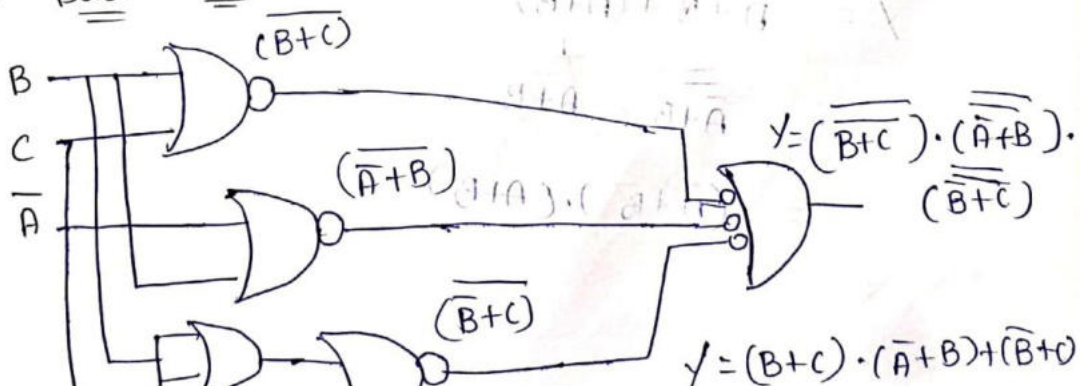
$= (\bar{\bar{B}} + \bar{\bar{C}}).(\bar{A}+\bar{\bar{B}}).(\bar{B}+\bar{\bar{C}})$

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

(i) OR - AND

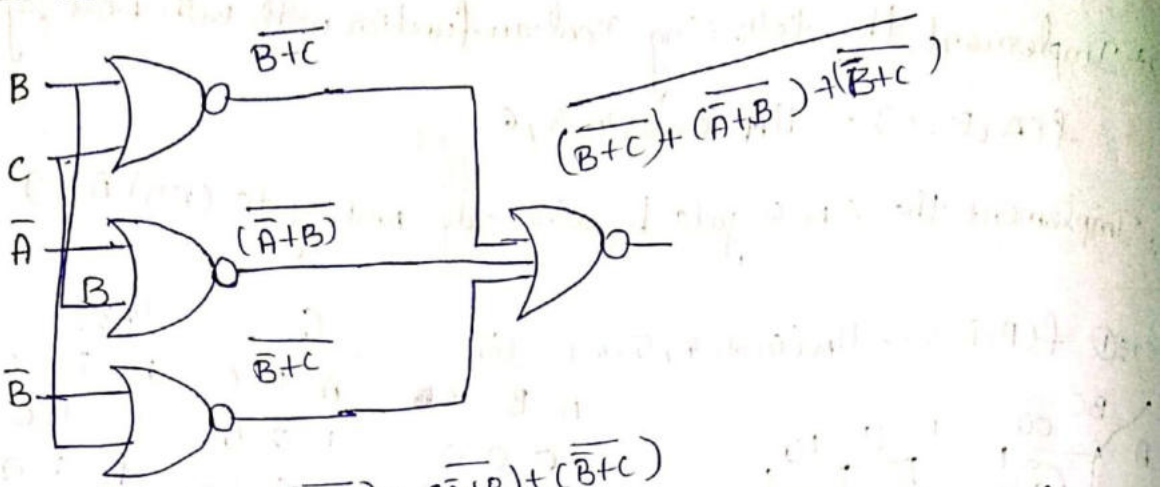


(ii) NOR - Bubbled AND



(iii) NOR-NOR

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



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

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

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

(2) Implement the ~~X-NOR~~ gate by using only NOR gate

The X-OR gate function is two inputs and one output. The input Variables are A and B and the output Variable is Y

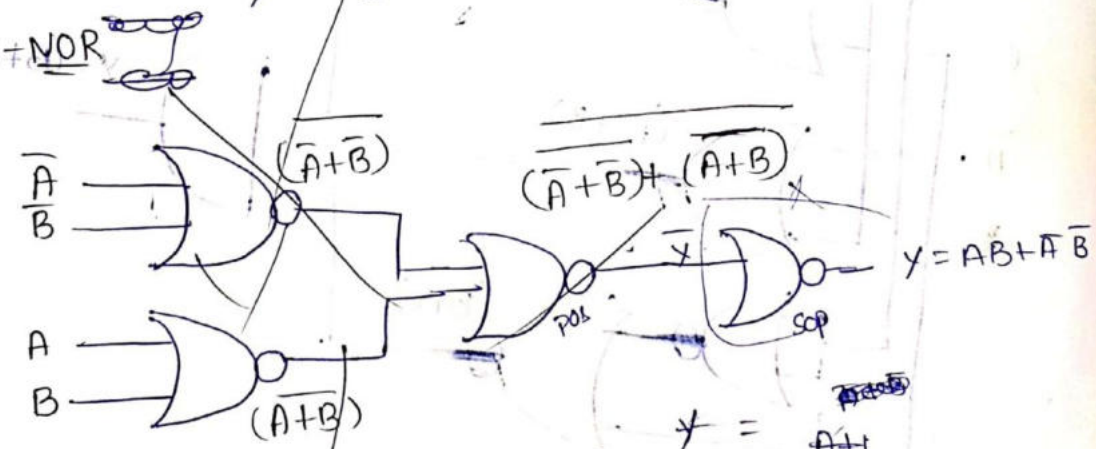
$Y = A\bar{B} + \bar{A}B$

~~NOR-NOR~~

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

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

NOR+NOR

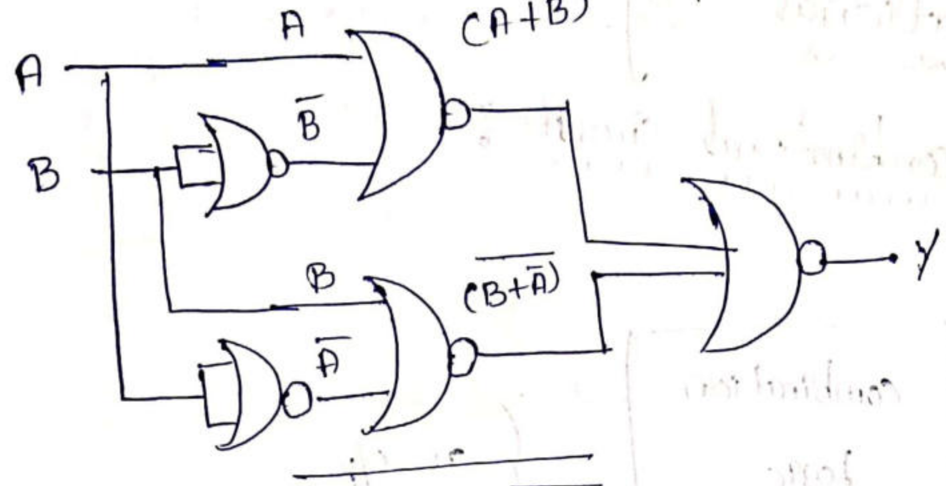


$\bar{Y} = \overline{\bar{A}+B} + \overline{(A+\bar{B})}$

$\overline{\bar{A}+B} \cdot \overline{A+\bar{B}}$

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

Ex: Implement the x-



$$y = \overline{\overline{(A+\bar{B})} + \overline{(B+\bar{A})}}$$

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

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

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

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

$\left[\because B\bar{B} = 0,\ A\bar{A} = 0\right]$

D: nsel: 1 i/p

M: 1: 8: 1

M.JOSHNA

M.JOSHNA

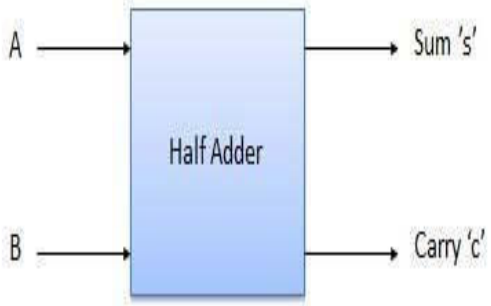
Combinational circuit is a circuit in which we combine the different gates in the circuit, for example encoder, decoder, multiplexer and demultiplexer. Some of the characteristics of combinational circuits are following −

1. The output of combinational circuit at any instant of time, depends only on the levels present at input terminals.
2. 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.
3. A combinational circuit can have an n number of inputs and m number of outputs.

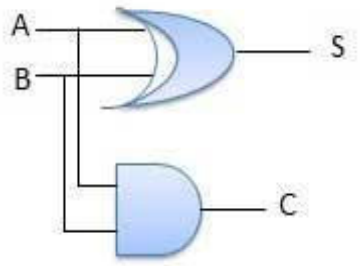
Block diagram

Half Adder

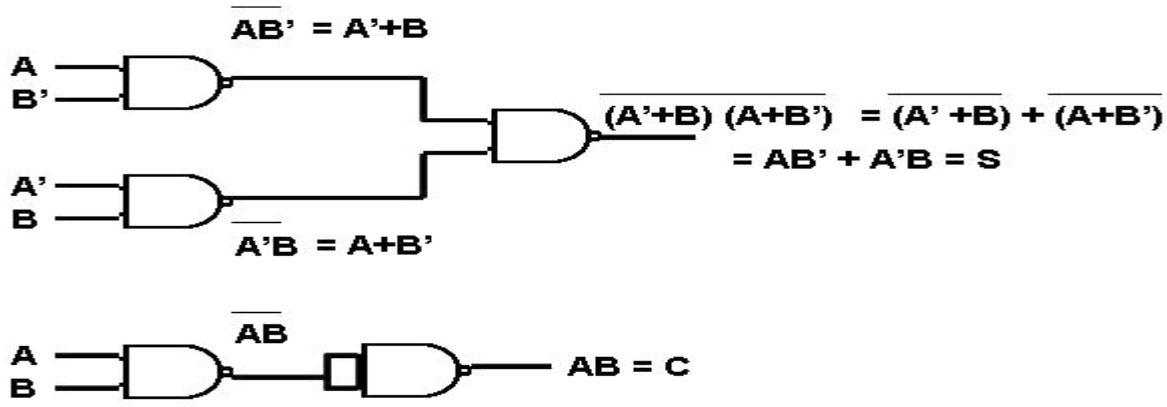
Half adder is a combinational logic circuit with two inputs and two outputs. The half adder circuit is designed to add two single bit binary number A and B. It is the basic building block for addition of two **single** bit numbers. This circuit has two outputs **carry** and **sum**. Block diagram Truth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



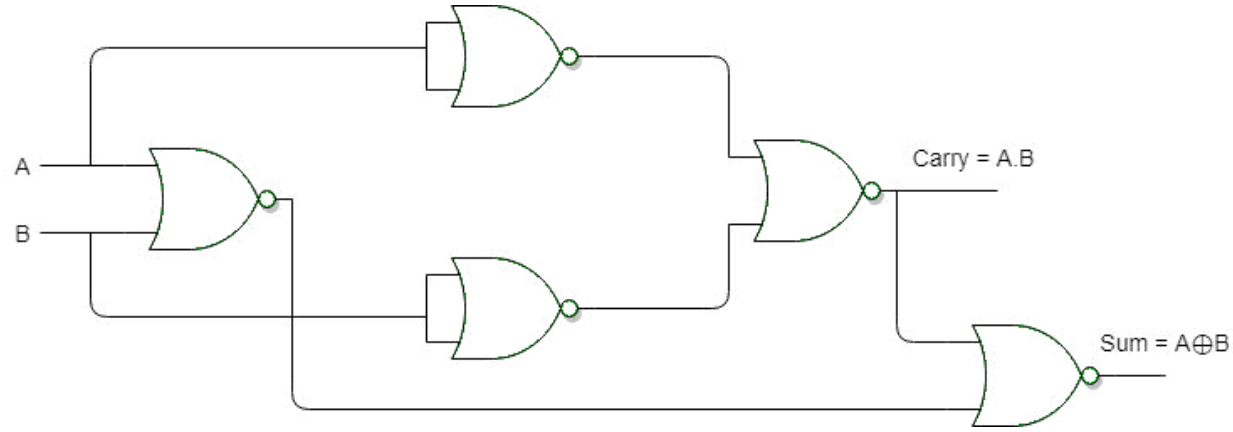
Inputs		Output	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Half adder using NAND gates



Half Adder using NOR gates

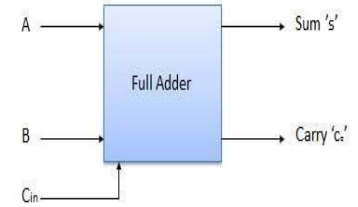


Full Adder

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.

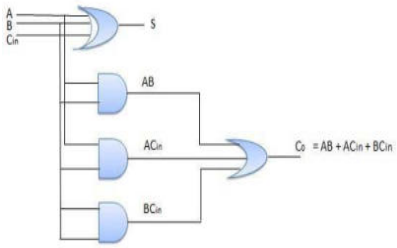
Block diagram Truth Table Circuit Diagram

M.JOSHNA



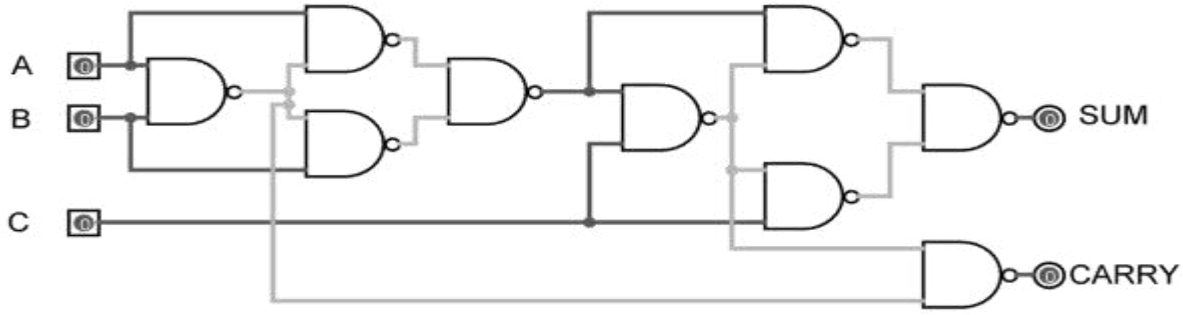
Inputs			Output	
A	B	Cin	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

S

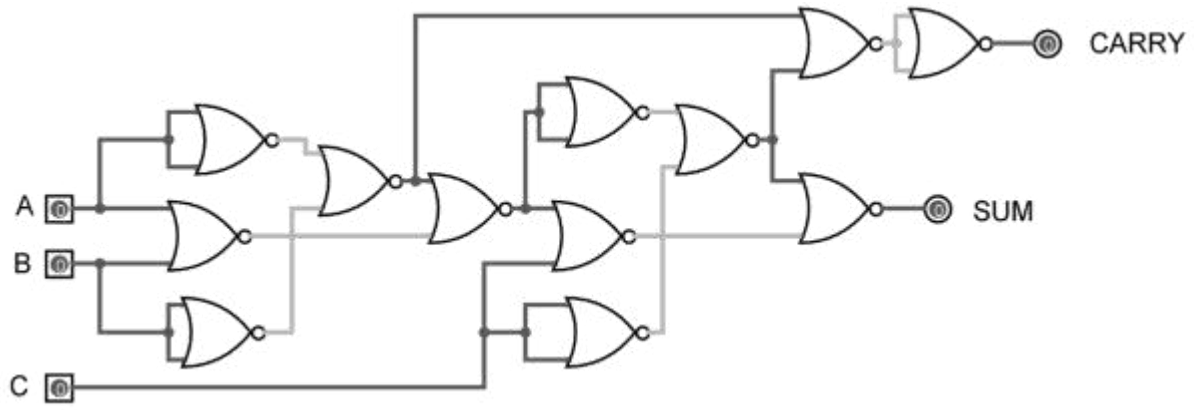


Full Adder using NAND gate

A NAND gate is one kind of universal gate, used to execute any kind of logic design. The FA circuit with the NAND gates diagram is shown below.



Full Adder using NOR gate

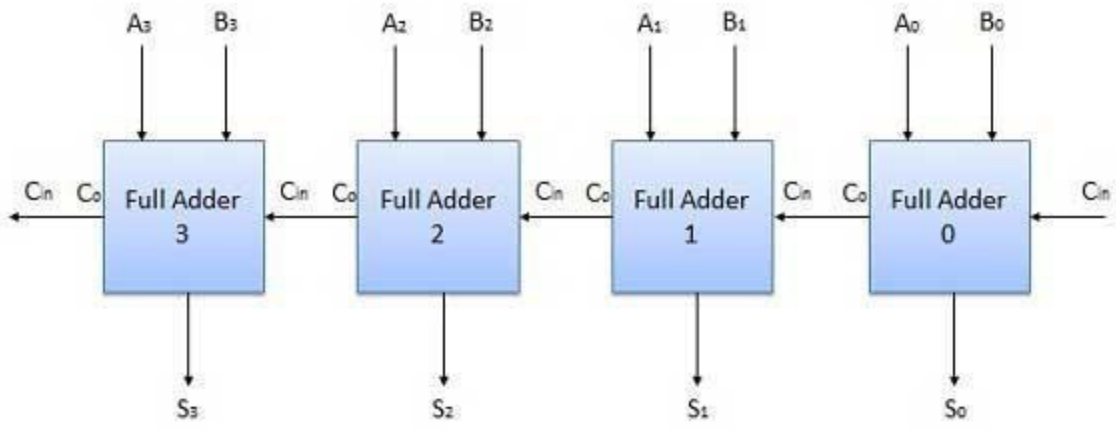


N-Bit Parallel Adder

The Full Adder is capable of adding only two single digit binary number along with a carry input. But in practical we need to add binary numbers which are much longer than just one bit. To add two n-bit binary numbers we need to use the n-bit parallel adder. It uses a number of full adders in cascade. The carry output of the previous full adder is connected to carry input of the next full adder. 4 Bit Parallel Adder

In the block diagram, A_0 and B_0 represent the LSB of the four bit words A and B. Hence Full Adder-0 is the lowest stage. Hence its C_{in} has been permanently made 0. The rest of the connections are exactly same as those of n-bit parallel adder is shown in fig. The four bit parallel adder is a very common logic circuit.

Block diagram



Combinational circuits consist of Logic gates. These circuits operate with binary values.

Binary Adder

The most basic arithmetic operation is addition. The circuit, which performs the addition of two binary numbers is known as **Binary adder**. First, let us implement an adder, which performs the addition of two bits.

Half Adder

Half adder is a combinational circuit, which performs the addition of two binary numbers A and B are of **single bit**. It produces two outputs sum, S & carry, C.

The **Truth table** of Half adder is shown below.

Inputs		Outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

When we do the addition of two bits, the resultant sum can have the values ranging from 0 to 2 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can't represent decimal digit 2 with single bit in binary. So, we require two bits for representing it in binary.

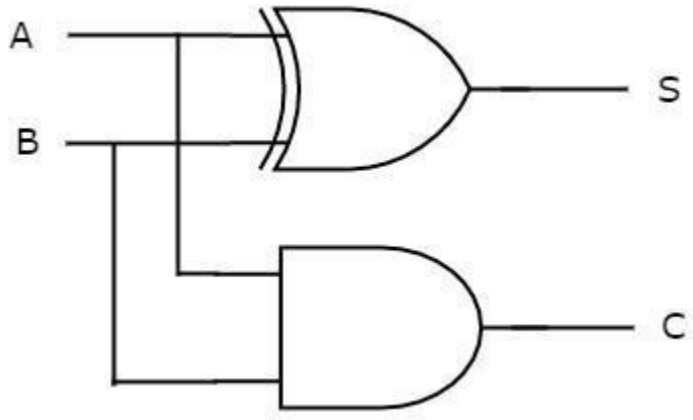
Let, sum, S is the Least significant bit and carry, C is the Most significant bit of the resultant sum. For first three combinations of inputs, carry, C is zero and the value of S will be either zero or one based on the **number of ones** present at the inputs. But, for last combination of inputs, carry, C is one and sum, S is zero, since the resultant sum is two.

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

$$S=A\oplus BS=A\oplus B$$

$$C=ABC=AB$$

We can implement the above functions with 2-input Ex-OR gate & 2-input AND gate. The **circuit diagram** of Half adder is shown in the following figure.



In the above circuit, a two input Ex-OR gate & two input AND gate produces sum, S & carry, C respectively. Therefore, Half-adder performs the addition of two bits. Full Adder

Full adder is a combinational circuit, which performs the **addition of three bits** A, B and C_{in}. Where, A & B are the two parallel significant bits and C_{in} is the carry bit, which is generated from previous stage. This Full adder also produces two outputs sum, S & carry, C_{out}, which are similar to Half adder.

The **Truth table** of Full adder is shown below.

Inputs			Outputs	
A	**B**	**C_{in}**	**C_{out}**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

When we do the addition of three bits, the resultant sum can have the values ranging from 0 to 3 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can't represent the decimal digits 2 and 3 with single bit in binary. So, we require two bits for representing those two decimal digits in binary.

Let, sum, S is the Least significant bit and carry, C_{out} is the Most significant bit of resultant sum. It is easy to fill the values of outputs for all combinations of inputs in the truth table. Just count the **number of ones** present at the inputs and write the equivalent binary number at outputs. If C_{in} is equal to zero, then Full adder truth table is same as that of Half adder truth table.

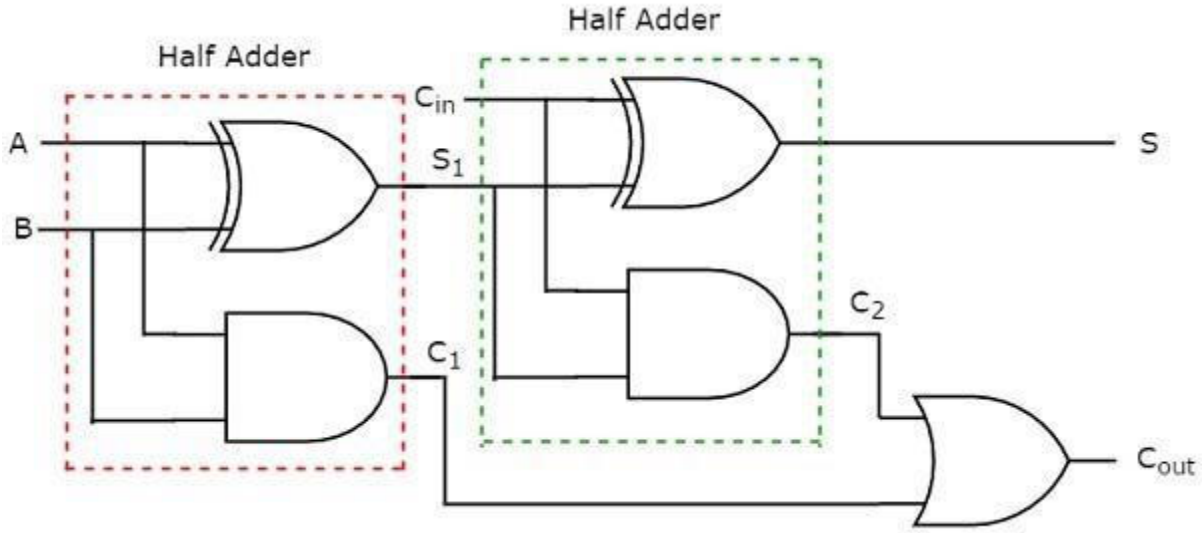
We will get the following **Boolean functions** for each output after simplification.

$$S=A \oplus B \oplus C_{in} S=A \oplus B \oplus Cin$$

$$c_{out}=AB+(A \oplus B)c_{in} cout=AB+(A \oplus B)cin$$

The sum, S is equal to one, when odd number of ones present at the inputs. We know that Ex-OR gate produces an output, which is an odd function. So, we can use either

two 2input Ex-OR gates or one 3-input Ex-OR gate in order to produce sum, S. We can implement carry, C_{out} using two 2-input AND gates & one OR gate. The **circuit diagram** of Full adder is shown in the following figure.



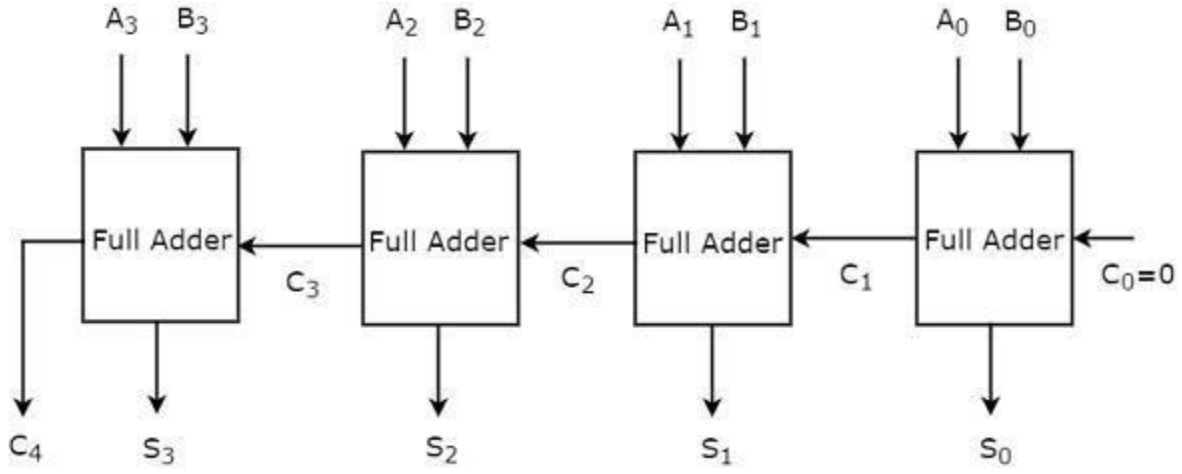
This adder is called as **Full adder** because for implementing one Full adder, we require two Half adders and one OR gate. If C_{in} is zero, then Full adder becomes Half adder. We can verify it easily from the above circuit diagram or from the Boolean functions of outputs of Full adder.

4-bit Binary Adder

The 4-bit binary adder performs the **addition of two 4-bit numbers**. Let the 4-bit binary numbers, $A=A_3A_2A_1A_0$A=A3A2A1A0 and $B=B_3B_2B_1B_0$B=B3B2B1B0. We can implement 4-bit binary adder in one of the two following ways.

1. Use one Half adder for doing the addition of two Least significant bits and three Full adders for doing the addition of three higher significant bits.
2. Use four Full adders for uniformity. Since, initial carry C_{in} is zero, the Full adder which is used for adding the least significant bits becomes Half adder.

For the time being, we considered second approach. The **block diagram** of 4-bit binary adder is shown in the following figure.



Here, the 4 Full adders are cascaded. Each Full adder is getting the respective bits of two parallel inputs A & B. The carry output of one Full adder will be the carry input of subsequent higher order Full adder. This 4-bit binary adder produces the resultant sum having at most 5 bits. So, carry out of last stage Full adder will be the MSB.

In this way, we can implement any higher order binary adder just by cascading the required number of Full adders. This binary adder is also called as **ripple carry** binarybinary **adder** because the carry propagates ripplesripples from one stage to the next stage.

Binary Subtractor

The circuit, which performs the subtraction of two binary numbers is known as **Binary subtractor**. We can implement Binary subtractor in following two methods.

1. Cascade Full subtractors
2. 2's complement method

In first method, we will get an n-bit binary subtractor by cascading 'n' Full subtractors. So, first you can implement Half subtractor and Full subtractor, similar to Half adder & Full adder. Then, you can implement an n-bit binary subtractor, by cascading 'n' Full subtractors. So, we will be having two separate circuits for binary addition and subtraction of two binary numbers.

In second method, we can use same binary adder for subtracting two binary numbers just by doing some modifications in the second input. So, internally binary addition operation takes place but, the output is resultant subtraction.

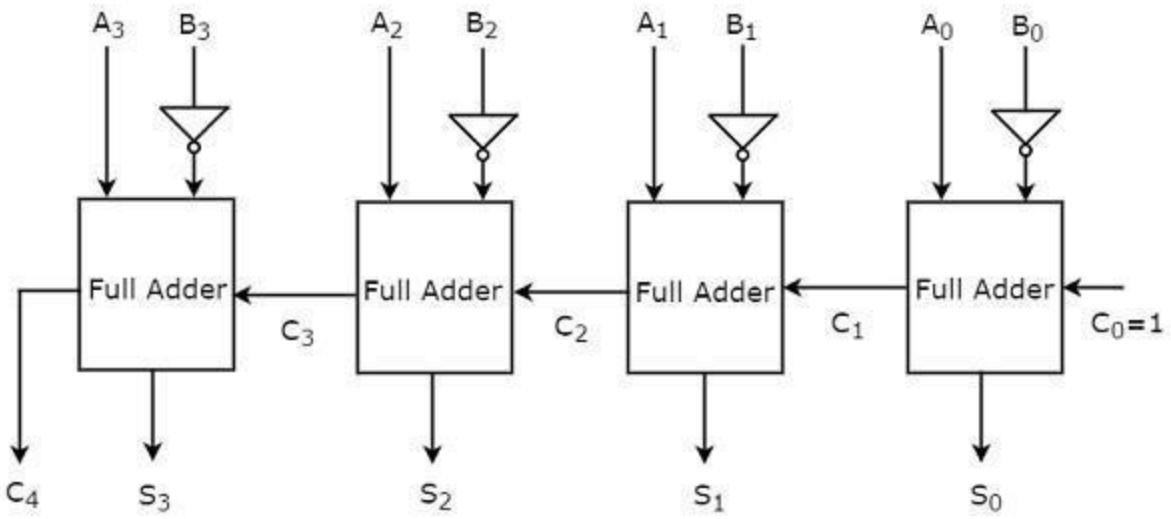
We know that the subtraction of two binary numbers A & B can be written as,

$$A-B=A+(2's complimentofB)A-B=A+(2'scomplimentofB)$$

$$\Rightarrow A-B=A+(1's complimentofB)+1\Rightarrow A-B=A+(1'scomplimentofB)+1$$

4-bit Binary Subtractor

The 4-bit binary subtractor produces the **subtraction of two 4-bit numbers**. Let the 4bit binary numbers, $A=A_3A_2A_1A_0A=A3A2A1A0$ and $B=B_3B_2B_1B_0B=B3B2B1B0$. Internally, the operation of 4-bit Binary subtractor is similar to that of 4-bit Binary adder. If the normal bits of binary number A, complemented bits of binary number B and initial carry borrowborrow, C_{in} as one are applied to 4-bit Binary adder, then it becomes 4-bit Binary subtractor. The **block diagram** of 4-bit binary subtractor is shown in the following figure.



This 4-bit binary subtractor produces an output, which is having at most 5 bits. If Binary number A is greater than Binary number B, then MSB of the output is zero and the remaining bits hold the magnitude of A-B. If Binary number A is less than Binary number B, then MSB of the output is one. So, take the 2's complement of output in order to get the magnitude of A-B.

In this way, we can implement any higher order binary subtractor just by cascading the required number of Full adders with necessary modifications.

Binary Adder / Subtractor

The circuit, which can be used to perform either addition or subtraction of two binary numbers at any time is known as **Binary Adder / subtractor**. Both, Binary adder and Binary subtractor contain a set of Full adders, which are cascaded. The input bits of binary number A are directly applied in both Binary adder and Binary subtractor.

There are two differences in the inputs of Full adders that are present in Binary adder and Binary subtractor.

1. The input bits of binary number B are directly applied to Full adders in Binary adder, whereas the complemented bits of binary number B are applied to Full adders in Binary subtractor.
2. The initial carry, $C_0 = 0$ is applied in 4-bit Binary adder, whereas the initial carry borrowborrow, $C_0 = 1$ is applied in 4-bit Binary subtractor.

We know that a **2-input Ex-OR gate** produces an output, which is same as that of first input when other input is zero. Similarly, it produces an output, which is complement of first input when other input is one.

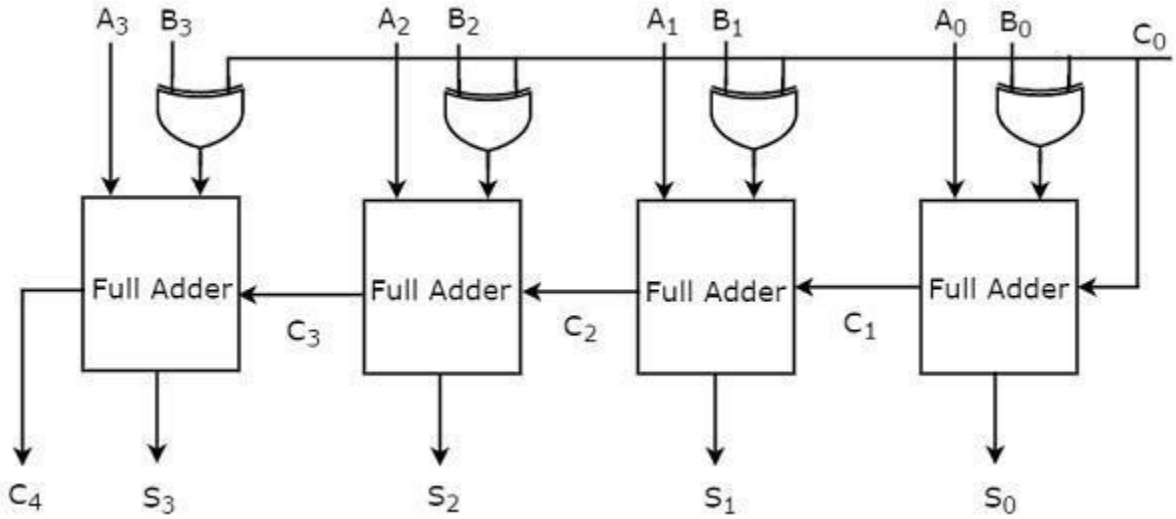
Therefore, we can apply the input bits of binary number B, to 2-input Ex-OR gates. The other input to all these Ex-OR gates is C_0. So, based on the value of C_0, the Ex-OR gates produce either the normal or complemented bits of binary number B.

4-bit Binary Adder / Subtractor

The 4-bit binary adder / subtractor produces either the addition or the subtraction of two 4-bit numbers based on the value of initial carry or borrow, C_0. Let the 4-bit binary

numbers, $A=A_3A_2A_1A_0$A=A3A2A1A0 and $B=B_3B_2B_1B_0$B=B3B2B1B0. The operation of 4-bit Binary adder / subtractor is similar to that of 4-bit Binary adder and 4-bit Binary subtractor.

Apply the normal bits of binary numbers A and B & initial carry or borrow, C_0 from externally to a 4-bit binary adder. The **block diagram** of 4-bit binary adder / subtractor is shown in the following figure.



If initial carry, C_0 is zero, then each full adder gets the normal bits of binary numbers A & B. So, the 4-bit binary adder / subtractor produces an output, which is the **addition of two binary numbers** A & B.

If initial borrow, C_0 is one, then each full adder gets the normal bits of binary number A & complemented bits of binary number B. So, the 4-bit binary adder / subtractor produces an output, which is the **subtraction of two binary numbers** A & B.

Therefore, with the help of additional Ex-OR gates, the same circuit can be used for both addition and subtraction of two binary numbers.

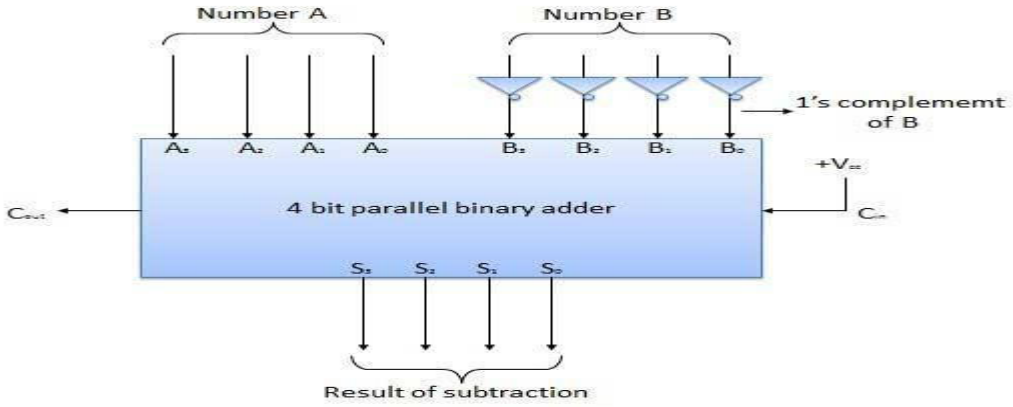
N-Bit Parallel Subtractor

The subtraction can be carried out by taking the 1's or 2's complement of the number to be subtracted. For example we can perform the subtraction (A-B) by adding either 1's or 2's complement of B to A. That means we can use a binary adder to perform the binary subtraction.

4 Bit Parallel Subtractor

The number to be subtracted (B) is first passed through inverters to obtain its 1's complement. The 4-bit adder then adds A and 2's complement of B to produce the subtraction. S_3 S_2 S_1 S_0 represents the result of binary subtraction (A-B) and carry output C_{out} represents the polarity of the result. If A > B then Cout = 0 and the result of binary form (A-B) then C_{out} = 1 and the result is in the 2's complement form.

Block diagram



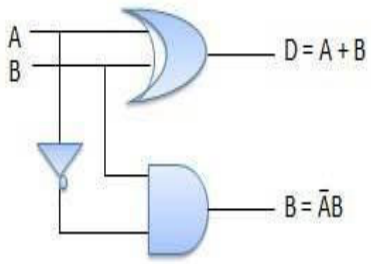
Half Subtractors

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

Truth Table

Inputs		Output	
A	B	(A – B)	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Circuit Diagram



Full Subtractors

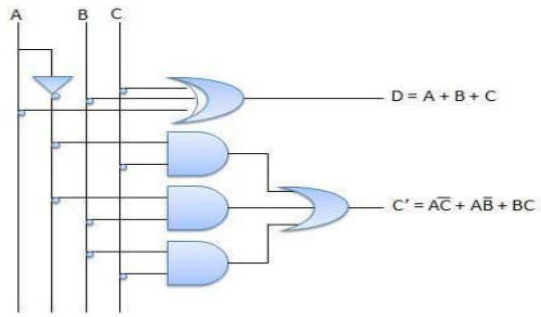
The disadvantage of a half subtractor is overcome by full subtractor. The full subtractor is a combinational circuit with three inputs A,B,C and two output D and C'. A is the 'minuend', B is 'subtrahend', C is the 'borrow' produced by the previous stage, D is the difference output and C' is the borrow output.

M.JOSHNA

Truth Table

Inputs			Output	
A	B	C	(A-B-C)	C'
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Circuit Diagram

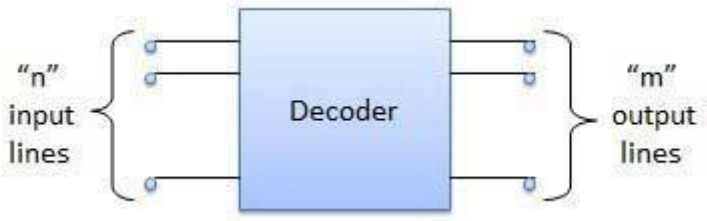


Decoder is a combinational circuit that has 'n' input lines and maximum of 2^n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing but the **min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

Decoder

A decoder is a combinational circuit. It has n input and to a maximum m = 2n outputs. Decoder is identical to a demultiplexer without any data input. It performs operations which are exactly opposite to those of an encoder.

Block diagram



Examples of Decoders are following.

1. Code converters
2. BCD to seven segment decoders

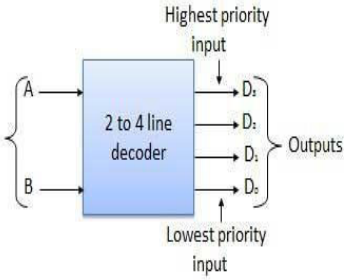
3. Nixie tube decoders
4. Relay actuator

2 to 4 Line Decoder

The 2-to-4 line binary decoder **consists of an array of four AND gates. The 2 binary inputs labelled A and B are decoded into one of 4 outputs**, hence the description of 2-to-4 binary decoder. Each output represents one of the minterms of the 2 input variables,

The block diagram of 2 to 4 line decoder is shown in the fig. A and B are the two inputs where D through D are the four outputs. Truth table explains the operations of a decoder. It shows that each output is 1 for only a specific combination of inputs.

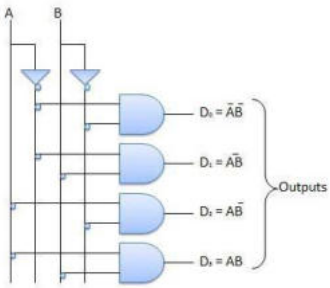
Block diagram



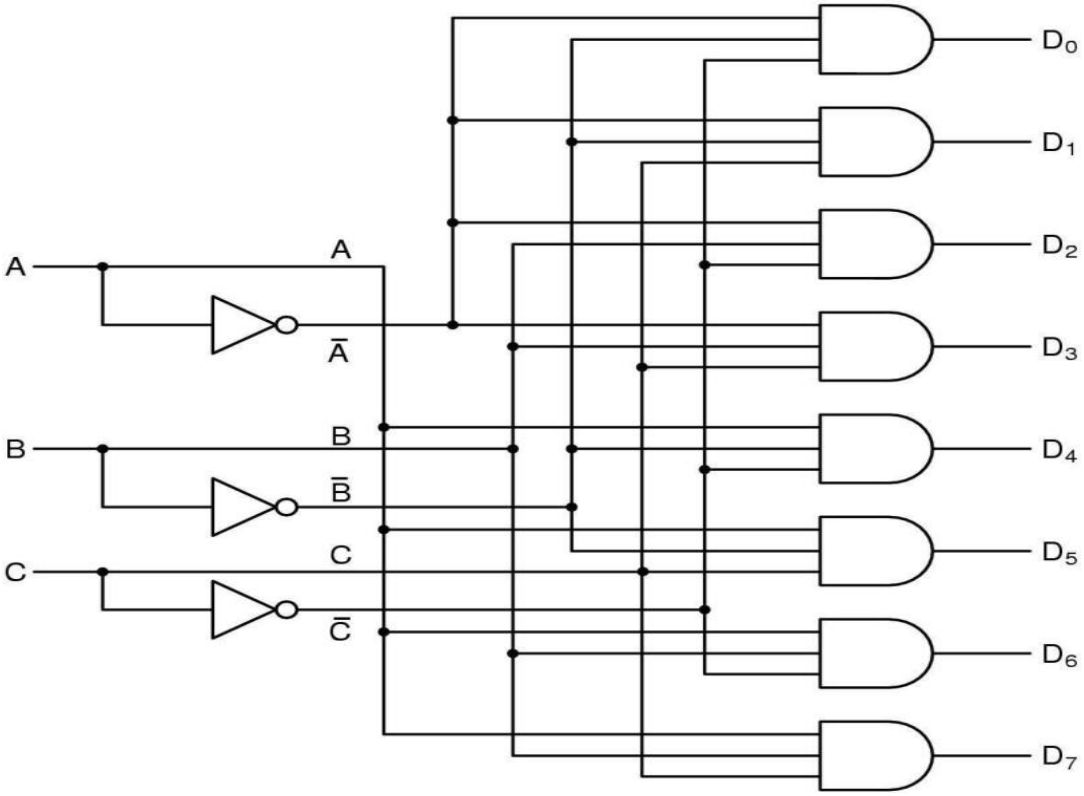
Truth Table

Inputs		Output			
A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
0	1	0	0	1	0
1	1	0	0	0	1

Logic Circuit



3to8 line Decoder



How do you implement a half adder using a 2to4 line decoder?

By connecting an OR gate with output line 1 & 2 of 2X4 Decoder. Half Adder can be implemented with 2X4 decoder. Similarly by connecting two Half Adders, we can form a Full Adder by using 2, 2X4 Decoder

Full Adder implementation using 2to8 decoder

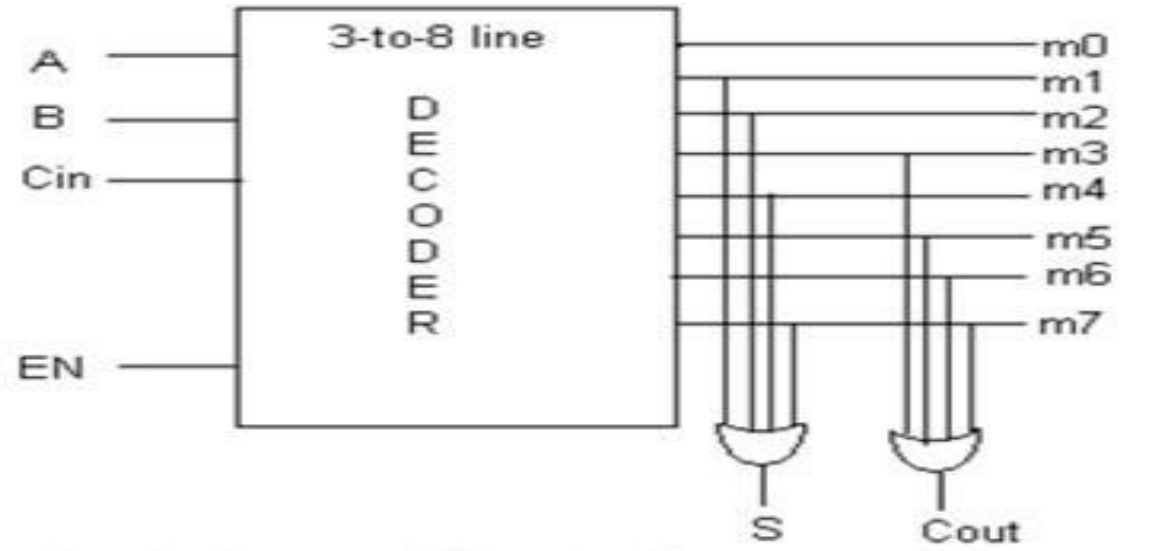
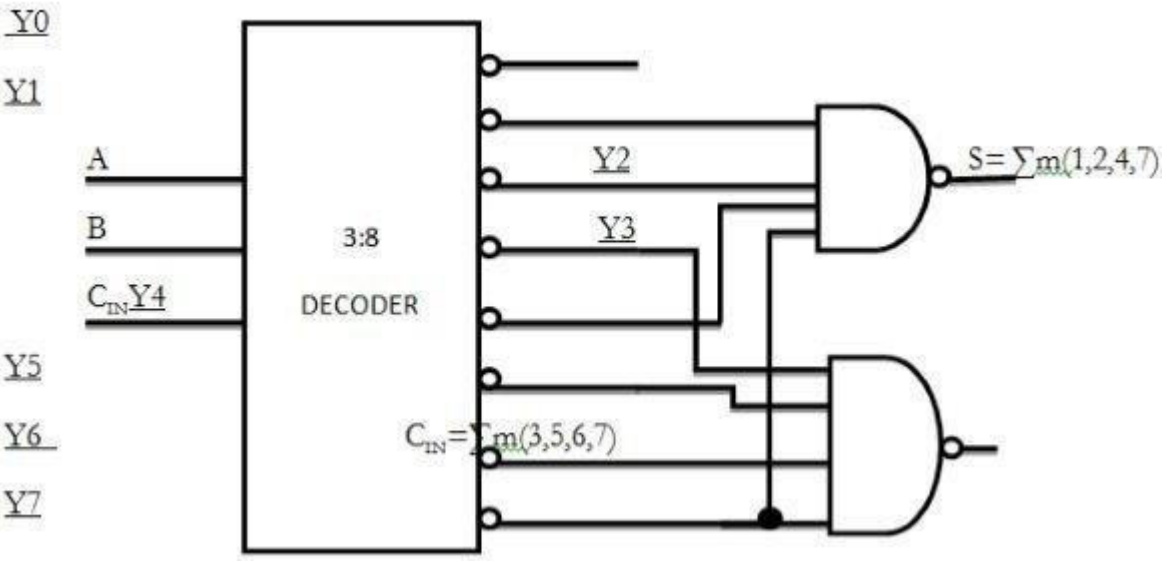


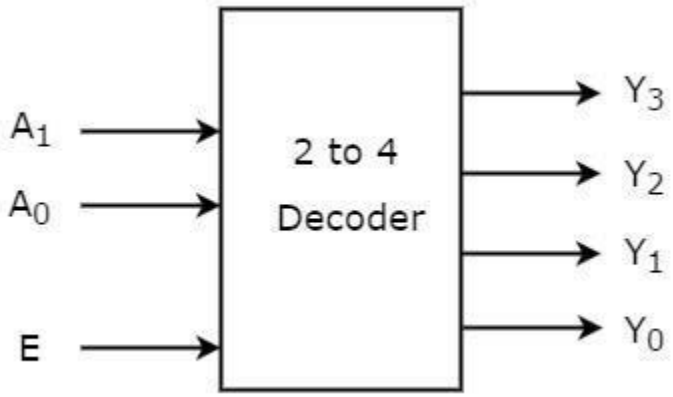
Fig1: Full Adder Implementation using 3:8 decoder

Full adder using decoder and NAND gates



2 to 4 Decoder

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



One of these four outputs will be '1' for each combination of inputs when enable, E is '1'. The **Truth table** of 2 to 4 decoder is shown below.

Enable	Inputs		Outputs			
E	**A_1**	**A_0**	**Y_3**	**Y_2**	**Y_1**	**Y_0**
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

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

$$Y_3=E.A_1.A_0 Y3=E.A1.A0$$

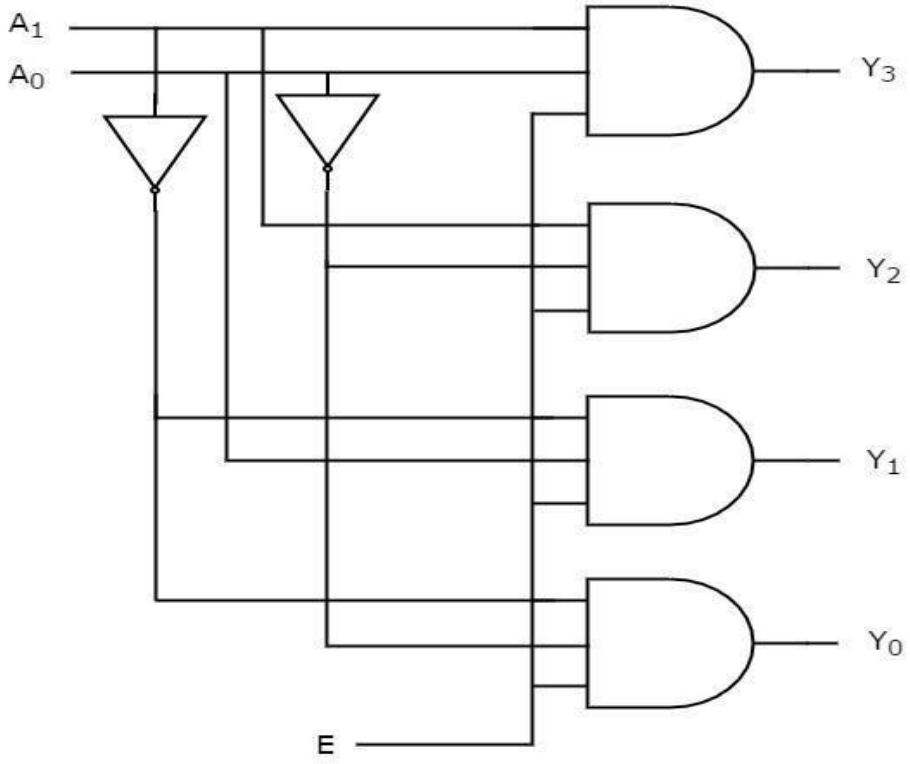
$$Y_2=E.A_1.A_{0'} Y2=E.A1.A0'$$

$$Y_1=E.A_{1'}.A_0 Y1=E.A1'.A0$$

$$Y_0=E.A_{1'}.A_{0'} Y0=E.A1'.A0'$$

Each output is having one product term. So, there are four product terms in total. We can implement these four product terms by using four AND gates having three inputs

each & two inverters. The **circuit diagram** of 2 to 4 decoder is shown in the following figure.



Therefore, the outputs of 2 to 4 decoder are nothing but the **min terms** of two input variables A_1 & A_0, when enable, E is equal to one. If enable, E is zero, then all the outputs of decoder will be equal to zero.

Similarly, 3 to 8 decoder produces eight min terms of three input variables A_2, A_1 & A_0 and 4 to 16 decoder produces sixteen min terms of four input variables A_3, A_2, A_1 & A_0.

Implementation of Higher-order Decoders

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

1. 3 to 8 decoder
2. 4 to 16 decoder

3 to 8 Decoder

In this section, let us implement **3 to 8 decoder using 2 to 4 decoders**. We know that 2 to 4 Decoder has two inputs, A_1 & A_0 and four outputs, Y_3 to Y_0. Whereas, 3 to 8 Decoder has three inputs A_2, A_1 & A_0 and eight outputs, Y_7 to Y_0.

We can find the number of lower order decoders required for implementing higher order decoder using the following formula.

Requirednumberoflowerorderdecoders=m2m1Requirednumberoflowerorderdecoders=m

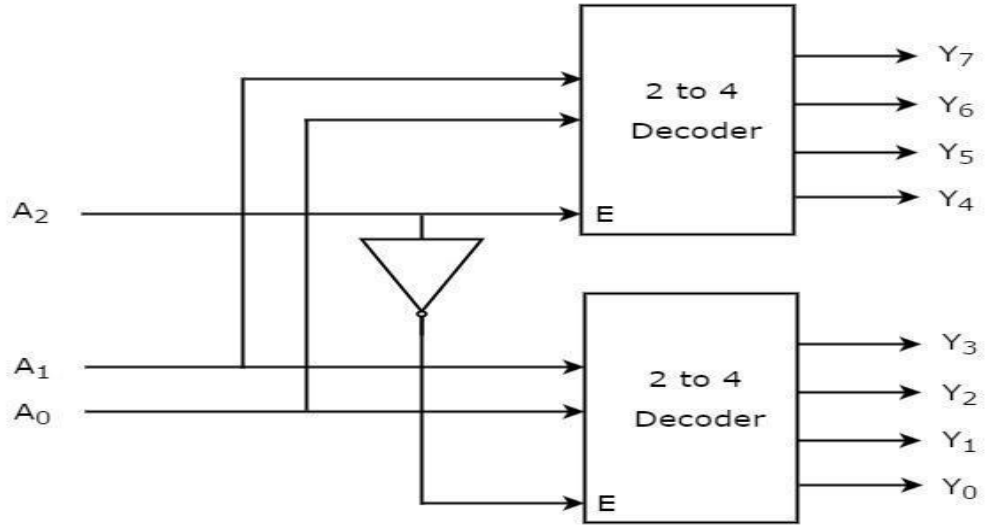
2m1

Where,

m1m1 is the number of outputs of lower order decoder. m2m2 is the number of outputs of higher order decoder. Here, m1m1 = 4 and m2m2 = 8.
Substitute, these two values in the above formula.

Requirednumberof2to4decoders=84=2Requirednumberof2to4decoders=84=2

Therefore, we require two 2 to 4 decoders for implementing one 3 to 8 decoder. The **block diagram** of 3 to 8 decoder using 2 to 4 decoders is shown in the following figure.



The parallel inputs A_1 & A_0 are applied to each 2 to 4 decoder. The complement of input A_2 is connected to Enable, E of lower 2 to 4 decoder in order to get the outputs, Y_3 to Y_0. These are the **lower four min terms**. The input, A_2 is directly connected to Enable, E of upper 2 to 4 decoder in order to get the outputs, Y_7 to Y_4. These are the **higher four min terms**.

4 to 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.

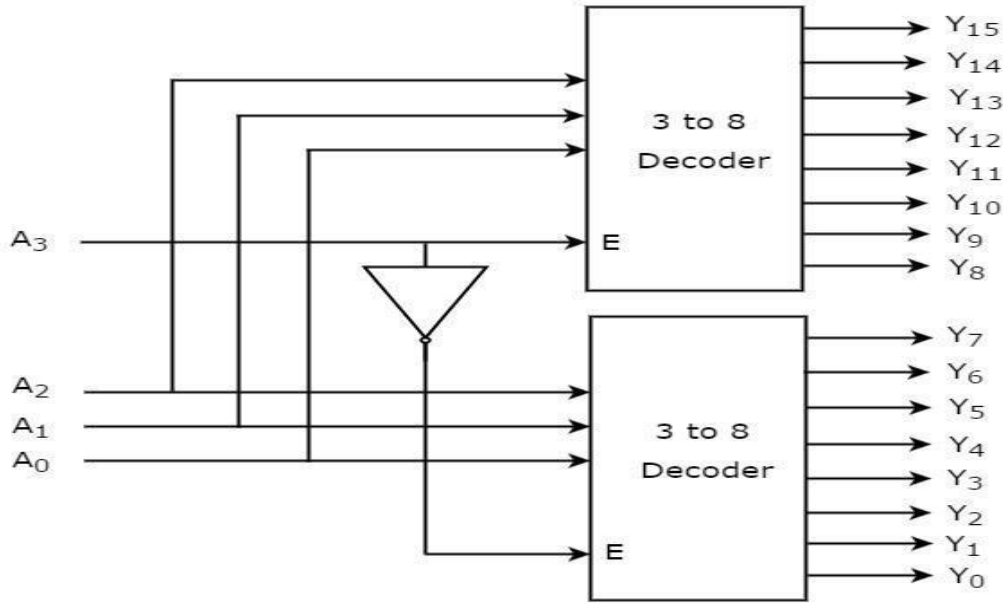
Requirednumberoflowerorderdecoders=m2m1Requirednumberoflowerorderdecoders=m

2m1

Substitute, m1m1 = 8 and m2m2 = 16 in the above formula.

Requirednumberof3to8decoders=168=2Requirednumberof3to8decoders=168=2

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.

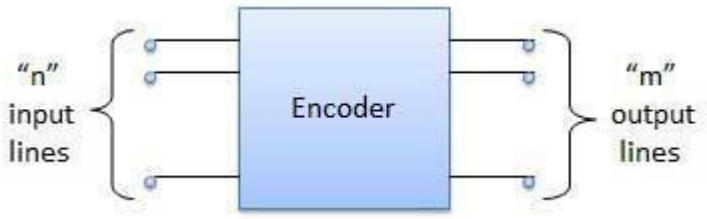


The parallel inputs A_2, A_1 & A_0 are applied to each 3 to 8 decoder. The complement of input, A3 is connected to Enable, E of lower 3 to 8 decoder in order to get the outputs, Y_7 to Y_0. These are the **lower eight min terms**. The input, A_3 is directly connected to Enable, E of upper 3 to 8 decoder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

Encoder

Encoder is a combinational circuit which is designed to perform the inverse operation of the decoder. An encoder has n number of input lines and m number of output lines. An encoder produces an m bit binary code corresponding to the digital input number. The encoder accepts an n input digital word and converts it into an m bit another digital word.

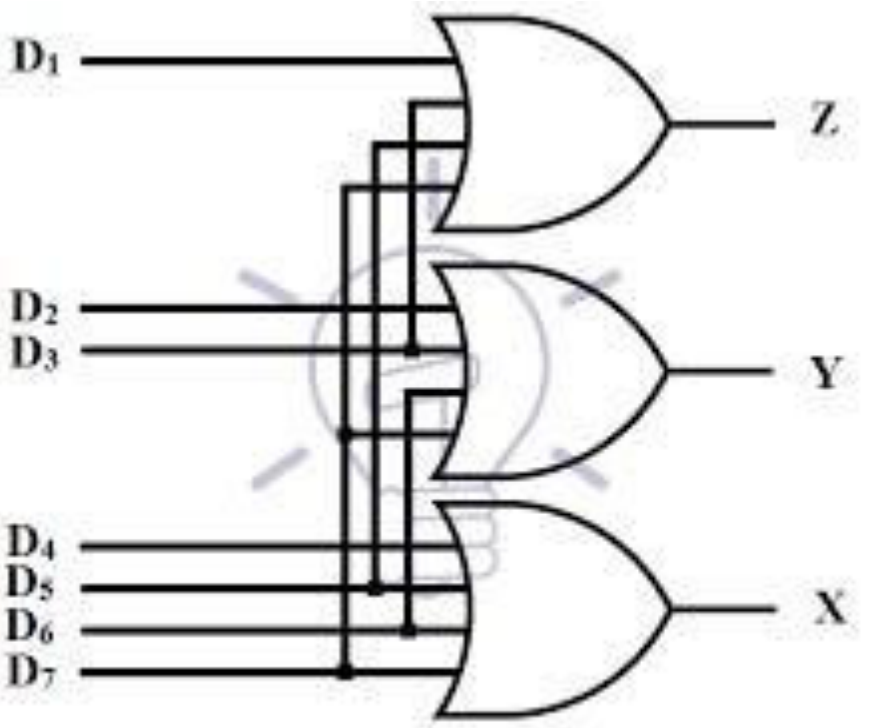
Block diagram



Examples of Encoders are following.

1. Priority encoders

2. Decimal to BCD encoder
3. Octal to binary encoder
4. Hexadecimal to binary encoder

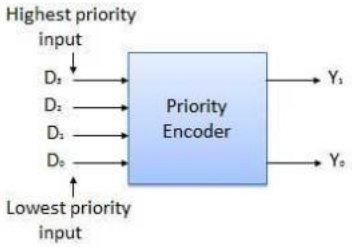


8 to 3 Line Encoder Using OR Gate

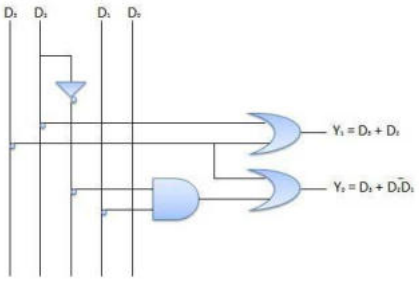
Priority Encoder

This is a special type of encoder. Priority is given to the input lines. If two or more input line are 1 at the same time, then the input line with highest priority will be considered. There are four input D_0, D_1, D_2, D_3 and two output Y_0, Y_1. Out of the four input D_3 has the highest priority and D_0 has the lowest priority. That means if $D_3 = 1$ then Y_1 $Y_1 = 11$ irrespective of the other inputs. Similarly if $D_3 = 0$ and $D_2 = 1$ then Y_1 $Y_0 = 10$ irrespective of the other inputs.

Block diagram Truth Table Logic Circuit



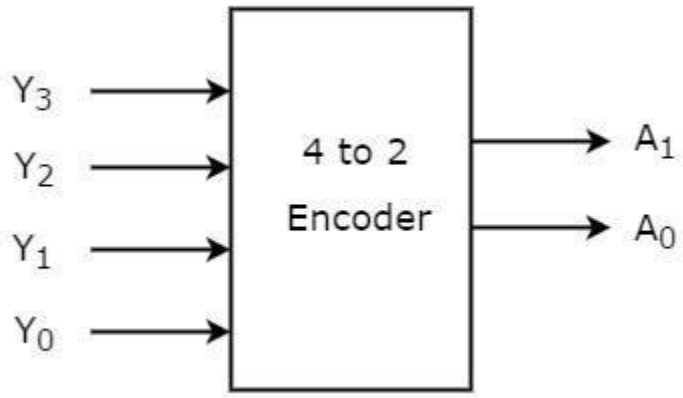
Highest	Inputs		Lowest	Outputs	
D_3	D_2	D_1	D_0	Y_0	Y_1
0	0	0	0	x	x
0	0	0	1	0	0
0	0	1	x	0	1
0	1	x	x	1	0
1	x	x	x	1	1



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

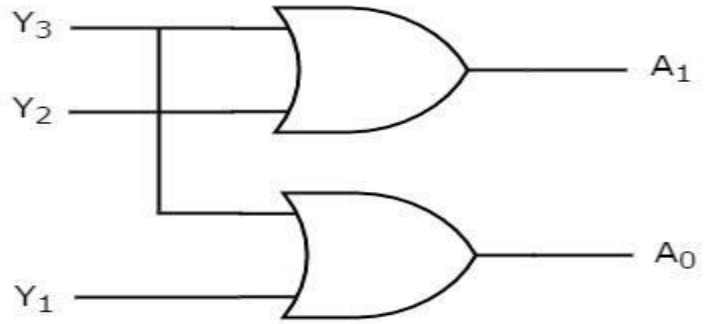
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

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

$$A_1=Y_3+Y_2 A1=Y3+Y2$$

$$A_0=Y_3+Y_1 A0=Y3+Y1$$

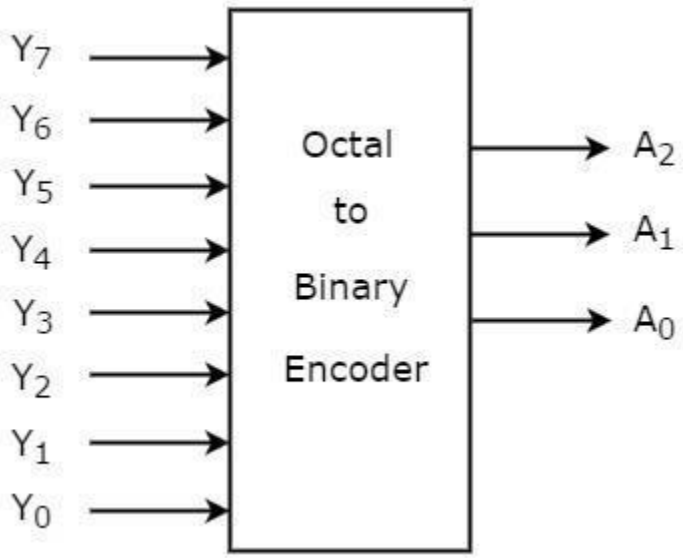
We can implement the above two Boolean functions by using two input OR gates. The **circuit diagram** of 4 to 2 encoder is shown in the following figure.



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

Octal to Binary Encoder

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



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

Inputs								Outputs		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

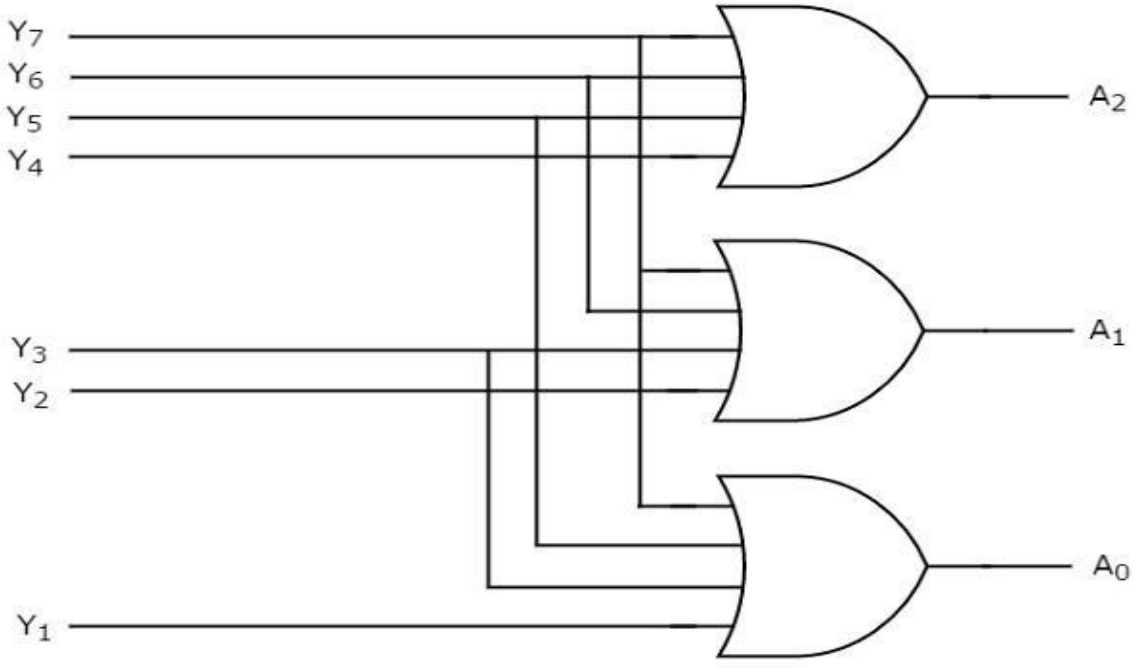
From Truth table, we can write the **Boolean functions** for each output as

$$A_2=Y_7+Y_6+Y_5+Y_4A2=Y7+Y6+Y5+Y4$$

$$A_1=Y_7+Y_6+Y_3+Y_2A1=Y7+Y6+Y3+Y2$$

$$A_0=Y_7+Y_5+Y_3+Y_1A0=Y7+Y5+Y3+Y1$$

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



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

Drawbacks of Encoder

Following are the drawbacks of normal encoder.

1. There is an ambiguity, when all outputs of encoder are equal to zero. Because, it could be the code corresponding to the inputs, when only least significant input is one or when all inputs are zero.
2. If more than one input is active High, then the encoder produces an output, which may not be the correct code. For **example**, if both Y_3 and Y_6 are '1', then the encoder produces 111 at the output. This is neither equivalent code corresponding to Y_3, when it is '1' nor the equivalent code corresponding to Y_6, when it is '1'.

So, to overcome these difficulties, we should assign priorities to each input of encoder. Then, the output of encoder will be the binarybinary code corresponding to the active High inputss, which has higher priority. This encoder is called as **priority encoder**.

Priority Encoder

A 4 to 2 priority encoder has four inputs Y_3, Y_2, Y_1 & Y_0 and two outputs A_1 & A_0. Here, the input, Y_3 has the highest priority, whereas the input, Y_0 has the lowest priority. In this case, even if more than one input is '1' at the same time, the output will be the binarybinary code corresponding to the input, which is having **higher priority**.

We considered one more **output, V** in order to know, whether the code available at outputs is valid or not.

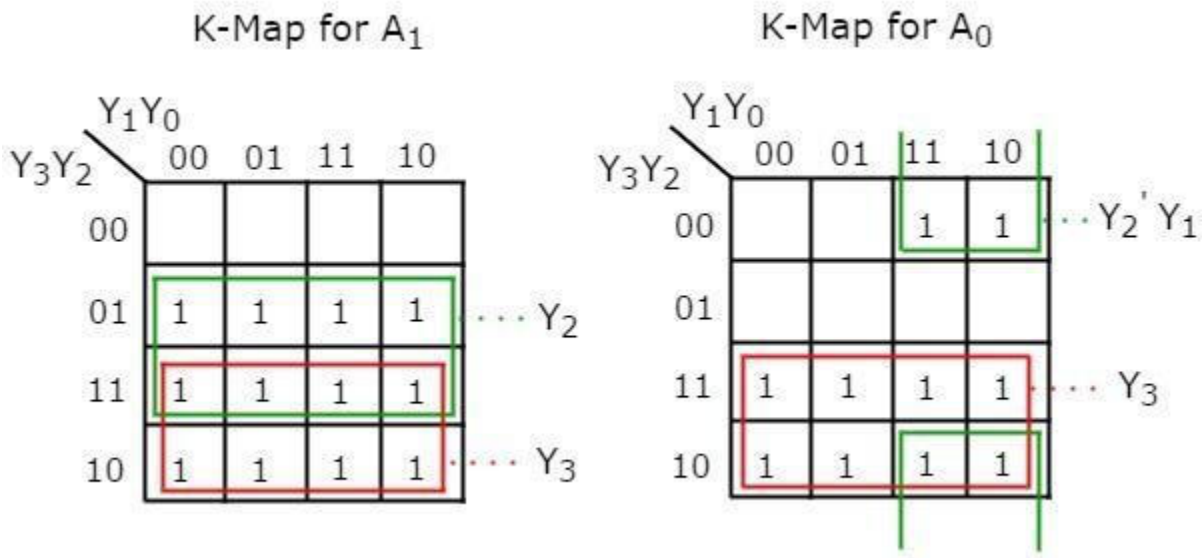
1. If at least one input of the encoder is '1', then the code available at outputs is a valid one. In this case, the output, V will be equal to 1.
2. If all the inputs of encoder are '0', then the code available at outputs is not a valid one. In this case, the output, V will be equal to 0.

The **Truth table** of 4 to 2 priority encoder is shown below.

Inputs				**Outputs**		
Y_3	**Y_2**	**Y_1**	**Y_0**	**A_1**	**A_0**	**V**
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1

1	x	x	x	1	1	1

Use **4 variable K-maps** for getting simplified expressions for each output.



The simplified **Boolean functions** are

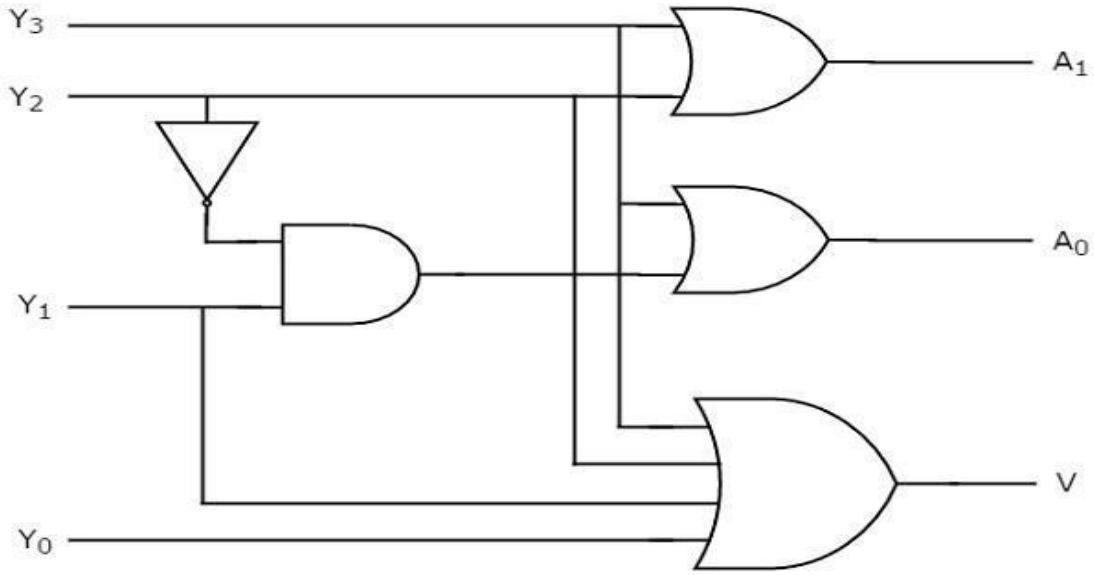
$$A_1=Y_3+Y_2 A1=Y3+Y2$$

$A_0=Y_3+Y_2{}'Y_1$ A0=Y3+Y2′Y1

Similarly, we will get the Boolean function of output, V as

$$V=Y_3+Y_2+Y_1+Y_0 V=Y3+Y2+Y1+Y0$$

We can implement the above Boolean functions using logic gates. The **circuit diagram** of 4 to 2 priority encoder is shown in the following figure.



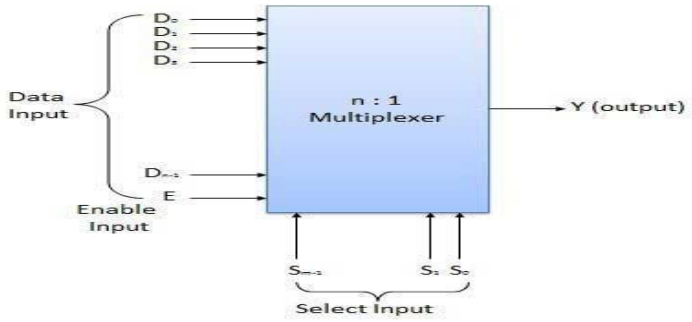
The above circuit diagram contains two 2-input OR gates, one 4-input OR gate, one 2input AND gate & an inverter. Here AND gate & inverter combination are used for

producing a valid code at the outputs, even when multiple inputs are equal to ‘1’ at the same time. Hence, this circuit encodes the four inputs with two bits based on the **priority** assigned to each input.

Multiplexers

Multiplexer is a special type of combinational circuit. There are n-data inputs, one output and m select inputs with 2m = n. It is a digital circuit which selects one of the n data inputs and routes it to the output. The selection of one of the n inputs is done by the selected inputs. Depending on the digital code applied at the selected inputs, one out of n data sources is selected and transmitted to the single output Y. E is called the strobe or enable input which is useful for the cascading. It is generally an active low terminal that means it will perform the required operation when it is low.

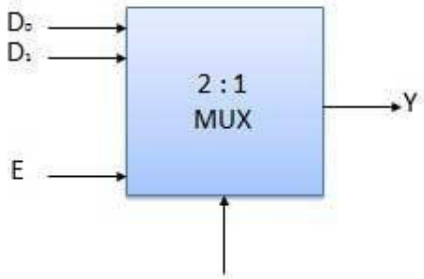
Block diagram



Multiplexers come in multiple variations

1. 2 : 1 multiplexer
2. 4 : 1 multiplexer
3. 16 : 1 multiplexer
4. 32 : 1 multiplexer

Block Diagram



Truth Table

Enable	Select	Output
E	S	Y
0	x	0
1	0	D_0
1	1	D_1

x = Don’t care

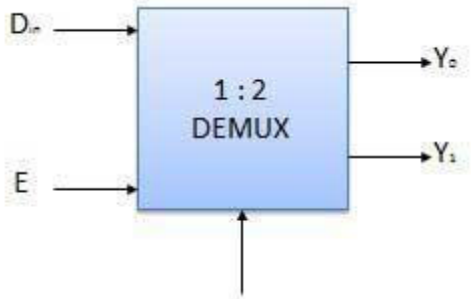
Demultiplexers

A demultiplexer performs the reverse operation of a multiplexer i.e. it receives one input and distributes it over several outputs. It has only one input, n outputs, m select input. At a time only one output line is selected by the select lines and the input is transmitted to the selected output line. A de-multiplexer is equivalent to a single pole multiple way switch as shown in fig.

Demultiplexers comes in multiple variations.

1. 1 : 2 demultiplexer
2. 1 : 4 demultiplexer
3. 1 : 16 demultiplexer
4. 1 : 32 demultiplexer

Block diagram



Truth Table

Enable	Select	Output
E	S	Y0 Y1
0	x	0 0
1	0	0 D_{in}
1	1	D_{in} 0

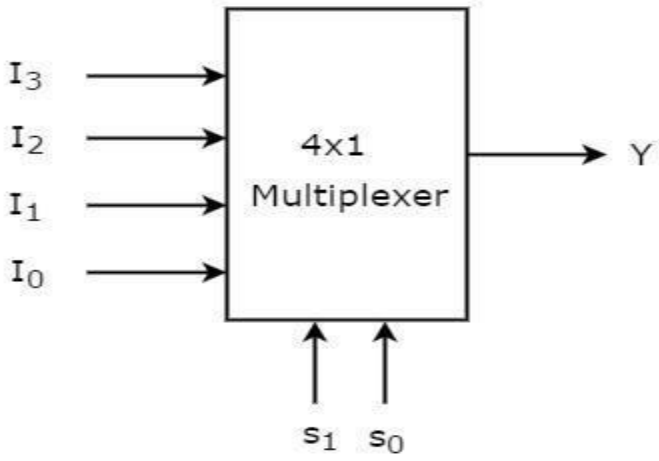
x = Don't care

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.



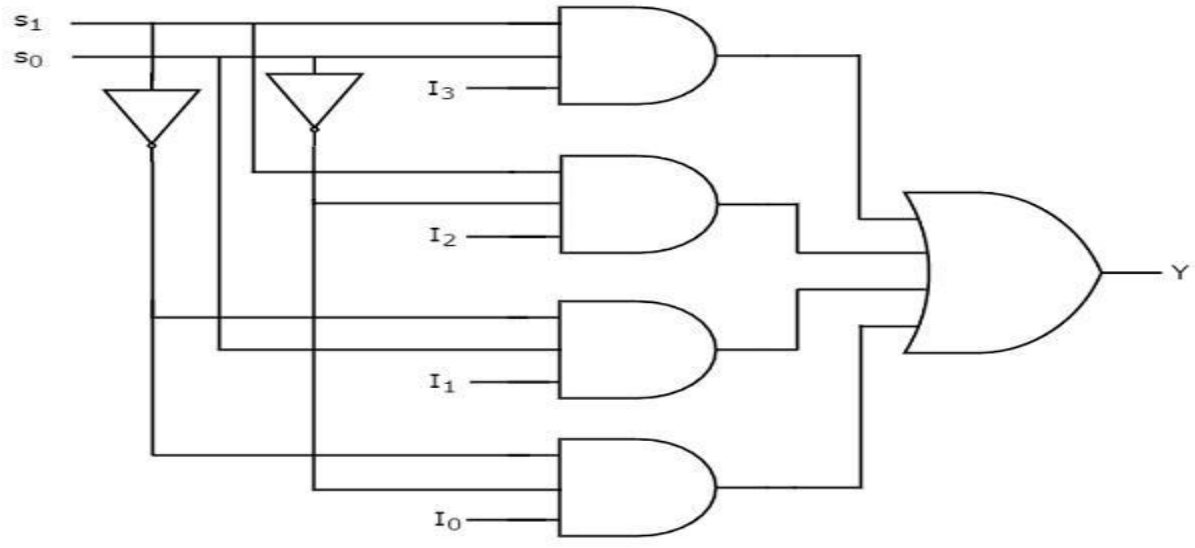
One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. **Truth table** of 4x1 Multiplexer is shown below.

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

$$Y=S_1'S_0'I_0+S_1'S_0I_1+S_1S_0'I_2+S_1S_0I_3 \quad Y=S1'S0'I0+S1'S0I1+S1S0'I2+S1S0I3$$

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

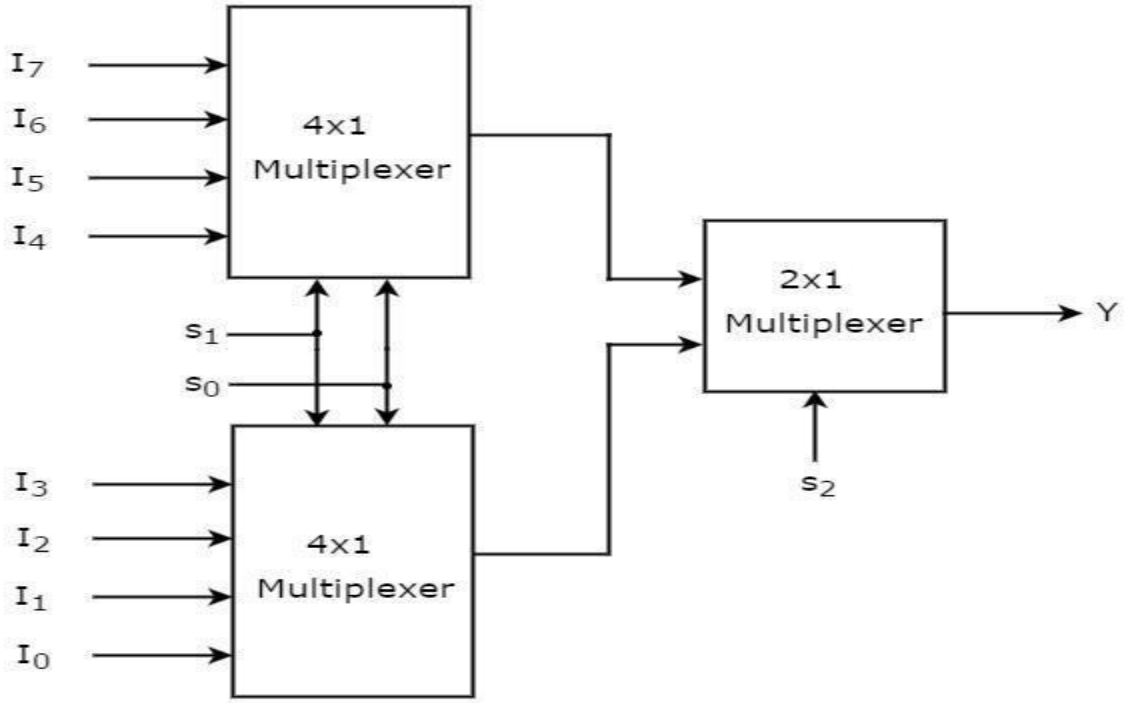
In this section, let us implement 8x1 Multiplexer using 4x1 Multiplexers and 2x1 Multiplexer. We know that 4x1 Multiplexer has 4 data inputs, 2 selection lines and one output. Whereas, 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output.

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.

1. 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.
2. 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.

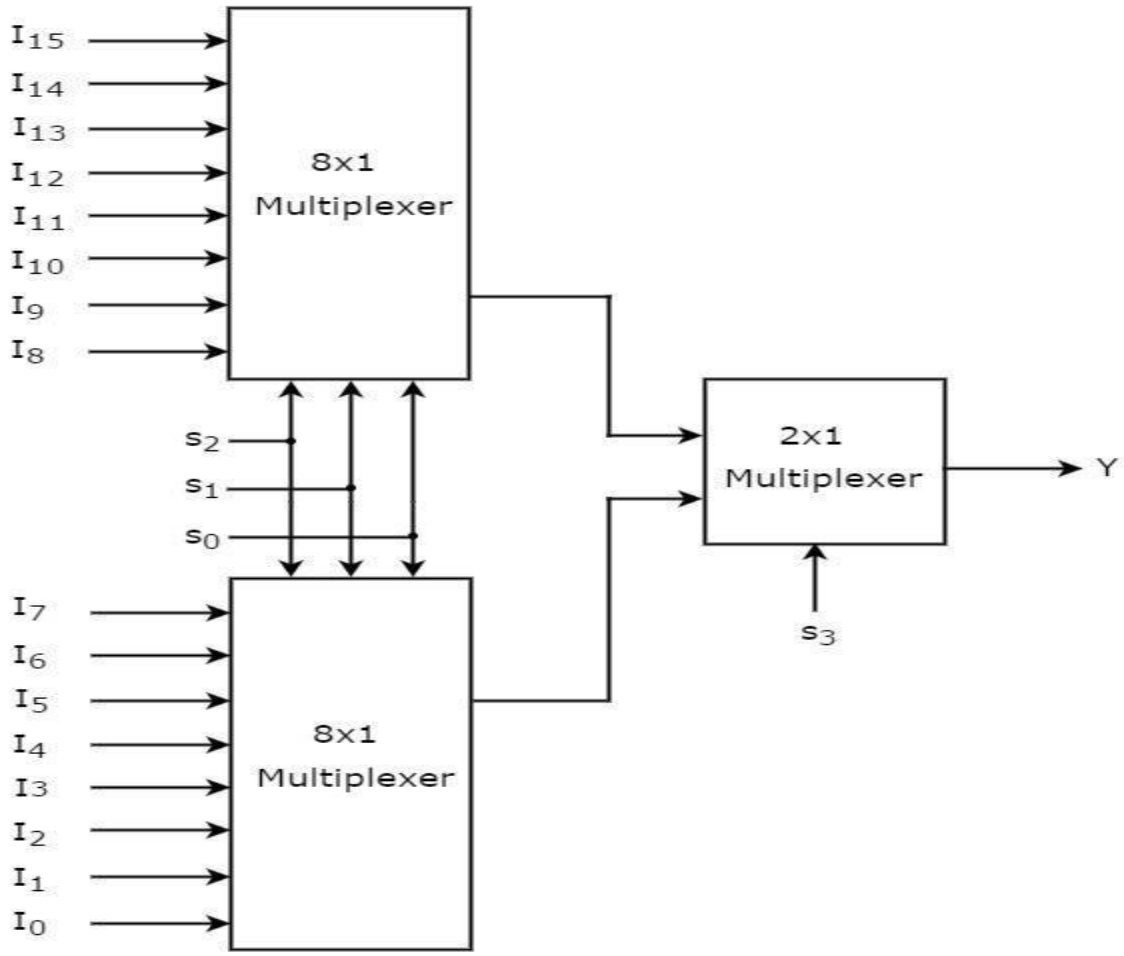
So, we require two **8x1 Multiplexers** in first stage in order to get the 16 data inputs. Since, each 8x1 Multiplexer produces one output, we require a 2x1 Multiplexer in second stage by considering the outputs of first stage as inputs and to produce the final output.

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.

1. 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.
2. 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.

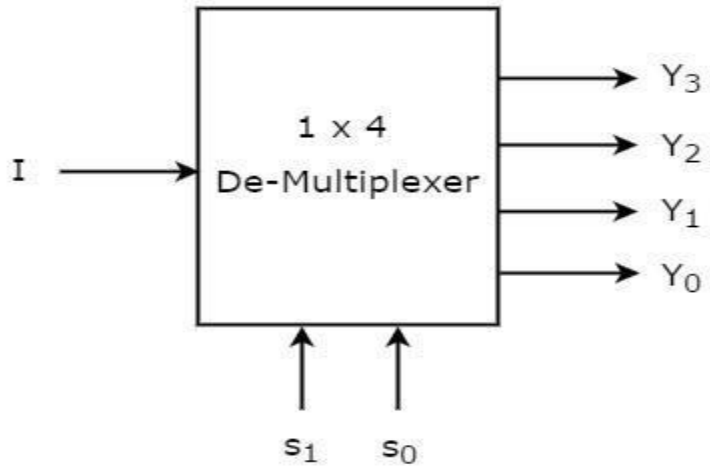
De-Multiplexer is a combinational circuit that performs the reverse operation of

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		**O ıtputs**			
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

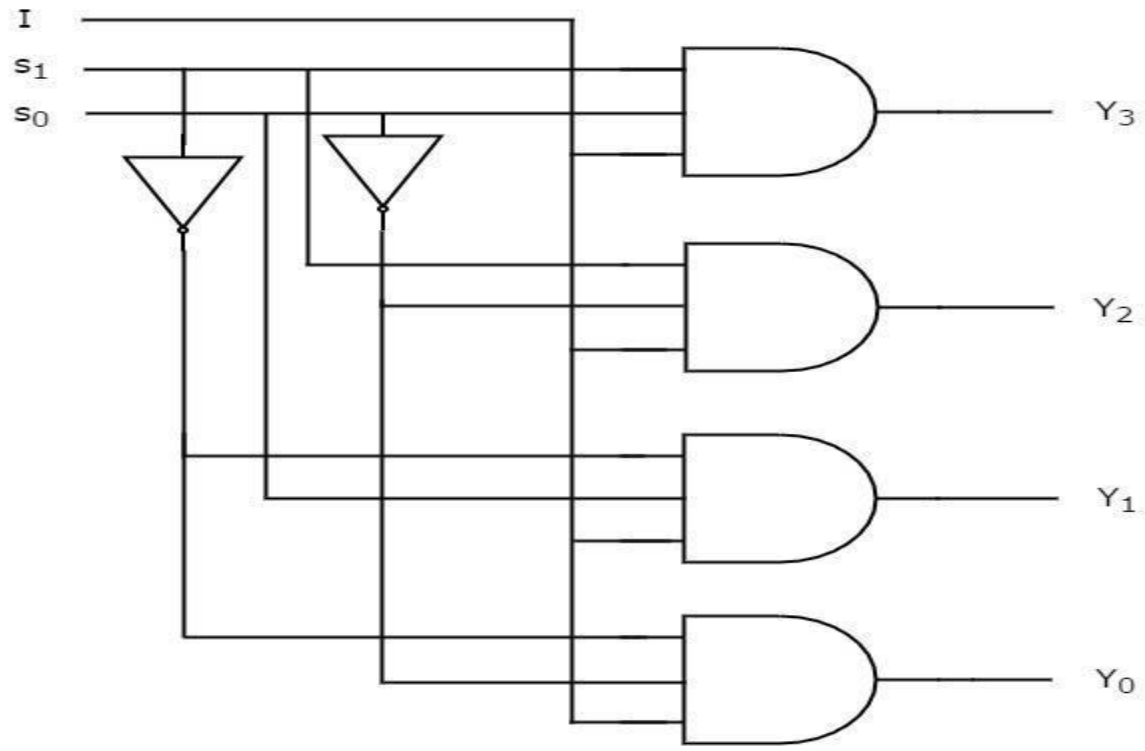
$$Y_3=s_1s_0I Y3=s1s0I$$

$$Y_2=s_1s_0'I Y2=s1s0′I$$

$$Y_1=s_1's_0IY1=s1's0I$$

$$Y_0=s_1's_0'IY0=s1's0'I$$

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.

1. 1x8 De-Multiplexer
2. 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.

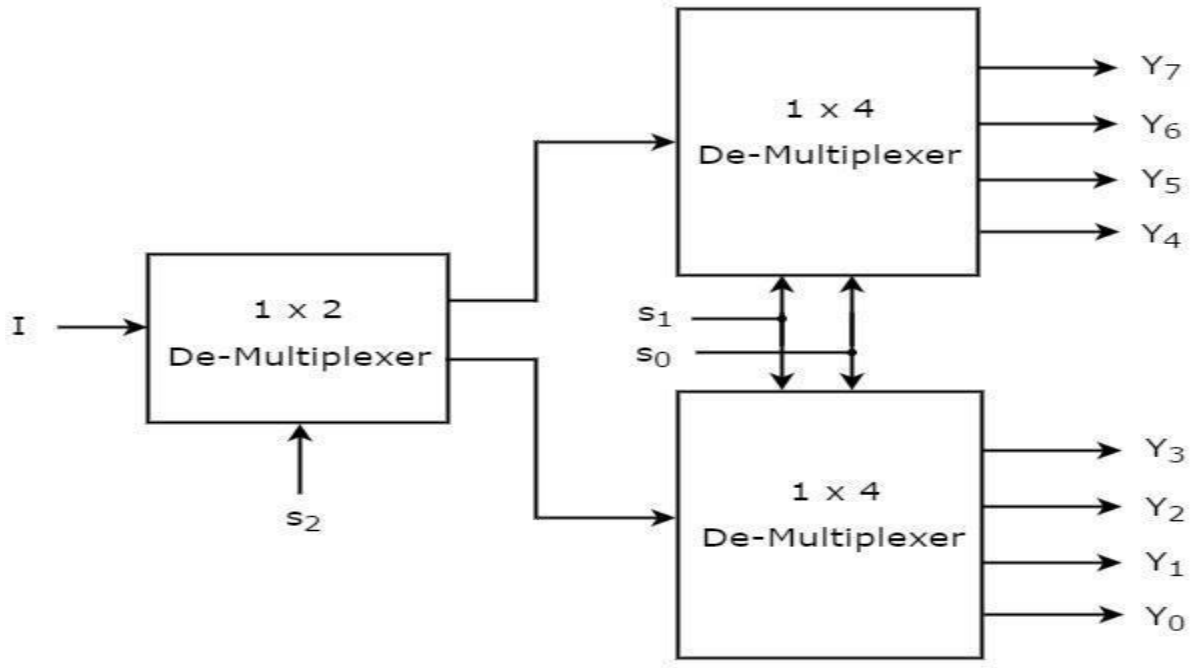
So, we require two **1x4 De-Multiplexers** in second stage in order to get the final eight outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer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

M.JOSHNA

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 DeMultiplexer are Y_3 to Y_0.

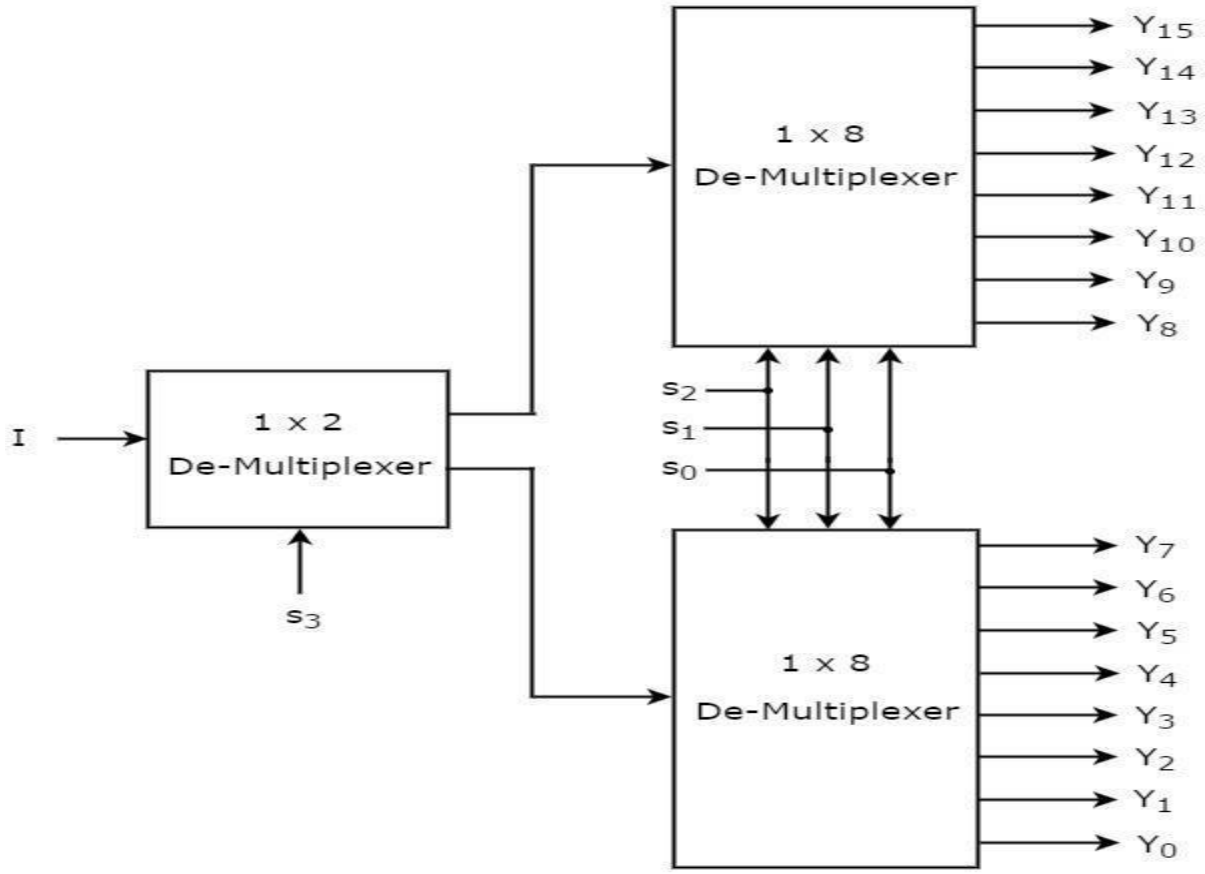
The other **selection line, s_2** is applied to 1x2 De-Multiplexer. If s_2 is zero, then one of the four outputs of lower 1x4 De-Multiplexer will be equal to input, I based on the values of selection lines s_1 & s_0. Similarly, if s_2 is one, then one of the four outputs of upper 1x4 DeMultiplexer will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

M.JOSHNA

UNIT-3: Computer Arithmetic and Process Organization

Computer Arithmetic: Algorithms for fixed point and floating point addition, subtraction, multiplication and division operations.

Processor Organization: Introduction to CPU, Execution of a Complete Instruction, Multiple-Bus Organization, Hardwired Control and Multi programmed Control.

Computer Arithmetic:

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations. To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation. If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems we give algorithms. In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations. And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods. A processor has an arithmetic processor(as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1’s complement or signed 2’s complement. Most computers use the signed magnitude representation for the mantissa.

Algorithms for fixed point and floating point addition, subtraction, multiplication and division operations:

In a computer, the basic arithmetic operations are Addition and Subtraction. Multiplication and Division can always be managed with successive addition or subtraction respectively. However, hardware algorithms are implemented for Multiplication and Division.

It is to be recollected that computers deal with binary numbers unless special hardware is implemented for dealing with other number systems. Although instructions may be available for treating signed and unsigned operations, the programmer must deal with the numbers and handling of the result. The hardware assists the programmer by way of appropriate instructions and flags.

Addition

Adding two numbers is an addition. We may add signed or unsigned numbers. When we add two numbers, say 8 and 5, the result is 13 i.e. while adding two single-digit numbers, we may get a two-digit number in the result. A similar possibility exists in the binary system too. Thumb rule of binary addition is:

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 10

Examples (a –e) of unsigned binary addition are given in figure 8.1.

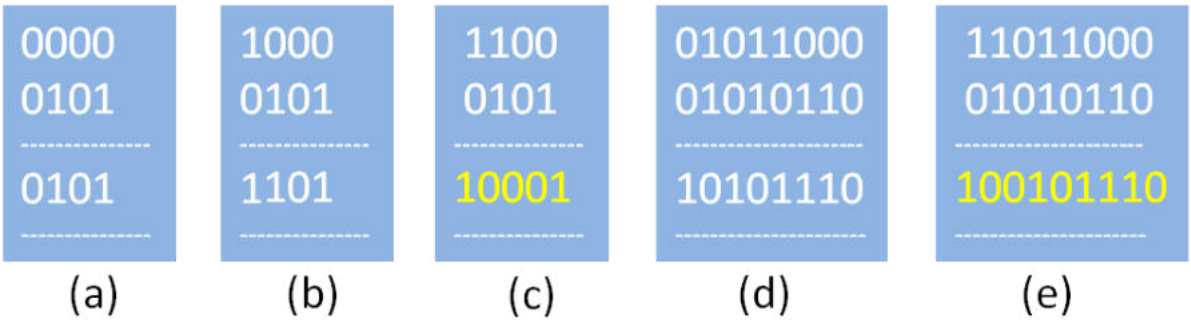


Figure 8.1 Examples of binary Addition

Adder

The hardware circuit which executes this addition is called Adder. There are two types of adders namely Half adder and Full adder. Basic adder circuit does 1-bit addition and is extended for n-bit addition. The adder circuit characteristics are detailed by a circuit, a truth

table, Formula and a block symbol. The adder circuits are constructed from logic gates which satisfy the formula as per truth table. These are also called combinational logic. A Combinational logic output reflects the input without clocking.

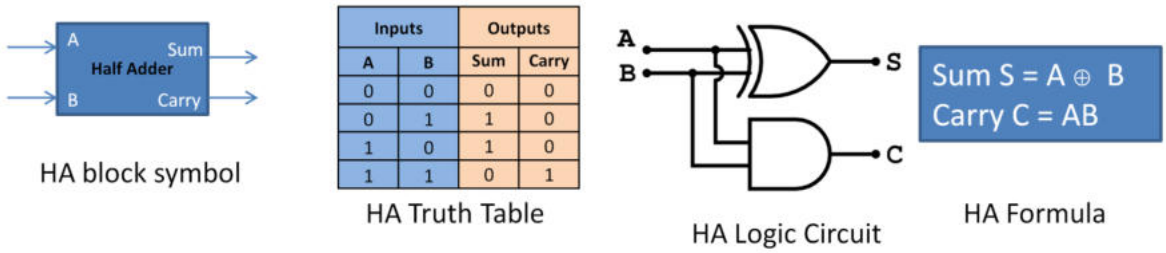


Figure 8.2 Half adder

The Half Adder (HA) has two inputs (A, B) and two outputs (Sum and Carry). The Sum is XOR of input while the Carry is AND of the input. The Half Adder is detailed in figure 8.2.

A Full Adder (FA) also performs 1-bit addition but taking 3 inputs (A, B and C_i) and produces two outputs (Sum and Carry). Like HA, FA generates result consisting of Sum (S) and Carry out (C_{out}). C_{out} is used as C_{i+1} while cascading for multiple bits of a word. Full Adder is detailed in figure 8.3. A full adder can also be constructed using half adder blocks as in figure 8.4.

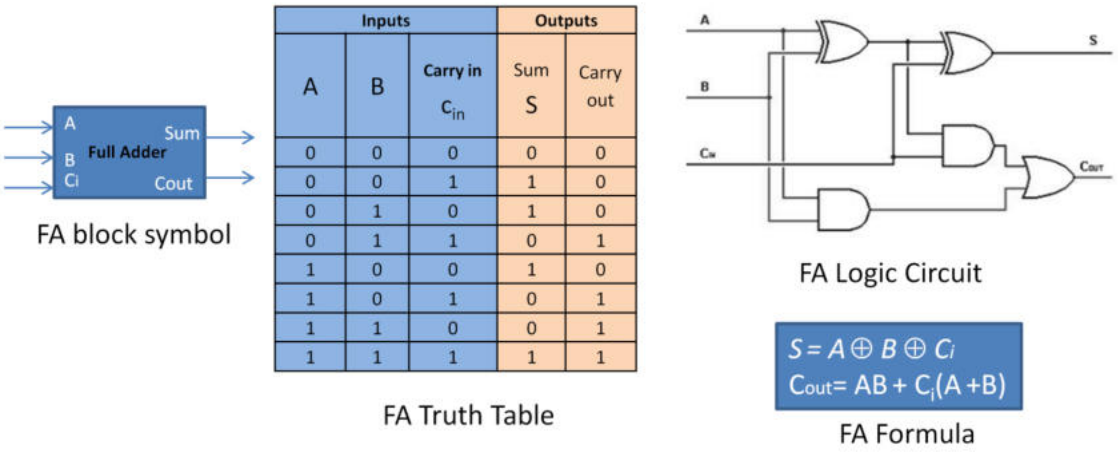


Figure 8.3 Full adder

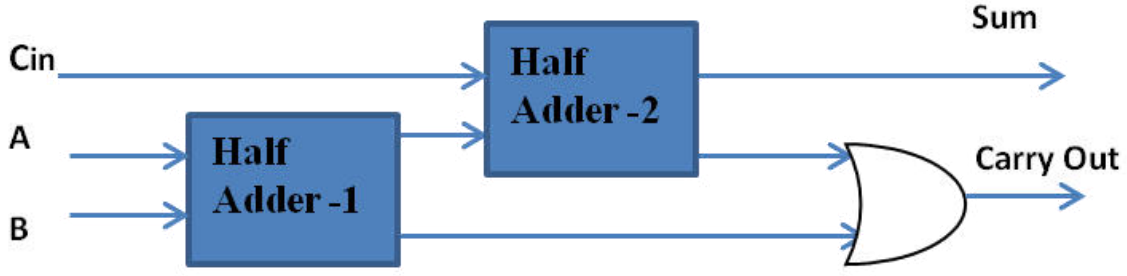


Figure 8.4 Full Adder using Half Adder blocks

Subtraction

Subtraction is finding the difference of B from A i.e A-B. Basis of binary subtraction is:

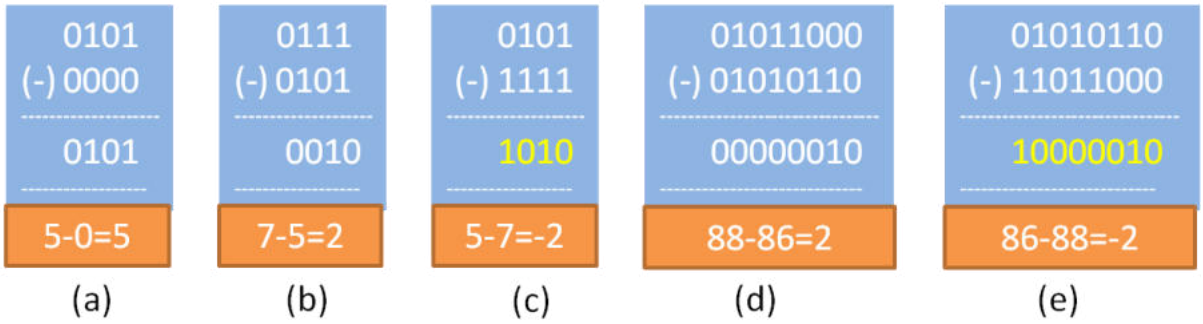
0 - 0 = 0

0 - 1 = -1

1 - 0 = 1

1 - 1 = 0

Of course, the usual borrow logic from the adjacent digit is applied as in the case of decimal numbers. Examples of signed binary Subtraction is as below:



Examples of signed binary subtraction

You may note that the above examples are in sign-magnitude representation. In sign-magnitude form, MSB is reserved for sign representation. This is only for basic understanding. Computers internally use 2's complement representation.

Recall: In 2's complement representation MSB is sign bit, (n-1) bits of the represent magnitude of the number. Conversion sample for 8-bit word is shown in figure 8.5.

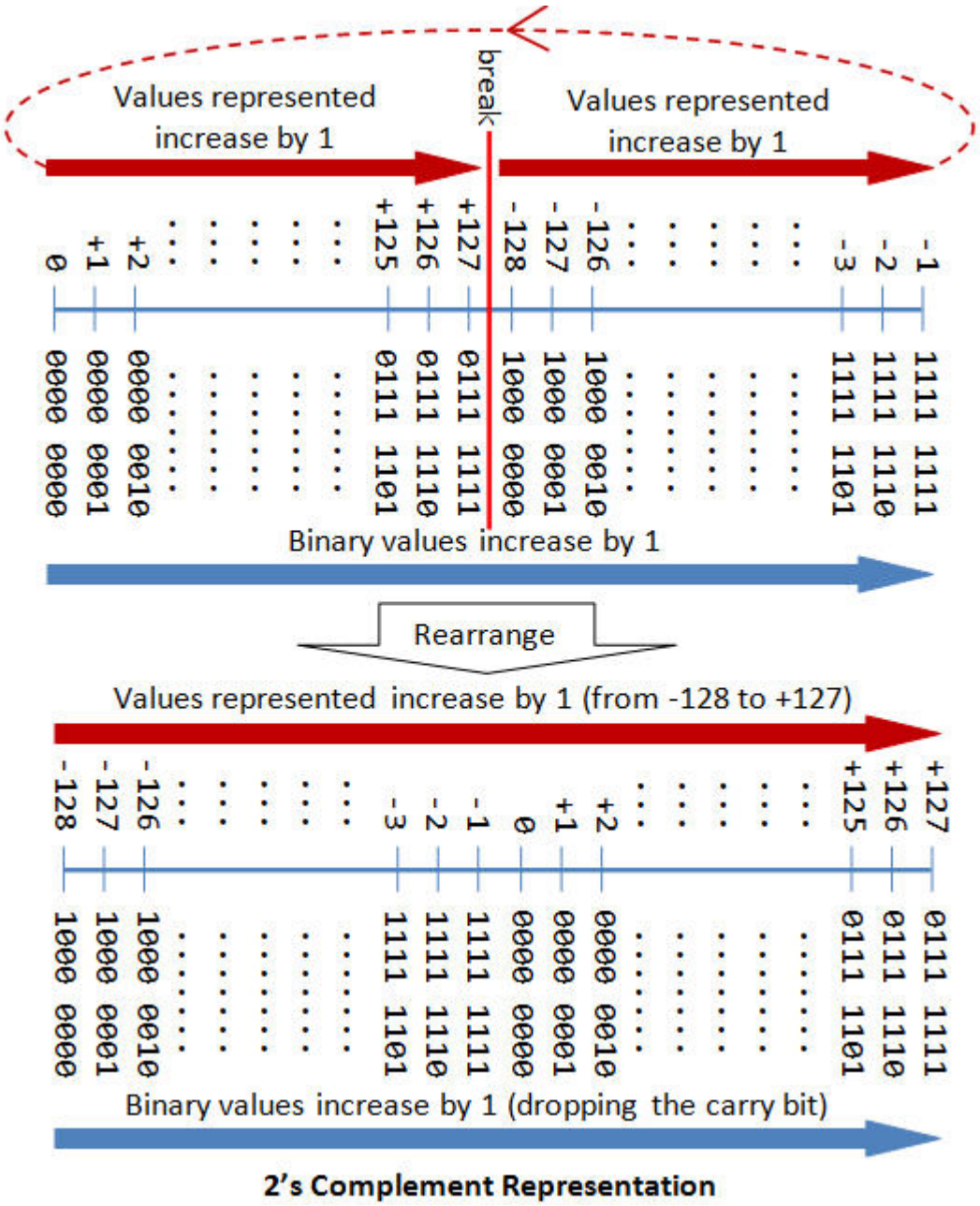
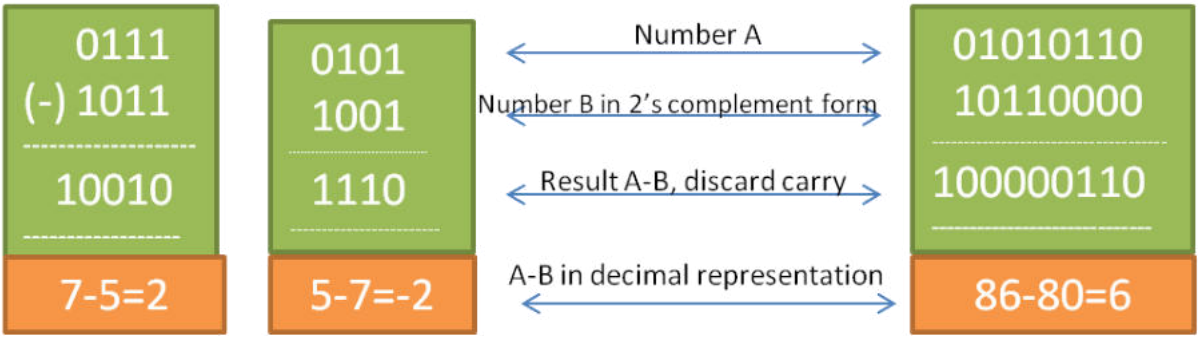


Figure 8.5 1's and 2's Complement Representation

2's Complement for Subtraction

"1's complement + 1 = 2's complement"

Generating this 2's complement is very simple using an XOR circuit. The XOR circuit will generate 1's complement. A control signal called SUBTRACT is used as add value of 1. This way, an adder executes subtraction. See the example below, where case (b), case (c) and case (e) are worked out as 2's complement representation; and A-B becomes A + (2's complement(B)). The result is obtained in 2's complement form discarding the carry. Observe that this method works for all kind of data.



Interpreting 2's complement numbers

- Observe the sign bit (MSB)
- If '0', the number is positive; the (n-1) bits mean the absolute value of the number in binary
- If '1', the number is negative; (n-1) bits mean the 2's complement value of the number in binary; Invert the (n-1) bits and add 1 to get the absolute value of this negative number.

Error Detection and Status Flags

No one does maths perfectly, but computers can do, provided your data is right! There is a probability that your data may not be rightly defined or may be out of range. For this reason, the CPU detects certain errors like OVERFLOW(O), UNDERFLOW(U) and CARRY(C). It also detects SIGN(S) and ZERO(Z) status. The acronym is ZSOC (Zero, sign, Overflow and Carry) as many processors may treat overflow and underflow together as out of range. The detection is done by the Arithmetic and Logic Unit (ALU) of the CPU. Upon detection corresponding flag is set to ON status. These flags have bit positions allotted in the Processor Status Register and most famously known as Processor Status Word (PSW). ZSOC flags are collectively known as Condition Codes. The purpose of these Condition Codes status flags is to facilitate the programmer to catch data dependant errors and act accordingly.

Overflow: To put in simple English, when a result obtained exceeds the maximum number possible to be represented, Overflow is said to occur. In other words, the addition of two numbers with sign bit '0' resulting in value with sign bit '1' is said to be an OVERFLOW.

For example : An 8 bit word can maximum represent +127 decimal, 01111111 in binary. If we add, 120 + 10 -> 130;

120 -> 0111 1000

10 -> 0000 1010

1000 0010 -> in sign magnitude form, MSB (Mos significant bit) '1'

means negative number while we expect +130

Max is +127, hence this is an overflow scenario

In Overflow scenario, the result is wrong and this needs to be communicated to the programmer/user that there is an error encountered. This situation is detected by the CPU hardware and sets a status bit called "OVERFLOW". The user, if interested, can catch this error by reading this OVERFLOW status bit and take necessary action over the data handling.

Underflow: While Overflow is related to positive magnitude, Underflow is related to negative magnitude for the same reasons. As an example, when you add two negative numbers like -120 and -10, the result expected is -130 which is beyond the representable range in an 8-bit signed word definition. This is a scenario of UNDERFLOW. In other words, the addition of two numbers with sign bits '1' resulting in a number with sign bit '0' is said to be UNDERFLOW. The CPU hardware detects and sets a status bit called UNDERFLOW to this effect. Again, this status bit is accessible to the programmer/user to take necessary action over the data handling.

Carry: CARRY is another status detected and set by CPU while executing arithmetic instructions. CARRY flag is relevant to Unsigned arithmetic operations while OVERFLOW and UNDERFLOW are relevant to signed operations.

The CARRY Flag is set by the CPU at the end of arithmetic operations if there is a Carry (C_{out}) out of the most significant bit of the word. The Carry is set at the end of an execution cycle of the addition or subtraction instructions. Many CPUs do not differentiate between signed and unsigned operations, in which case the CARRY and OVERFLOW may both be set by the CPU. However, there are CPUs which have different instruction codes and instructions for signed and unsigned integer operations and in this case, the CPU appropriately sets the CARRY or OVERFLOW flag.

Never forget that it is the programmer who decides whether he is operating with signed or unsigned numbers. So the programmer has to decide whether he should catch OVERFLOW, UNDERFLOW or CARRY flag for error detection and corrective action.

ZERO: At the end of an instruction execution cycle, if the accumulator value is zero, this status bit is set by CPU. This could be a possibility at the end of arithmetic or logical instructions or load instructions.

SIGN: Sign bit reflects the MSB of the accumulator. This is also set at the end of Instruction execution cycle.

n-Bit Adder Formation

A 4-bitFull Adder is integrated by cascading four numbers of 1-bit adders as in figure 8.6. When cascaded the C_{out} of ith goes as C_{in} of i+1th position and hence the carry is said to be propagated. S is the Sum bits, C_{out} is the final Carry out of the adder. A and B are the input

numbers. C_i is carry-in if available. Such cascading can be extended to any number of bits using 1-bit FA or n-bit FA blocks.

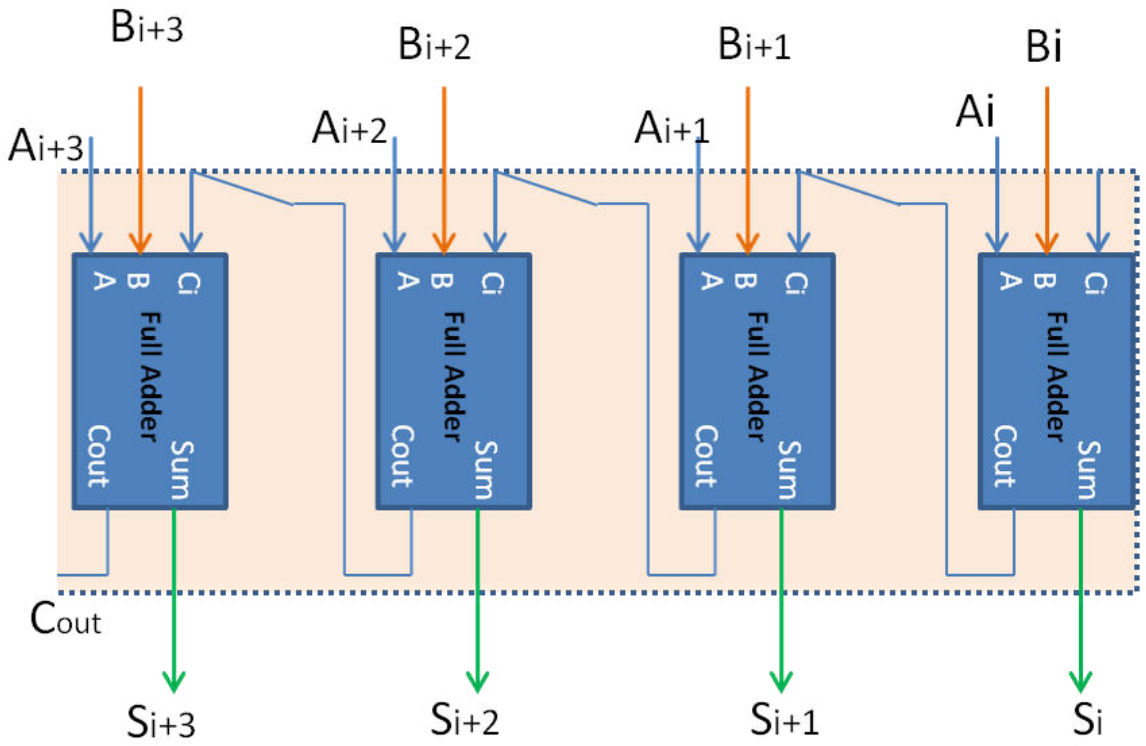


Figure 8.6 4-bit Ripple Carry Full Adder

This method of Adder expansion is known as Spatial Expansion as the output of all the n-bits are available at the same time as 1-bit operation, probably in one clock cycle. Spatial expansion is also known as Parallel Adder. The other name for this method is Ripple Carry adder as the carry is propagated internally. However, for large n-value, the carry propagation delay for clean and settled output proportionately increases. This is a disadvantage of Ripple Carry Adder which is solved by Carry-Look-Ahead Adder Technique.

Carry Look Ahead Adder

This is also a spatial expansion and ripple carry type. The Carry Look Ahead Adder (CLA) uses specialized logic called Carry Look ahead Logic to compute carries in parallel and hence is faster than Ripple Carry Adder.

Figure 8.7 4-bit Carry Look Ahead Full Adder

CLA Adder generates two other signals namely Propagate Carry and Generate Carry which can be used by the next stage for Carry Calculation.

Propagate Carry Pi = Ai + Bi

i.e when either of the number has '1' in the bit position, the carry is likely depending on the Ci

Generate Carry Gi = AiBi

i.e. when both the numbers have '1' in their bit position in which case carry is sure to be generated.

We already have defined the formula for Sum and Carry as

$S = A \oplus B \oplus C_i$

$C_{out} = AB + C_i(A+B)$

The carry formula can be rewritten in terms of Propagate and Generate carry as $C_{out} = P_i + C_iG_i$.

In a Carry look-ahead adder, the carries are computed in parallel using carry look-ahead logic, in one gate delay as compared to 2-gate delays per bit in the case of Ripple carry adder.

Implementation of Multiplication and Division.

Multiplication

Just recall with micro details as to how do we do multiplication using pen and paper. Then it is easier to visualize that it is possible to implement a hardware algorithm.

Multiplicand M = 12 1100

Multiplier Q = 11 x1011

1100

1100

0000

1100

Product P = 132 10000100

As you see, we start with LSB of the Multiplier Q, multiply the Multiplicand, the partial product is jotted down. Then we used the next higher digit of Q to multiply the multiplicand. This time while jotting the partial product, we shift the jotting to the left corresponding to the Q–digit position. This is repeated until all the digits of Q are used up and then we sum the partial products. By multiplying 12x11, we got 132. You may realize that we used binary values and the product also in binary. Binary multiplication was much simpler than decimal multiplication. Essentially this is done by a sequence of shifting and addition of multiplicand when the multiplier consists only of 1's and 0's. This is true and the same, in the case of Binary multiplication. Binary multiplication is simple because the multiplier would be either a 0 or 1 and hence the step would be equivalent to adding the multiplicand in proper shifted position or adding 0's.

It is to be observed that when we multiplied two 4-bit binary numbers, the product obtained is 8-bits. Hence the product register (P) is double the size of the M and Q register. The sign of the product is determined from the signs of multiplicand and multiplier. If they are alike, the sign of the product is positive. If they are unlike, the sign of the product is negative.

Unsigned Multiplication

When multiplication is implemented in a digital computer, it is convenient to change the process slightly. It is required to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register. The registers, Shift Counter and the ALU width is decided by the word size of the CPU. For simplicity of understanding, we will take 4-bit word length i.e the Multiplier (Q) and Multiplicand (M) are both 4-bits sized. The logic is extrapolated to the word size requirement.

We need registers to store the Multiplicand (M) and Multiplier (Q) and each 4-bits. However, we use 8-bit register which is standard and minimum and hence the register to collect Product (P) is 16-bits. Refer to figure 9.1. The Shift counter keeps track of the number of times the addition is to be done, which is equal to the number of bits in Q. The shifting of the contents of the registers is taken care of by shift register logic. The ALU takes care of addition and hence partial product and product are obtained here and stored in P register. The control unit controls the cycles for micro-steps. The product register holds the partial results. The final result is also available in P when the shift counter reaches the threshold value.

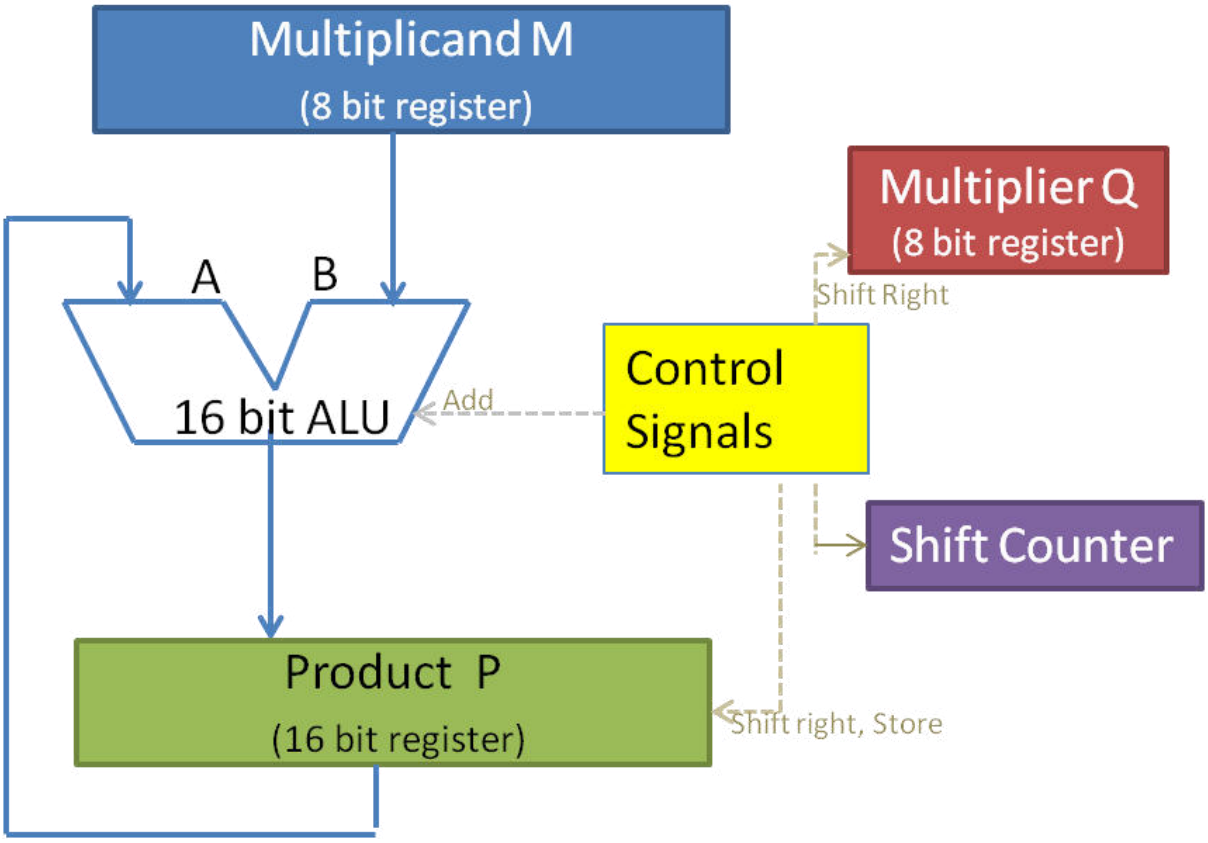


Figure 9.1 Data path for typical Multiplication

The flowchart for the unsigned multiplication is shown in figure 9.2 and table 9.1 explains the work out with an example of 12 x 11 values. The flowchart is self-explanatory of the unsigned multiplication algorithm. In an unsigned multiplication, the carry bit is used as an extension of the P register. Since the Q value is a 4-bit number, the algorithm stops when the shift counter reaches the value of 4. At this point, P holds the result of the multiplication.

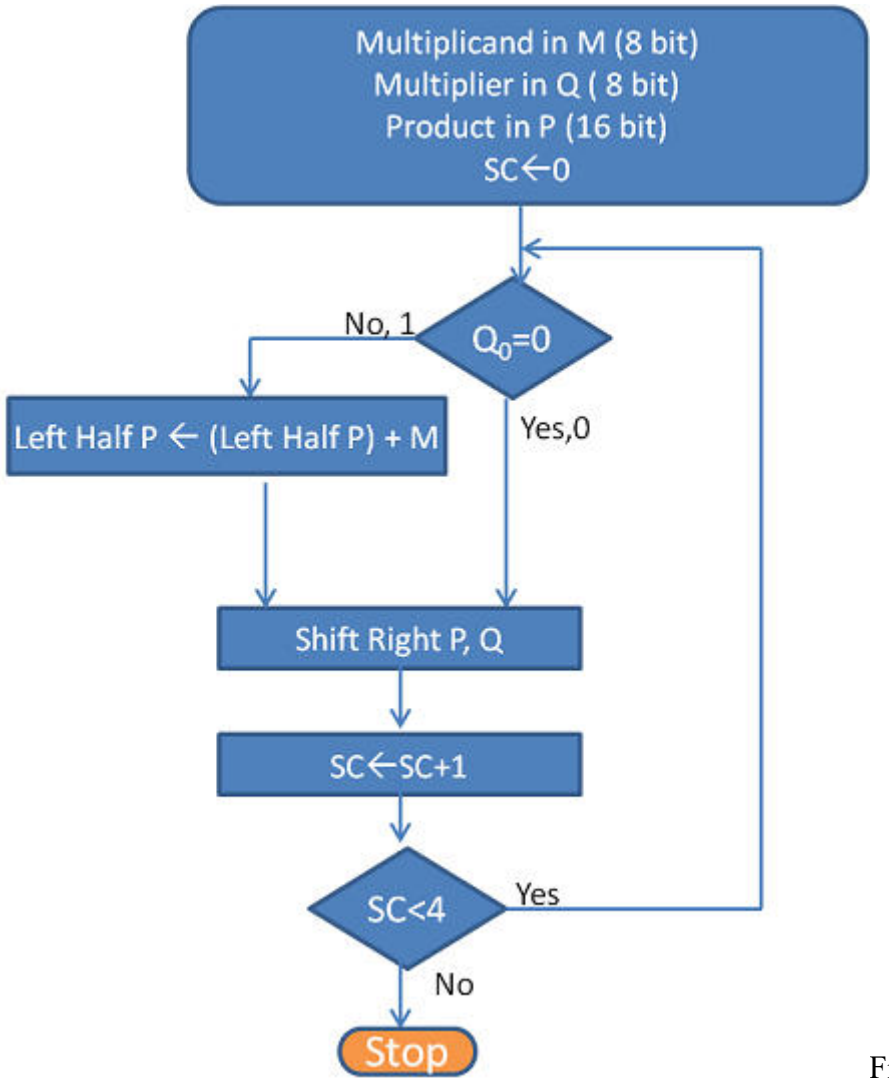


Figure 9.2 Flowchart for Unsigned Multiplication algorithm

Table 9.1 Workout for unsigned multiplication (12 x 11 = 132)

Operation Step	Shift Value	Counter	Multiplicand M	Multiplier Q	Product P
Initial Values for multiplication of 12x11	0		1100	1011	0000 0000
Q_0 = 1, So, Left half of P <- Left half of P + M	0		1100	1011	1100 0000
Shift Right P, Shift Right Q	0		1100	0101	0110 0000
SC <- SC + 1	1		1100	0101	0110 0000
Q_0 = 1, So, Left half of P <- Left half of P + M	1		1100	0101	10010 0000

Table 9.1 Workout for unsigned multiplication (12 x 11 = 132)

Operation Step	Shift Value	Counter	Multiplicand M	Multiplier Q	Product P
Shift Right P, Shift Right Q	1		1100	0010	1001 0000
SC <- SC + 1	2		1100	0010	1001 0000
Q_0 = 0, do nothing	2		1100	0010	1001 0000
Shift Right P, Shift Right Q	2		1100	0001	0100 1000
SC <- SC + 1	3		1100	0001	0100 1000
Q_0 = 1, So, Left half of P <- Left half of P + M	3		1100	0001	10000 1000
Shift Right P, Shift Right Q	3		1100	0000	1000 0100
SC <- SC + 1	4		1100	0000	1000 0100

Signed Multiplication

Signed numbers are always better handled in 2's complement format. Further, the earlier signed algorithm takes n steps for n digit number. The multiplication process although implemented in hardware 1-step per digit is costly in terms of execution time. Booths algorithm addresses both signed multiplication and efficiency of operation.

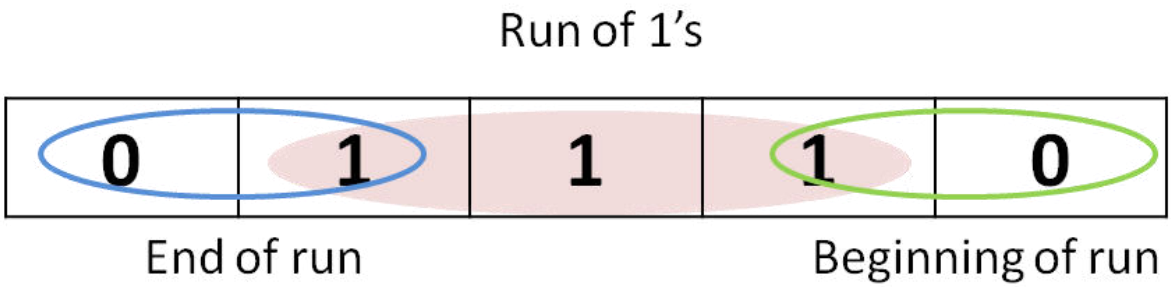
Booth's Algorithm

Booth observed that multiplication can also be done with mixed additions and subtractions, instead of only additions. And it deals with signed multiplication as well.

The motivation for Booth's Algorithm is that ALU with add or subtract can get the same result in more than one way .i.e. the multiplier 6 can be dealt as:

6 = – 2 + 8

Booth's Algorithm categorises the multiplier as the run of 1's and further as begin, middle and end of runs. The run is identified as below for a number 01110.



Run of 1's

Based on the run status, the operation to be performed in the multiplication process is defined as in table 9.2. The values of the current bit (Q0) and the outgoing bit (Qe) of the multiplier decide the operation to be performed. By this, the multiplication is achieved in less number of cycles based on the multiplier. A multiplier may have many combinations of runs based on its value. This algorithm is sensitive to bit patterns of Multiplier. A pattern like 01010101 may be the worse as it has many begin and end runs necessitating as many additions and subtractions and may not save cycle time. But by and large Booth's algorithm saves cycles.

Table 9.2 Booth Encoding for Multiplication – Operation regarding the run

Current Bit (Q_0)	**Bit to the right (Q_e)**	**Explaination**	**Example**	**Operation**
1	0	Begins run of 1s	00011**11**000	Subtract multiplicand from partial product
1	1	Middle of run of 1s	0001**11**1000	No arithmetic operation
0	1	End of run of 1s	0**001**111000	Add multiplicand to partial product
0	0	Middle of run of 0s	**00**01111000	No arithmetic operation

Booth's algorithm uses Arithmetic Shift Right for collecting partial product. Arithmetic Shift right is a sign-extended shift; i.e if the sign bit is 0, then 0 is extended while shifting; if the sign bit is 1, then 1 is extended while shifting. For this reason, n+1 is the register size. You may observe this in our work out in table 9.3. The work out is for (-12x -11). This example is taken to demonstrate the outcome of signed multiplication with Booth's algorithm. Both multiplicand (M) and Multiplier (Q) use 5-bits as against 4-digit binary number.

The partial product and Product is collected in P and Q register. The Q register initially holds the Multiplier; as it gets shifted out with every digit multiplication, the space in Q register is occupied by partial product. Qe is a 1-bit register holding the outgoing bit. Together PQQe is treated as one entity during the arithmetic shift, whereas only P is considered for addition or subtraction of multiplicand. The multiplicand is loaded in M. Both Multiplicand and Multiplier

are loaded in the simple binary form if these are positive numbers and in 2's complement form if these are negative numbers. The shift counter stops the operation once it reaches the digit count of Q, in this case, 4. Q0Qe are evaluated at every step to decide the operation to be carried out on M and P.

Booth's Algorithm Work out Example (-12 x -11 =132)

Comments	SC	Multiplicand M	Product P	Product Q	Q_e
Initial (-12 x -11)	0	10100	00000	10101	0
Q_0Q_e = 10; P←P-M	0	10100	01100	10101	0
Ar.sh.r . PQ Q_e	0	10100	00110	01010	1
SC ← SC+1	1	10100	00110	01010	1
Q_0Q_e = 01; P←P+M	1	10100	11010	01010	1
Ar.sh.r . PQ Q_e	1	10100	11101	00101	0
SC ← SC+1	2	10100	11101	00101	0
Q_0Q_e = 10; P←P-M	2	10100	01001	00101	0
Ar.sh.r . PQ Q_e	2	10100	00100	10010	1
SC ← SC+1	3	10100	00100	10010	1
Q_0Q_e = 01;	3	10100	11000	10010	1
Ar.sh.r . PQ Q_e	3	10100	11100	01001	0
SC ← SC+1	4	10100	11100	01001	0
Q_0Q_e = 10; P←P-M	4	10100	01000	01001	0
Ar.sh.r . PQ Q_e	4	10100	00100	00100	1
SC ← SC+1	5	10100	**00100**	**00100**	1

-12 x -11 = 132

Table 9.3 Booth's Signed Multiplication

It is seen that the resulting Product of multiplying two negative numbers is a positive number which is correct. One need not handle the signs separately. It is handled as part of the algorithm. The flow chart and the datapath may be drawn by an interested reader as an exercise or the reader may contact the author.

There is a category of Multipliers called Array multiplier which avoids this sequential operation and produces the result at once. These require a large number of gates for implementation. However, with the advancement in VLSI, it is a reality. Different CPUs have different implementations.

DIVISION:

Performing division is a difficult task as we have seen in case of fixed point arithmetic also. Divider architectures are complex to implement. Floating point division is nothing but a fixed point division with some extra hardwares to take care for the exponents. This extra hardwares make the divider circuit more complex. A floating point division where a number divides another number can be expressed as

Thus it can be said that in a floating point division, mantissas are divided and exponents are subtracted.

The major steps for a floating point division are

- Extract the sign of the result from the two sign bits.
- Find the magnitude of the difference between two exponents ( ). Add to the bias if or subtract from the bias if .
- Divide mantissa of ( ) by mantissa of ( ) considering the hidden bits.
- If there is a leading zero then normalize the result by shifting it left.
- Due to the normalization, the exponent is to be decremented according to the number of left shifts.

Floating point division can be more clearer with an example. Lets discuss a division operation between two numbers and . The result of the division operation is .

Example : Floating Point Division

- Representation: The input operands are represented as and
- Sign extraction: As one of the number is negative then sign of the output will be negative. Thus .

- Exponent subtraction: and . Thus magnitude of their difference is . As thus the resulted exponent is .
- Mantissa division: Divide the mantissas by any division algorithm used in the fixed point arithmetic. Considering the hidden bits, the division operation is restricted to 12-bits. The result of the division is .
- There is a leading zero in the result thus a left shift can be applied to normalize the result. Thus the new result is . The final value of the mantissa ( ) is excluding the hidden bit.
- The action of normalization step must reflect on exponent correction. The value of the exponent is corrected by a decrement corresponding to a left shift. The new value of the exponent ( ) is .
- The final result is . The decimal value of this is .

A simple architecture for floating point division is shown below in Figure 1 . There are three 4-bit subtractors used in the divider architecture, two for exponent subtraction and one for correction of exponents. The major hardware block is the divider block. The divider used here is a 12-bit unsigned divider and that can be any divider circuit as discussed in the blog for division. If the result of the divider contains any leading zero then normalizing step is executed. But here in this case, as the hidden bit is also considered thus the result can not go below . Thus there will be maximum of one leading zero present in the result. This why only the MSB of the result ( ) is considered and left shift block shifts only by one bit. Pipeline registers are also must be inserted according to the pipe lining stages of the divider.

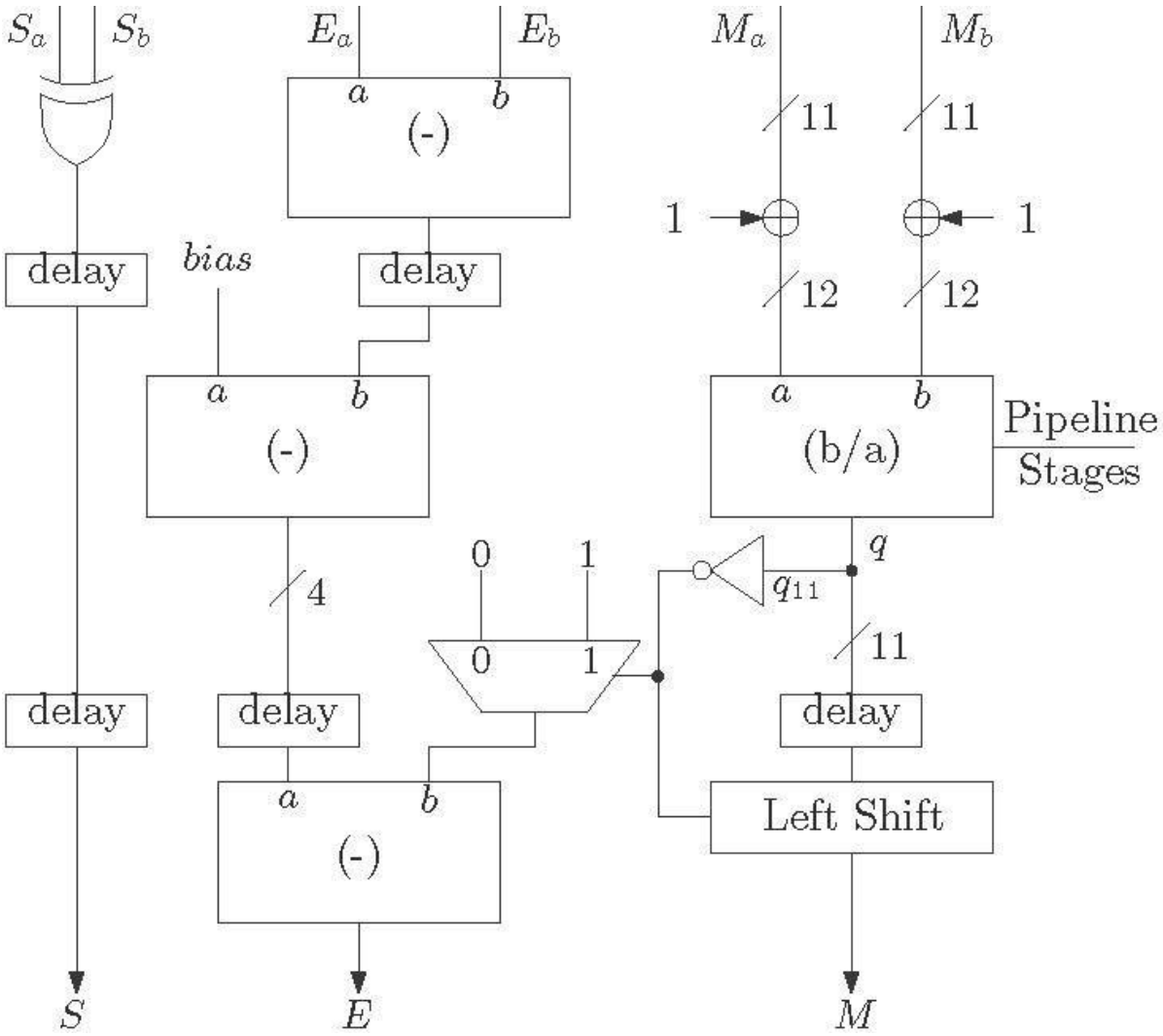


Figure 1: A Basic Scheme for Floating Point Division

Introduction to CPU

Introduction to CPU (Central Processing Unit)

The CPU (Central Processing Unit) is the brain of the computer responsible for executing instructions of a program. It interprets, controls, and processes the data stored in memory using arithmetic and logical operations.

In Digital Logic Design & Computer Organization (DLD&CO), the CPU is studied as the control and execution unit of the computer system.

---

1. Functional Role of CPU

- Fetches instructions from memory.
- Decodes the instructions to understand what operation to perform.
- Executes the instructions using ALU and registers.
- Stores results back into registers or memory.

This cycle is called the Instruction Cycle (Fetch → Decode → Execute → Store).

---

2. Major Components of CPU

1. Arithmetic Logic Unit (ALU):
    - Performs arithmetic operations (addition, subtraction, multiplication, division).
    - Performs logical operations (AND, OR, NOT, XOR, comparison).
2. Control Unit (CU):
    - Directs the flow of data between memory, ALU, and I/O devices.
    - Generates control signals for execution of instructions.
    - Ensures proper sequencing of instruction cycle.
3. Registers:

- Small, high-speed storage units inside CPU.
- Examples:
  - Program Counter (PC): Holds address of next instruction.
  - Instruction Register (IR): Holds current instruction.
  - Accumulator (ACC): Stores intermediate results.
  - General Purpose Registers (R0, R1, …): Temporary storage.
  - Memory Address Register (MAR) & Memory Data Register (MDR): For memory communication.

---

3. CPU Organization

- Single Bus Organization: All units share a single communication path.
- Multiple Bus Organization: Separate buses for instructions, data, and control (faster).

---

4. Types of CPU Operations

- Data transfer (move, load, store).
- Arithmetic operations (+, –, ×, ÷).
- Logical operations (AND, OR, NOT, shift).
- Control operations (branch, jump, halt).

---

5. CPU Performance Factors

- Clock Speed: Number of instructions executed per second.
- Word Length: Number of bits processed at a time.
- Instruction Set Architecture (ISA): Defines operations CPU can perform (RISC, CISC).
- Number of Cores: Multiple processing units improve performance.

Execution of a Complete Instruction

n the blog post on the von Neumann Architecture, we established that the CPU consists of a control unit for processing the instructions sent to the CPU, the arithmetic logic unit for performing the operations specified in the instructions, and registers for storing instructions and data that are immediately required by the CPU.

The fetch-decode-execute cycle makes use of these components in addition to the memory unit.
In the fetch step of the cycle, the instructions are retrieved from the memory unit (RAM) and stored in the registers on the CPU. Next, the control unit decodes the instructions, which are then executed by the arithmetic and logic unit. The results of the instruction execution are sent back to RAM for storage, and the next instruction cycle begins.

What is CPU Clock Speed?

The number of instruction cycles a CPU can execute is stated as clock speed and measured in Hertz. If a CPU has a clock speed of 2 700 000 000 Hertz or 2.7 GHz, it executes 2.7 billion instruction cycles per second.

The Fetch Decode Execute Cycle Step By Step

In the following section, we will walk through the operations performed during the instruction cycle. Recall that a CPU has several different registers

- Program counter
- Memory address register
- Memory data register
- Current instruction register
- Accumulator

For an explanation of what these registers do, check out my post on von Neumann architecture.

What is a Fetch Cycle?

The instruction cycle begins with the fetch operation. The program counter keeps track of the next instruction to be processed. A fetch operation starts by loading the memory address of the next instruction into the program counter. In the next step, the processor transfers the address from the program counter to the memory address register and subsequently loads the data stored at that memory location into the memory data register. The program counter is automatically incremented to the next memory location unless the current instruction explicitly points to a different memory location for the next instruction.

Let's see how that works in practice using a concrete example:

1. The program counter initially points to the memory address 001
2. The memory address 001 is loaded into the memory address register by the processor
3. The processor next retrieves the instruction stored at memory address 001 and loads it into the memory data register.

4. Since the data contains the instruction “Get 203”, it is forwarded to the instruction register.
5. The program counter is incremented by 1, pointing to 002
6. The instruction in the instruction register is forwarded to the control unit.

What is an Execute Cycle?

After the fetch operation, the instruction cycle continues with the decode and execute portions. During the fetch stage, the control unit has been supplied with the instruction. It now needs to decode the instruction so that the processor can understand what to do next. In our example, I’ve supplied the instruction in plain English, such as “Get 203” which tells the processor to get the piece of data stored at memory location 203. In memory, the instruction is supplied in binary. For example, in a 16-bit memory, the first 4 bits may encode the operation to be performed, while the remaining bits specify the address from which to load the data.
Lastly, the processor will execute the instruction supplied. So if the instruction is to get some other piece of data, the “execute” action will consist of retrieving the data from the supplied memory address and storing it in the appropriate register. If the instruction specifies a calculation such as adding two numbers, the execution of the calculation will be handed off to the arithmetic and logic unit (ALU)

Let’s continue with our concrete example:

1. The Control Unit decodes the instruction and tells the processor to go to memory address 203 and fetch the piece of information stored there.
2. The address 203, is stored in the memory address register.

3. The data, the number 4, is stored in the memory data register.
4. The
5. Since the data is a number that will be necessary for a future operation and not another instruction, the number is stored in the accumulator.

This concludes the first fetch-decode-execute cycle. The processor starts the next cycle by fetching the next instruction stored in the program counter.
The fetching process is the same as in the previous cycle. This time, the instructions tell the processor to add the number stored at memory location 204 to the number currently stored in the accumulator.

After fetching the instruction, the processor retrieves the number stored at memory address 204 and places it in the accumulator while the previously stored number is forwarded to the arithmetic and logic unit (ALU).

Then, the number 3 is also forwarded to the arithmetic and logic unit, where the addition specified in the instruction is performed. Finally, the result is returned to the accumulator, where it will sit until the next instruction is executed.

In this example, we've used two instruction cycles to perform the addition. But modern processors may also load several pieces of data and perform calculations in one cycle.

What is an Interrupt in Computer Organization and Architecture?

As the term implies, an interrupt is a mechanism by which the normal course of actions of the processor is interrupted. This may be necessary for a variety of reasons, such as hardware failure or waiting for an I/O operation to complete.

Interrupts are part of a broader class of events known as exceptions. Exceptions essentially handle cases when the CPU encounters conditions that interfere with normal processing.

The main utility of interrupts lies in their ability to improve efficiency. Performing I/O operations is usually orders of magnitude slower than normal processing. If the computer had to communicate with an external device attached via USB, such as a flash drive, without the use of interrupts, the processor would have to wait until the i/O operation completes. The processor would spend thousands of instruction cycles just polling the peripheral device, asking if it was done processing without doing any useful work.

To make processing more efficient, the processor can receive an interrupt signal from the I/O device enabling it to work on something that is unrelated to the I/O operation while that operation is in progress.

Once the I/O device is done with its operations and requires communication with the processor, it sends an interrupt request signal to the processor. The processor then interrupts the execution of its current program and services the I/O device. This is achieved via a special device known as the interrupt handler. When the processor is finished with the I/O processing, it returns to the original process.

What happens during an Interrupt?

Once an Interrupt signal arrives, the processor has to perform a series of steps to handle the interrupt and continue processing:

- The CPU needs to save the current context as it exists in the registers to memory. Some processor architectures push the context onto a stack and then pop it off the stack. That way, they can restore the previous context in reverse order.
- Secondly, the CPU needs to retrieve the instructions of the interrupt handler from memory. The interrupt handler is basically a set of instructions stored in memory. Each type of interrupt has its own associated set of instructions.
- The CPU executes the instructions specified by the interrupt handler
- After concluding the operations specified by the interrupt handler, the CPU needs to restore the context of its previous operations by loading the associated instructions and data into the registers
- The CPU continues the previous flow of operations.

To include the handling of interrupts into the instruction cycle, an additional interrupt cycle is included.

Source: “Computer Organization and Architecture” 10th Edition by William Stallings

Multiple Interrupts

If multiple interrupts occur, there are essentially two options.

If an interrupt is currently being handled and a second interrupt occurs, the processor can push the second and all subsequent interrupts onto a stack and execute them in reverse sequential order. This has the disadvantage that we cannot prioritize interrupts. If an

I/O device that causes a notoriously long interrupt, like a printer, is currently executing, all other interrupts would have to wait.

Alternatively, interrupts can be associated with priorities. If an interrupt with a higher priority were to occur while a lower priority interrupt is being handled, the lower priority interrupt would itself be interrupted. The processor then would handle the higher-priority interrupt first before turning back to the lower-priority one. Naturally, the second approach engenders more complexity but is usually more efficient.

Types of Interrupts

So far, we have focused on interrupts as caused by I/O devices. In fact, there are several reasons for a processor to interrupt its course of action leading to different types of interrupt handlers.

Failure of Hardware

A computer relies on electricity. If there is a power outage or something overheats, the processor needs to be able to handle that case when the underlying hardware fails. The hardware failure handler also kicks in when there is an inconsistency in memory access. For example, if a piece of data is different in memory when it is accessed from when it was stored, it may cause system crashes.

Scheduled Interrupt

Interrupts may be generated by the processor on a regular basis to perform updates or other functions that may be necessary.

Program Condition

If an error occurs during the execution of a program, the program itself can trigger an interrupt. If you are a programmer, you probably have run into buffer overflow errors or other errors generated when executing your program. These errors are triggered when your

program attempts to do something that the processor cannot or will not handle. In that case, the processor will generate an interrupt or an exception. In fact, good programmers anticipate potential modes of failure and handle these through exceptions in their code.

I/O Devices

The controller of an I/O device can trigger interrupts as described previously. They either start a new interrupt by requesting service from the processor, signal normal completion of an I/O process, or indicate an error condition.

MULTIPLE BUS ORGANIZATION:

Rout, R=B, IRin

R4outA, R5outB, SelectA, Add, R6in, End

• Instruction execution proceeds as follows:

Step 1--> Contents of PC are passed through ALU using R=B control-signal and loaded into

MAR to start a memory Read operation. At the same time, PC is incremented by 4.

Step2--> Processor waits for MFC signal from memory.

Step3--> Processor loads requested-data into MDR, and then transfers them to IR.

Step4--> The instruction is decoded and add operation take place in a single step.

Note:

To execute instructions, the processor must have some means of generating the control signals needed in the

proper sequence. There are two approaches for this purpose:

1) Hardwired control and 2) Microprogrammed control.

Hardwired Control and Multi programmed Control:-

Introduction :

In computer architecture, the control unit is responsible for directing the flow of data and instructions within the CPU. There are two main approaches to implementing a control unit: hardwired and micro-programmed.

A hardwired control unit is a control unit that uses a fixed set of logic gates and circuits to execute instructions. The control signals for each instruction are hardwired into the control unit, so the control unit has a dedicated circuit for each possible instruction. Hardwired control units are simple and fast, but they can be inflexible and difficult to modify.

On the other hand, a micro-programmed control unit is a control unit that uses a microcode to execute instructions. The microcode is a set of instructions that can be modified or updated, allowing for greater flexibility and ease of modification. The control signals for each instruction are generated by a microprogram that is stored in memory, rather than being hardwired into the control unit.

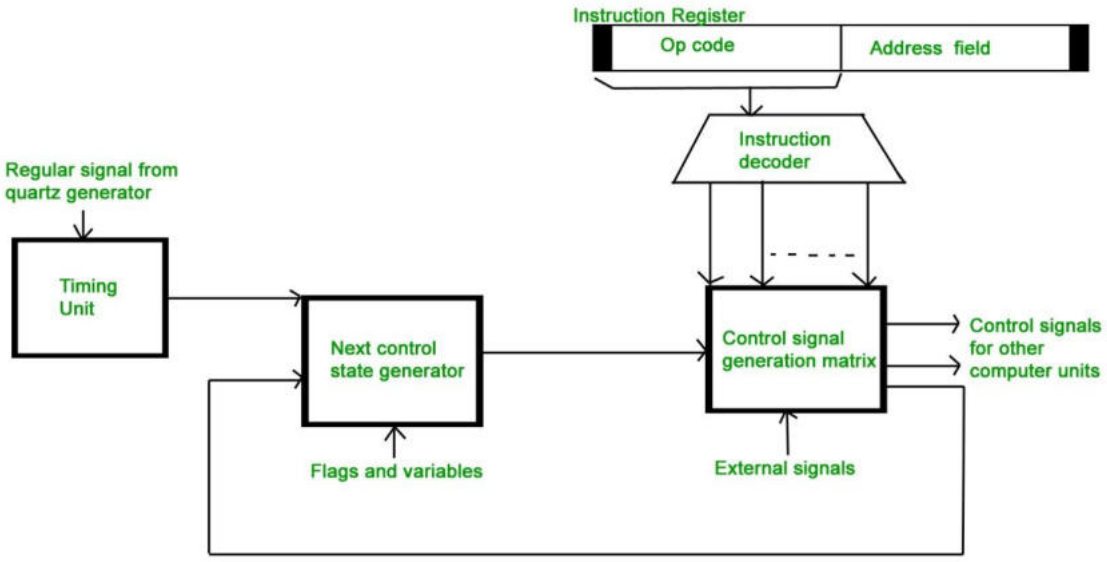
The control unit is the brain of the CPU, and it can be implemented in two ways: hardwired or micro-programmed. Understanding the differences between these implementations is essential for those studying computer organization.

If we talk about Micro-programmed control units they are generally slower than hardwired control units because they require an extra step of decoding the microcode to generate control signals, but they are more flexible and easier to modify. They are commonly used in modern CPUs because they allow for easier implementation of complex instruction sets and better support for instruction set extensions.

To execute an instruction, the control unit of the CPU must generate the required control signal in the proper sequence. There are two approaches used for generating the control signals in proper sequence as Hardwired Control unit and the Micro-programmed control unit.

Hardwired Control Unit: The control hardware can be viewed as a state machine that changes from one state to another in every clock cycle, depending on the contents of the instruction register, the condition codes, and the external inputs. The outputs of the state machine are the control signals. The sequence of the operation carried out by this machine is determined by the wiring of the logic elements and hence named “hardwired”.

- Fixed logic circuits that correspond directly to the Boolean expressions are used to generate the control signals.
- Hardwired control is faster than micro-programmed control.
- A controller that uses this approach can operate at high speed.
- RISC architecture is based on the hardwired control unit



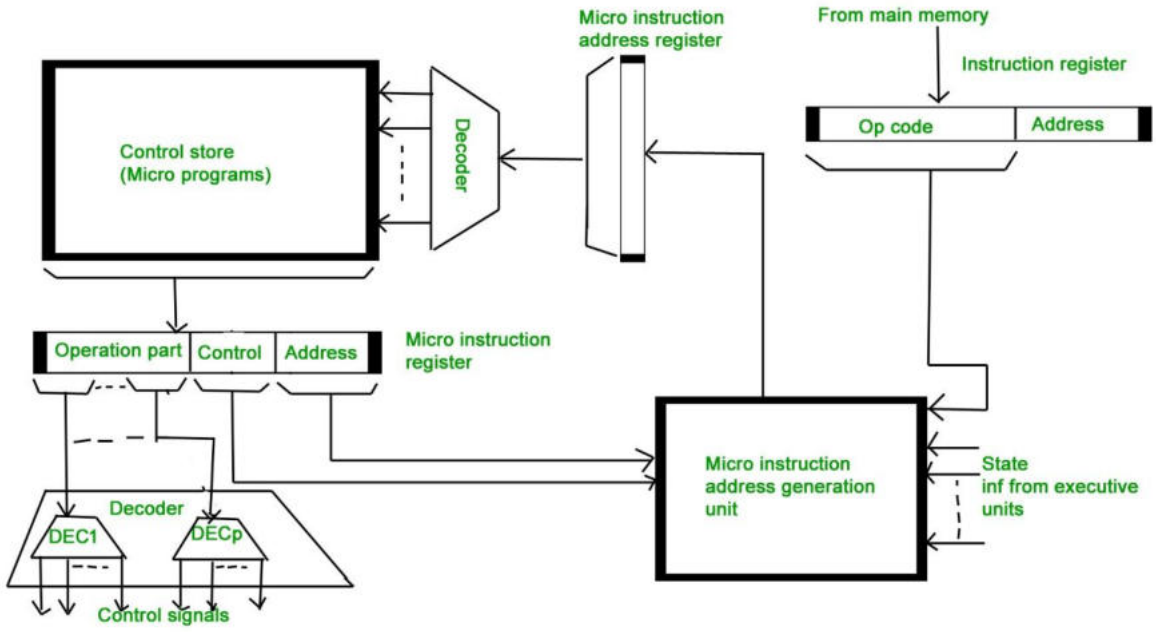
Micro-programmed Control Unit -

- The control signals associated with operations are stored in special memory units inaccessible by the programmer as Control Words.
- Control signals are generated by a program that is similar to machine language programs.
- The micro-programmed control unit is slower in speed because of the time it takes to fetch microinstructions from the control memory.

Some Important Terms

1. Control Word: A control word is a word whose individual bits represent various control signals.
2. Micro-routine: A sequence of control words corresponding to the control sequence of a machine instruction constitutes the micro-routine for that instruction.
3. Micro-instruction: Individual control words in this micro-routine are referred to as microinstructions.

4. Micro-program: A sequence of micro-instructions is called a micro-program, which is stored in a ROM or RAM called a Control Memory (CM).
5. Control Store: the micro-routines for all instructions in the instruction set of a computer are stored in a special memory called the Control Store.



The differences between hardwired and micro-programmed control units:

	Hardwired Control Unit	Micro-programmed Control Unit
Implementation	Fixed set of logic gates and circuits	Microcode stored in memory

	Hardwired Control Unit	Micro-programmed Control Unit
Flexibility	Less flexible, difficult to modify	More flexible, easier to modify
Instruction Set	Supports limited instruction sets	Supports complex instruction sets
Complexity of Design	Simple design, easy to implement	Complex design, more difficult to implement
Speed	Fast operation	Slower operation due to microcode decoding
Debugging and Testing	Difficult to debug and test	Easier to debug and test
Size and Cost	Smaller size, lower cost	Larger size, higher cost

	Hardwired Control Unit	Micro-programmed Control Unit
Maintenance and Upgradability	Difficult to upgrade and maintain	Easier to upgrade and maintain

Types of Micro-programmed Control Unit - Based on the type of Control Word stored in the Control Memory (CM), it is classified into two types :

1. Horizontal Micro-programmed Control Unit :
The control signals are represented in the decoded binary format that is 1 bit/CS. Example: If 53 Control signals are present in the processor then 53 bits are required. More than 1 control signal can be enabled at a time.

- It supports longer control words.
- It is used in parallel processing applications.
- It allows a higher degree of parallelism. If degree is n, n CS is enabled at a time.
- It requires no additional hardware(decoders). It means it is faster than Vertical Microprogrammed.
- It is more flexible than vertical microprogrammed

2. Vertical Micro-programmed Control Unit :
The control signals are represented in the encoded binary format. For N control signals- Log2(N) bits are required.

- It supports shorter control words.

- It supports easy implementation of new control signals therefore it is more flexible.
- It allows a low degree of parallelism i.e., the degree of parallelism is either 0 or 1.
- Requires additional hardware (decoders) to generate control signals, it implies it is slower than horizontal microprogrammed.
- It is less flexible than horizontal but more flexible than that of a hardwired control unit.

Note: Types of Control Unit in descending order of speed :

Hardwired control unit > Horizontal microprogrammed CU > Vertical microprogrammed CU

UNIT-IV: THE MEMORY MANAGEMENT SYSTEM

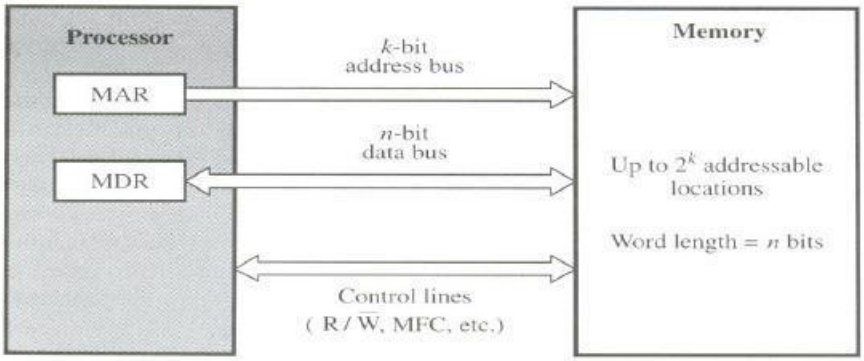
Basic Concepts, Semiconductor RAM, Types of Read-only Memory (ROM), Cache Memory, Performance Considerations, Virtual Memory, Secondary Storage.

4.1 Basic Concepts:

The maximum size of the memory that can be used in any computer is determined by the addressing scheme.

Address	Memory Locations
16 Bit	$2^{16} = 64$ K
32 Bit	$2^{32} = 4$G (Giga)
40 Bit	$2^{40} =$ IT (Tera)

Fig: Connection of Memory to Processor:



If MAR is k bits long and MDR is n bits long, then the memory may contain upto 2K addressable locations and the n-bits of data are transferred between the memory and processor.

This transfer takes place over the processor bus.

The processor bus has,

- Address Line
- Data Line
- Control Line (R/W, MFC – Memory Function Completed)

The control line is used for co-ordinating data transfer.

The processor reads the data from the memory by loading the address of the required memory location into MAR and setting the R/W line to 1.

The memory responds by placing the data from the addressed location onto the data lines and confirms this action by asserting MFC signal.

Upon receipt of MFC signal, the processor loads the data onto the data lines into MDR register.

The processor writes the data into the memory location by loading the address of this location into MAR and loading the data into MDR sets the R/W line to 0.

 Measures for the speed of a memory:

 memory access time.

 It is the time that elapses between the initiation of an Operation and the completion of that operation.

 memory cycle time.

 It is the minimum time delay that required between the initiation of the two successive memory operations.

RAM (Random Access Memory):

In RAM, if any location that can be accessed for a Read/Write operation in fixed amount of time, it is independent of the location's address.

Cache Memory:

It is a small, fast memory that is inserted between the larger slower main memory and the processor.

It holds the currently active segments of a program and their data.

Virtual memory:

The address generated by the processor does not directly specify the physical locations in the memory.

The address generated by the processor is referred to as a virtual / logical address.

The virtual address space is mapped onto the physical memory where data are actually stored.

The mapping function is implemented by a special memory control circuit is often called the memory management unit.

Only the active portion of the address space is mapped into locations in the physical memory.

The remaining virtual addresses are mapped onto the bulk storage devices used, which are usually magnetic disk.

As the active portion of the virtual address space changes during program execution, the memory management unit changes the mapping function and transfers the data between disk and memory.

Thus, during every memory cycle, an address processing mechanism determines whether the addressed in function is in the physical memory unit.

If it is, then the proper word is accessed and execution proceeds. If it is not, a page of words containing the desired word is transferred from disk to memory.

This page displaces some page in the memory that is currently inactive.

Semiconductor RAM

Semi-Conductor memories are available is a wide range of speeds. Their cycle time ranges from 100ns to 10ns.

INTERNAL ORGANIZATION OF MEMORY CHIPS:

Memory cells are usually organized in the form of array, in which each cell is capable of storing one bit of information.

Each row of cells constitute a memory word and all cells of a row are connected to a common line called as word line.

The cells in each column are connected to Sense / Write circuit by two bit lines.

The Sense / Write circuits are connected to data input or output lines of the chip. During a write operation, the sense / write circuit receive input information and store it in the cells of the selected word.

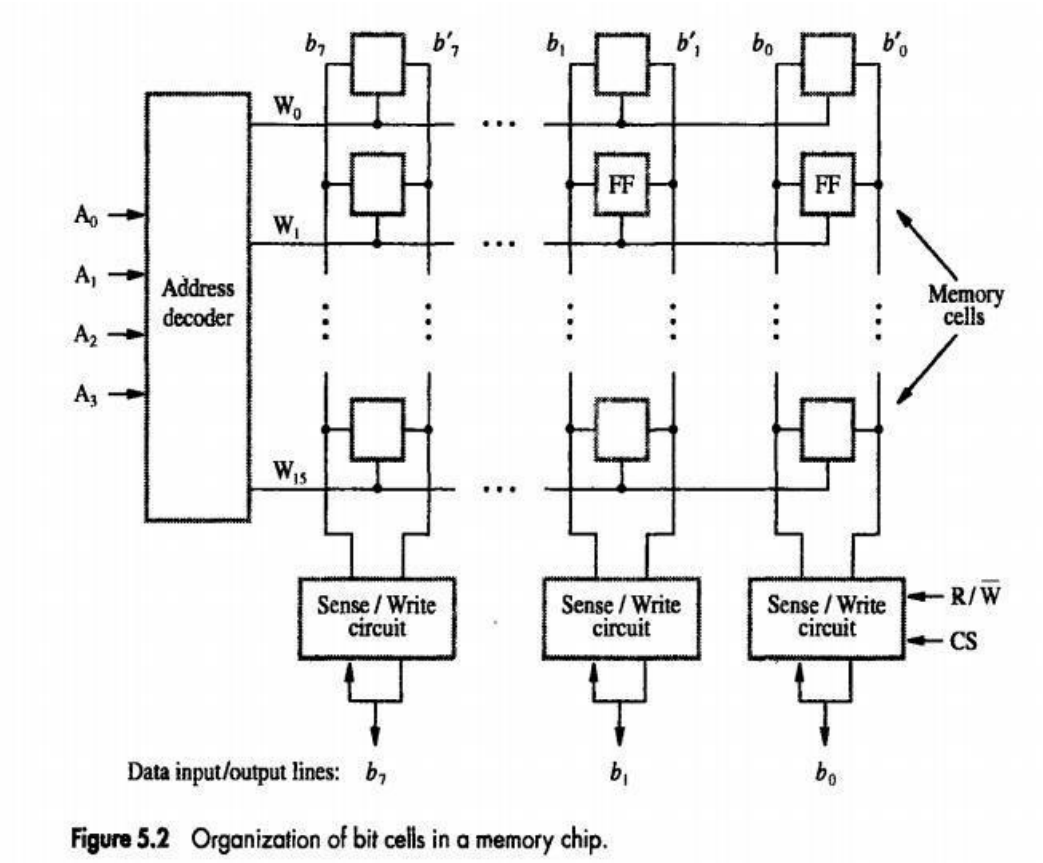


Figure 5.2 Organization of bit cells in a memory chip.

The data input and data output of each senses / write ckt are connected to a single bidirectional data line that can be connected to a data bus of the cptr.

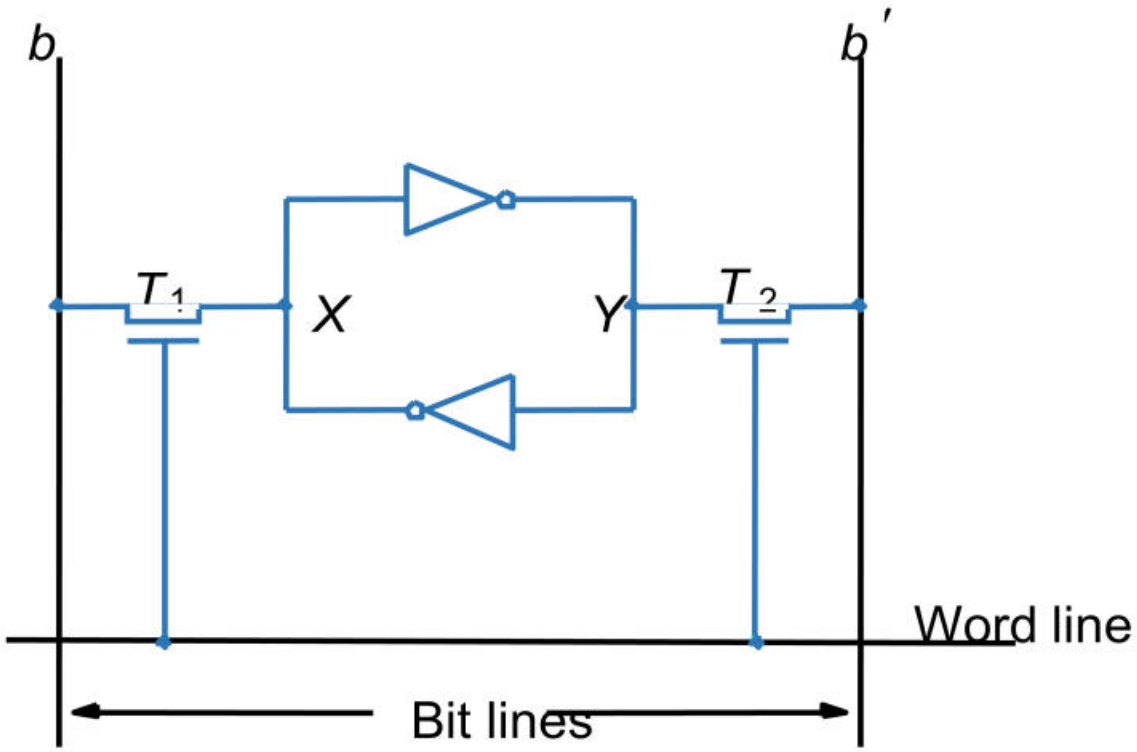
R / W Specifies the required operation.

CS Chip Select input selects a given chip in the multi-chip memory system

Static Memories:

Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memory.

Fig: Static RAM cell



- Two inverters are cross connected to form a batch.
- The batch is connected to two bit lines by transistors T1 and T2.
- These transistors act as switches that can be opened / closed under the control of the word line.
- When the word line is at ground level, the transistors are turned off and the latch retain its state.

Read Operation:

- In order to read the state of the SRAM cell, the word line is activated to close switches T1 and T2.
- If the cell is in state 1, the signal on bit line b is high and the signal on the bit line b is low. Thus b and b are complements of each other.
- Sense / write circuit at the end of the bit line monitors the state of b and b' and set the output accordingly.

Write Operation:

 The state of the cell is set by placing the appropriate value on bit line b and its complement on b and then activating the word line. This forces the cell into the corresponding state.

 The required signal on the bit lines are generated by Sense / Write circuit.

Fig:CMOS cell (Complementary Metal oxide Semi Conductor):

Figure 5.4 A static RAM cell.

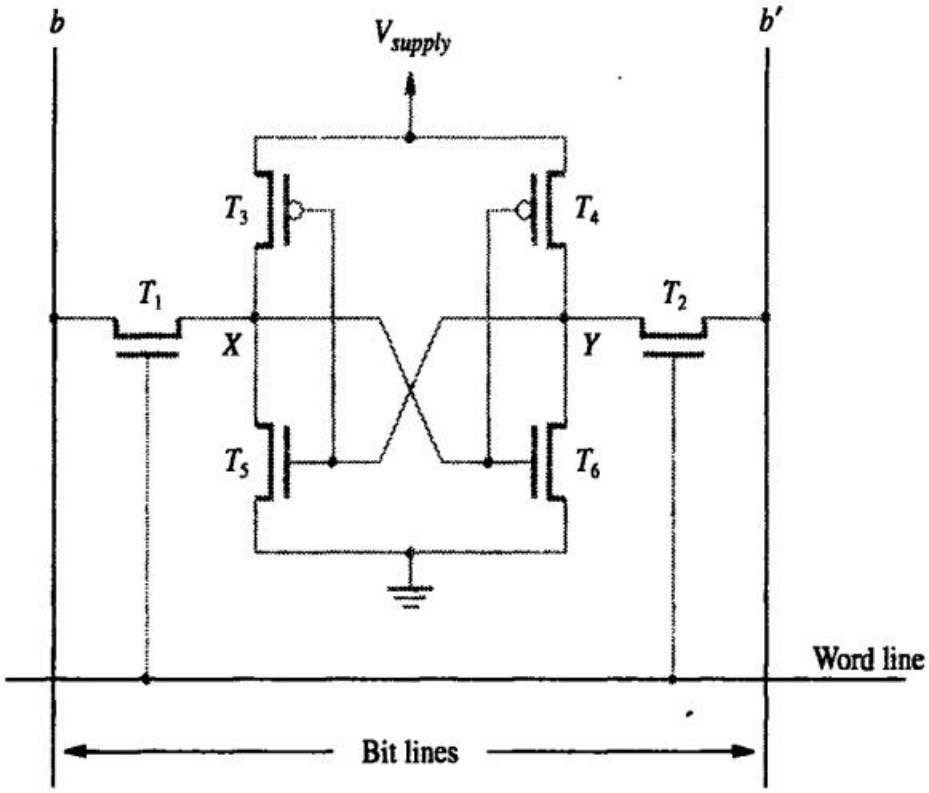


Figure 5.5 An example of a CMOS memory cell.

 Transistor pairs (T3, T5) and (T4, T6) form the inverters in the latch.

 In state 1, the voltage at point X is high by having T5, T6 on and T4, T5 are OFF.

 Thus T1 and T2 returned ON (Closed), bit line b and b will have high and low signals respectively.

 The CMOS requires 5V (in older version) or 3.3.V (in new version) of power supply voltage.

 The continuous power is needed for the cell to retain its state

Merit :

 It has low power consumption because the current flows in the cell only when the cell is being activated accessed.

 Static RAM's can be accessed quickly. It access time is few Nano seconds.

Demerit:

 SRAM's are said to be volatile memories because their contents are lost when the power is interrupted.

Asynchronous DRAMS:-

 Less ex pensive RAMs can be implemented if simplex call s are used such c ell s cannot retain their state indefinitely. Hence they are called Dynamic RAM's (DRAM).

 The information stored in a dynamic memory cell in the form of a charge on a capacitor and this charge can be maintained only for tens of Milliseconds.

 The contents must be periodically refreshed by restoring by restoring this capacitor charge to its full value.

 In order to store information in the cell, the transistor T is turned on & the appropriate voltage is applied to the bit line, which charges the capacitor.

 After the transistor is turned off, the capacitor begins to discharge which is caused by the capacitor's own leakage resistance.

 Hence the information stored in the cell can be retrieved correctly before the threshold value of the capacitor drops down.

Fig:A single transistor dynamic Memory cell

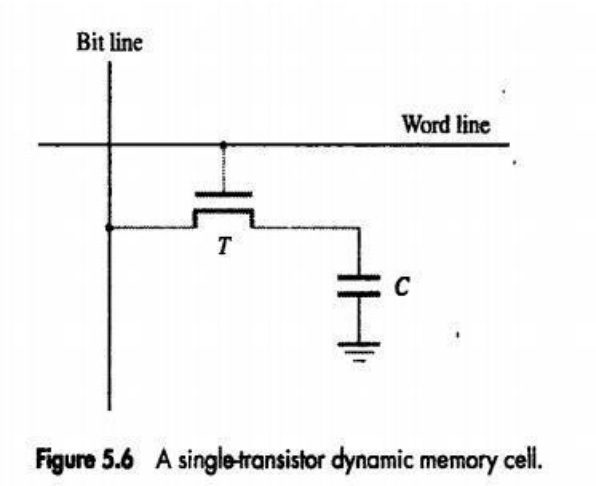


Figure 5.6 A single-transistor dynamic memory cell.

During are ad operation, the transistor is turned „o n" & a sense amplifier connected to the bit line detects whether the charge on the capacitor is above the threshold value.

If charge on capacitor > threshold value -> Bit line will have logic value 1.

If charge on capacitor < threshold value -> Bit line will set to logic value 0.

Fig:Internal organization of a 2M X 8 dynamic Memory chip

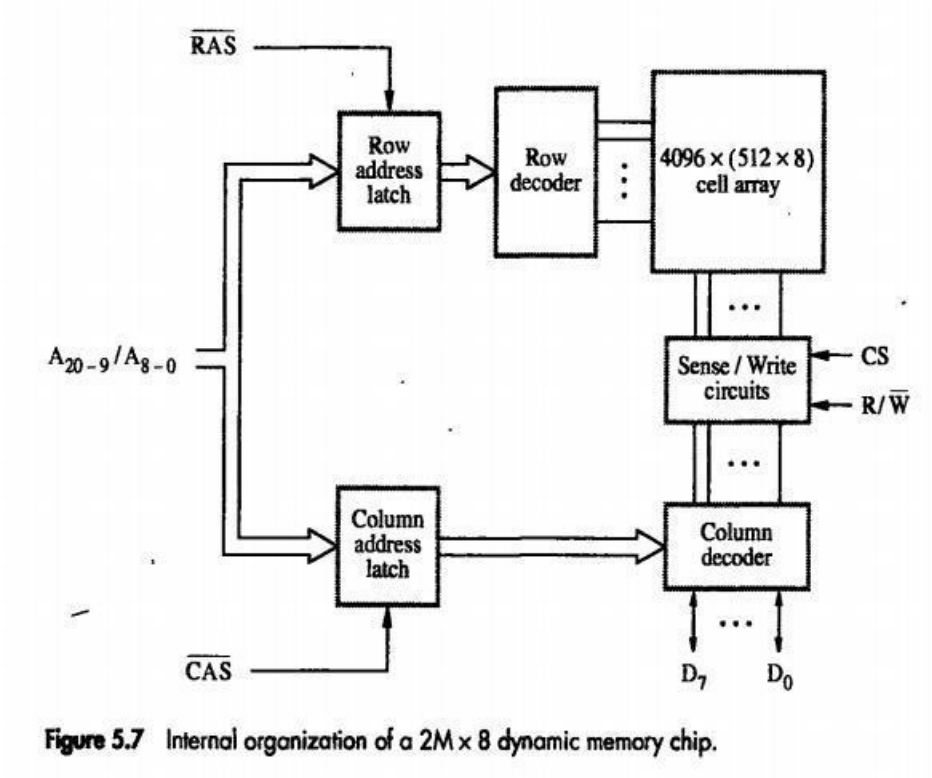


Figure 5.7 Internal organization of a 2M x 8 dynamic memory chip.

DESCRIPTION:

- The 4 bit cells in each row are divided into 512 groups of 8.
- 21 bit address is needed to access a byte in the memory(12 bit→To select a row,9 bit→Specify the group of 8 bits in the selected row).

  A_{8-0} →Row address of a byte.
  A_{20-9} →Column address of a byte.

- During Read/ Write operation ,the row address is applied first. It is loaded into the row address latch in response to a signal pulse on **Row Address Strobe(RAS)** input of the chip.
- When a Read operation is initiated, all cells on the selected row are read and refreshed.
- Shortly after the row address is loaded,the column address is applied to the address pins & loaded into **Column Address Strobe(CAS).**
- The information in this latch is decoded and the appropriate group of 8 Sense/Write circuits are selected.
- R/W =1(read operation)→The output values of the selected circuits are transferred to the data lines D0 - D7.
- R/W =0(write operation)→The information on D0 - D7 are transferred to the selected circuits.

- RAS and CAS are active low so that they cause the latching of address when they change from high to low. This is because they are indicated by RAS & CAS.
- To ensure that the contents of a DRAM 's are maintained, each row of cells must be accessed periodically.
- Refresh operation usually perform this function automatically.
- A specialized memory controller circuit provides the necessary control signals RAS & CAS, that govern the timing.
- The processor must take into account the delay in the response of the memory. Such memories are referred to as **Asynchronous DRAM's.**

Fast Page Mode:

Transferring the bytes in sequential order is achieved by applying the consecutive sequence of column address under the control of successive CAS signals.

This scheme allows transferring a block of data at a faster rate. The block of transfer capability is called as Fast Page Mode.

Synchronous DRAM:

 Here the operations are directly synchronized with clock signal.

 The address and data connections are buffered by means of registers.

 The output of each sense amplifier is connected to a latch.

 A Read operation causes the contents of all cells in the selected row to be loaded in these latches.

Fig: Synchronous DRAM

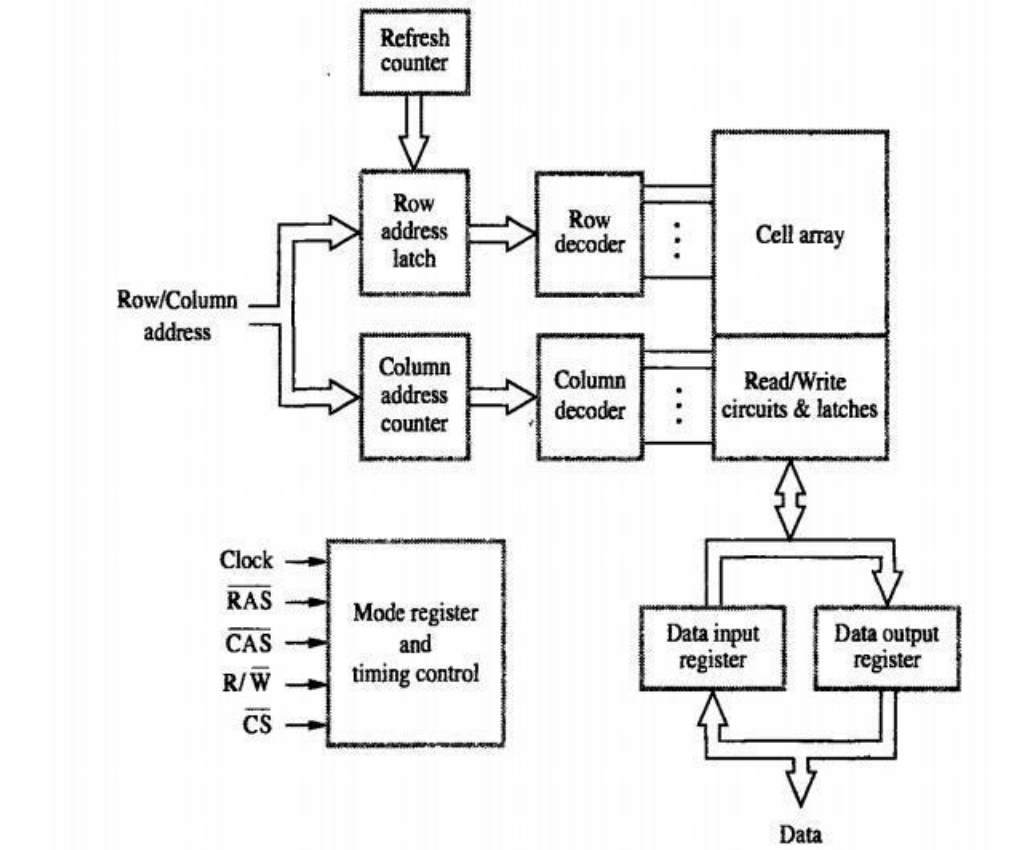
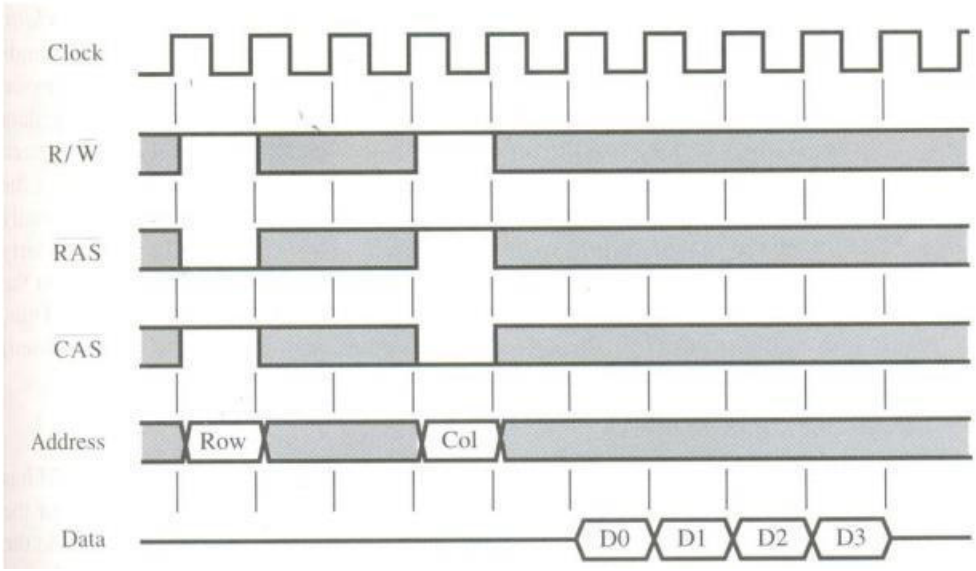


Figure 5.8 Synchronous DRAM.

 Data held in the latches that correspond to the selected columns are transferred into the data output register, thus becoming available on the data output pins.

Fig: Timing Diagram Burst Read of Length 4 in an SDRAM

Fig:Timing Diagram →Burst Read of Length 4 in an SDRAM



- First, the row address is latched under control of RAS signal.
- The memory typically takes 2 or 3 clock cycles to activate the selected row.
- Then the column address is latched under the control of CAS signal.
- After a delay of one clock cycle, the first set of data bits is placed on the data lines.
- The SDRAM automatically increments the column address to access the next 3 sets of bits in the selected row, which are placed on the data lines in the next 3 clock cycles.

Latency & Bandwidth:

A good indication of performance is given by two parameters. They are,

- Latency
- Bandwidth

Latency:

- It refers to the amount of time it takes to transfer a word of data to or from the memory.
- For a transfer of single word, the latency provides the complete indication of memory performance.
- For a block transfer, the latency denotes the time it takes to transfer the first word of data.

Bandwidth:

 It is defined as the number of bits or bytes that can be transferred in one second.

 Bandwidth mainly depends upon the speed of access to the stored data & on the number of bits that can be accessed in parallel.

Double Data Rate SDRAM (DDR-SDRAM):

 The standard SDRAM performs all actions on the rising edge of the clock signal.

 The double data rate SDRAM transfer data on both the edges (loading edge, trailing edge).

 The Bandwidth of DDR-SDRAM is doubled for long burst transfer.

 To make it possible to access the data at high rate, the cell array is organized into two banks.

 Each bank can be accessed separately.

 Consecutive words of a given block are stored in different banks.

 Such interleaving of words allows simultaneous access to two words that are transferred on successive edge of the clock.

Larger Memories:

Dynamic Memory System:

 The physical implementation is done in the form of Memory Modules.

 If a large memory is built by placing DRAM chips directly on the main system printed circuit board that contains the processor, often referred to as Motherboard; it will occupy large amount of space on the board.

 These packaging considerations have led to the development of larger memory unit s known as SIMMs & DIMMs.

o SIMM-Single Inline memory Module

o DIMM-Dual Inline memory Module

 SIMM & DIMM consists of several memory chips on a separate small board that plugs vertically into single socket on the motherboard.

MEMORY SYSTEM CONSIDERATION:

M.JOSHNA

To reduce the number of pins, the dynamic memory chips use multiplexed

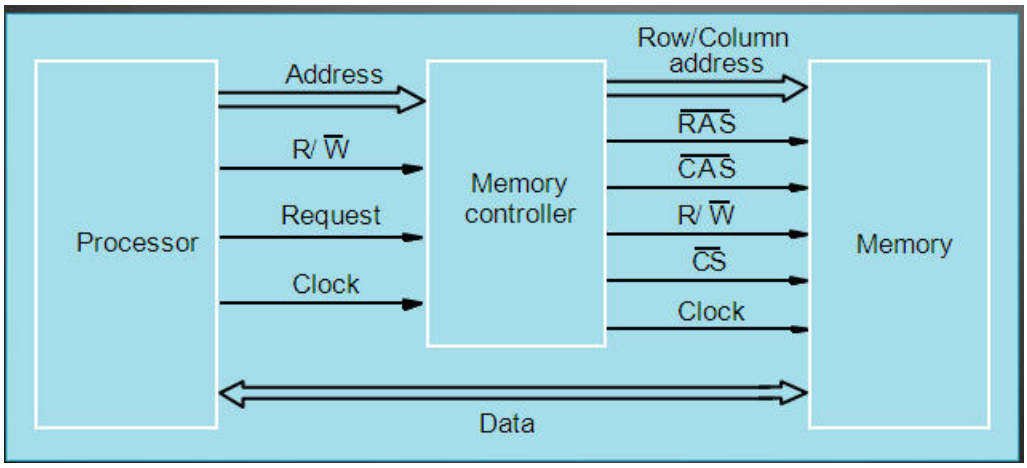
Address inputs. The address is divided into two parts. They are,

 High Order Address Bit (Select a row in cell array & it is provided first and latched into memory chips under the control of RAS signal).

 Low Order Address Bit(Selects a column and they are provided on same Address pins and latched using CAS signals).

The Multiplexing of address bit is usually done by Memory Controller Circuit.

Fig:Use of Memory Controller



 The Controller accepts a complete address & R/W signal from the processor, under the control of a Request signal which indicates that a memory access operation is needed.

 The Controller then forwards the row & column portions of the address to the memory and generates RAS &CAS signals.

 It also sends R/W &CS signals to the memory. The CS signal is usually active low, hence it is shown as CS.

Refresh Overhead:

All dynamic memories have to be refreshed. In DRAM ,the period for refreshing all rows is 16ms whereas 64ms in SDRAM.

Eg:Given a cell array of 8K(8192).

Clock cycle=4

M.JOSHNA

Clock Rate=133MHZ

No of cycles to refresh all rows =8192*4 =32,768

Time needed to refresh all rows=32768/133*10=246*10-6 sec=0.246sec

Refresh Overhead=0.246/64

Refresh Overhead =0.0038

Rambus Memory:

 The usage of wide bus is expensive.

 Rambus developed the implementation of narrow bus

Rambus technology is a fast signaling method used to transfer information between chips.

 Instead of using signals that have voltage levels of either 0 or Vsupply to represent the logical values, the signals consists of much smaller voltage swings around a reference voltage Vref.

 The reference Voltage is about 2V and the two logical values are represented by 0.3V swings above and below Vref

 This type of signaling is generally is known as Differential Signalling.

 Rambus provides a complete specification for the design of communication links (Special Interface circuits) called as Rambus Channel.

 Rambus memory has a clock frequency of 400MHZ. The data are transmitted on both the edges of the clock so that the effective data transfer rate is 800MHZ.

 The circuitry needed to interface to the Rambus channel is included on the chip. Such chips are known as Rambus DRAMs (RD RAM).

Rambus channel has,

 9 Data lines(1-8 Transfer the data,9th line parity checking).

 Control line

 Power line

A two channel rambus has 18 data lines which has no separate address lines. It is

also called as Direct RD RAM's. Communication between processor or some other device that can serves as a master and RDRAM modules are serves as slaves, is carried out by means of packets transmitted on the data lines.

There are 3 types of packets. They are,

 Request

 Acknowledge

 Data

Types of Read-only Memory (ROM)

 Both SRAM and DRAM chips are volatile, which means that they lose the stored information if power is turned off.

 Many application requires Non-volatile memory (which retain the stored

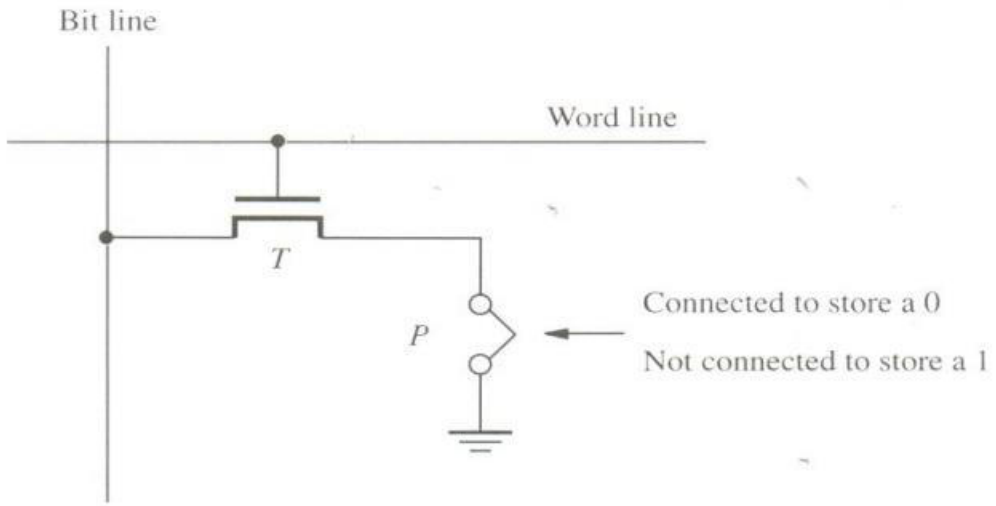
information if power is turned off).

 Eg: Operating System software has to be loaded from disk to memory which

 requires the program that boots the Operating System ie. It requires non-volatile memory.

 Non-volatile memory is used in embedded system.

 Since the normal operation involves only reading of stored data ,a memory of this type is called ROM.



At Logic value '0' Transistor (T) is connected to the ground point(P). Transistor switch is closed & voltage on bitline nearly drops to zero.

At Logic value '1' Transistor switch is open.

The bitline remains at high voltage.

To read the state of the cell, the word line is activated.

A Sense circuit at the end of the bitline generates the proper output value.

Types of ROM:

Different types of non-volatile memory are,

- PROM
- EPROM
- EEPROM
- Flash Memory

PROM:-Programmable ROM:

- PROM allows the data to be loaded by the user.
- Programm ability is achieved b y inserting a fuse at point P in a ROM cell .
- Before it is programmed, the memory contains all 0‟s
- The user can insert 1s at the required location by burning out t he fuse at these locations using high-current pulse.
- This process is irreversible.

Merit:

- It provides flexibility.
- It is faster.
- It is less expensive because they can be programmed directly by the user.

EPROM:-Erasable reprogrammable ROM:

- EPROM allows the stored data to be erased and new data to be loaded.
- In an EPROM cell , a connection to ground is always made at P and a special transistor is used, which has the ability to function either as a normal transistor or as a disabled transistor that is always turned off.

 This transistor can be programmed to behave as a permanently open switch, by injecting charge into it that becomes trapped inside.

 Erasure requires dissipating the charges trapped in the transistor of memory cells. This can be done by exposing the chip to ultra-violet light, so that EPROM chips are mounted in packages that have transparent windows.

Merits:

 It provides flexibility during the development phase of digital system.

 It is capable of retaining the stored information for a long time.

Demerits:

 The chip must be physically removed from the circuit for reprogramming and its entire contents are erased by UV light.

Electrically erasable programmable read-only memory (EEPROM)

Electrically erasable programmable read-only memory (EEPROM) chips that can be electrically programmed and erased. EEPROMs are typically changed 1 byte at time. Erasing EEPROM takes typically quite long. The drawback of EEPROM is their speed. EEPROM chips are too slow to use in many products that make quick changes to the data stored on the chip. Typically EEPROMs are found in electronics devices for storing the small amounts of nonvolatile data in applications where speed is not the most important. Small EEPROMs with serial interfaces are commonly found in many electronics devices.

Flash Memory:

 In EEPROM, it is possible to read & write the contents of a single cell.

 In Flash device, it is possible to read the contents of a single cell but it is only possible to write the entire contents of a block.

 Prior to writing, the previous contents of the block are erased. Eg. In MP3 player, the flash memory stores the data that represents sound.

 Single flash chips cannot provide sufficient storage capacity for embedded system application.

 There are 2 methods for implementing larger memory modules consisting of number of chips. They are,

o Flash Cards

o Flash Drives.

Merits:

 Flash drives have greater density which leads to higher capacity & low cost per bit.

 It requires single power supply voltage & consumes less power in their operation.

Flash Cards:

 One way of constructing larger module is to mount flash chips on a small card.

 Such flash card have standard interface.

 The card is simply plugged into a conveniently accessible slot.

 Its memory size are of 8,32,64MB.

 Eg:A minute of music can be stored in 1MB of memory. Hence 64MB flash cards

can store an hour of music.

Flash Drives:

 Larger flash memory module can be developed by replacing the hard disk drive.

 The flash drives are designed to fully emulate the hard disk.

 The flash drives are solid state electronic devices that have no movable parts.

Merits:

 They have shorter seek and access time which results in faster response.

 They have low power consumption which makes them attractive for battery driven application.

 They are insensitive to vibration.

Demerits:

 The capacity of flash drive (<1GB) is less than hard disk(>1GB).

 It leads to higher cost per bit.

 Flash memory will deteriorate after it has been written a number of times(typically at least 1 million times.)

SPEED,SIZE COST:

Characteristics	SRAM	DRAM	Magnetis Disk
Speed	Very Fast	Slower	Much slower than DRAM
Size	Large	Small	Small
Cost	Expensive	Less Expensive	Low price

Magnetic Disk:

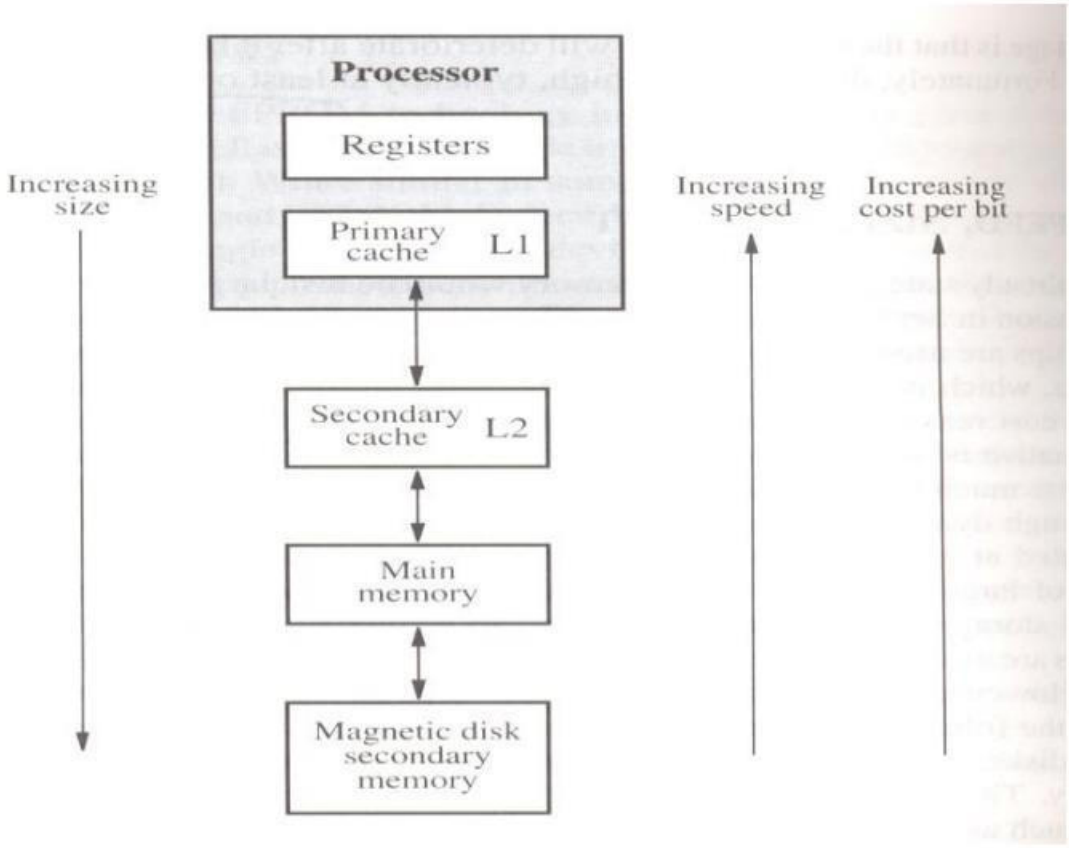
- A huge amount of cost effective storage can be provided by magnetic disk;The main memory can be built with DRAM which leaves SRAM's to be used in smaller units where speed is of essence.

Memory	Speed	Size	Cost
Registers	Very high	Lower	Very Lower
Primary cache	High	Lower	Low
Secondary cache	Low	Low	Low
Main memory	Lower than Seconadry cache	High	High
Secondary Memory	Very low	Very High	Very High

4.4 Cache Memory

Ideally, computer memory should be fast, large and inexpensive. Unfortunately, it is impossible to meet all the three requirements simultaneously. Increased speed and size are achieved at increased cost. Very fast memory systems can be achieved if SRAM chips are used. These chips are expensive and for the cost reason it is impracticable to build a large main memory using SRAM chips. The alternative used to use DRAM chips for large main memories. The processor fetches the code and data from the main memory to execute the program. The DRAMs which form the main memory are slower devices. So it is necessary to insert wait states in memory read/write cycles. This reduces the speed of execution. The solution for this problem is in the memory system small section of SRAM is added along with the main memory, referred to as cache memory. The program which is to be executed is loaded in the main memory, but the part of the program and data accessed from the cache memory. The cache controller looks after this swapping between main memory and cache memory with the help of DMA controller, Such cache memory is called **secondary cache**. Recent processors have the built in cache memory called **primary cache.** The size of the memory is still small compared to the demands of the large programs with the voluminous data. A solution is provided by using secondary storage, mainly magnetic disks and magnetic tapes to implement large memory spaces, which is available at reasonable prices. To make efficient computer system it is not possible to rely on a single memory component, but to employ a memory hierarchy which uses all different types of memory units that gives efficient computer system. A typical memory hierarchy is illustrated below in the figure:

Fig:Memory Hierarchy



• Fastest access is to the data held in processor registers. Registers are at the top of the memory hierarchy.

• Relatively small amount of memory that can be implemented on the processor chip. This is processor cache.

• Two levels of cache. Level 1 (L1) cache is on the processor chip. Level 2 (L2) cache is in between main memory and processor.

• Next level is main memory, implemented as SIMMs. Much larger, but much slower than cache memory.

• Next level is magnetic disks. Huge amount of inexpensive storage.

• Speed of memory access is critical, the idea is to bring instructions and data that will be used in the near future as close to the processor as possible.

The effectiveness of cache mechanism is based on the property of " Locality of reference'.

Locality of Reference:

Many instructions in the localized areas of the program are executed repeatedly during some time period and remainder of the program is accessed relatively infrequently.

It manifests itself in 2 ways. They are,

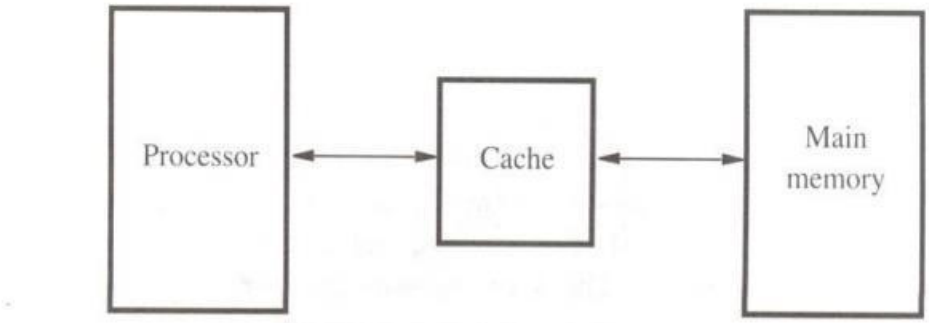
 Temporal(The recently executed instruction are likely to be executed again very soon.)

 Spatial(The instructions in close proximity to recently executed instruction are also likely to be executed soon.) If the active segment of the program is placed in cache memory, then the total execution time can be reduced significantly.

The term Block refers to the set of contiguous address locations of some size.

The cache line is used to refer to the cache block.

Fig: Use of Cache Memory



 The Cache memory stores a reasonable number of blocks at a given time but this number is small compared to the total number of blocks available in Main Memory.

 The correspondence between main memory block and the block in cache memory is specified by a mapping function.

 The Cache control hardware decide that which block should be removed to create space for the new block that contains the referenced word.

 The collection of rule for making this decision is called the replacement algorithm.

 The cache control circuit determines whether the requested word currently exists in the cache.

 If it exists, then Read/Write operation will take place on appropriate cache location. In this case Read/Write hit will occur.

 In a Read operation, the memory will not involve.

 The write operation is proceeding in 2 ways. They are,

o Write-through protocol

o Write-back protocol

Write-through protocol:

Here the cache location and the main memory locations are updated simultaneously.

Write-back protocol:

 This technique is to update only the cache location and to mark it as with associated flag bit called dirty/modified bit

 The word in the main memory will be updated later, when the block containing this marked word is to be removed from the cache to make room for a new block.

 If the requested word currently not exists in the cache during read operation, then read miss will occur.

 To overcome the read miss Load – through / Early restart protocol is used.

Read Miss:

The block of words that contains the requested word is copied from the main memory into cache.

Load – through:

 After the entire block is loaded into cache, the particular word requested is forwarded to the processor.

 If the requested word not exists in the cache during write operation, then Write Miss will occur.

 If Write through protocol is used, the information is written directly into main memory.

 If Write back protocol is used then block containing the addressed word is first brought into the cache and then the desired word in the cache is over-written with the new information.

Cache Memories – Mapping Functions

First generation processors, those designed with vacuum tubes in 1950 or those designed with integrated circuits in 1965 or those designed as microprocessors in 1980 were generally about the same speed as main memory. On such processors, this naive model was perfectly

reasonable. By 1970, however, transistorized supercomputers were being built where the central processor was significantly faster than the main memory, and by 1980, the difference had increased, although it took several decades for the performance difference to reach today's extreme.

Solution to this problem is to use what is called a ***cache memory*** between the central processor and the main memory. Cache memory takes advantage of the fact that, with any of the memory technologies available for the past half century, we have had a choice between building large but slow memories or small but fast memories. This was known as far back as 1946, when Berks, Goldstone and Von Neumann proposed the use of a memory hierarchy, with a few fast registers in the central processor at the top of the hierarchy, a large main memory in the middle, and a library of archival data, stored off-line, at the very bottom.

A cache memory sits between the central processor and the main memory. During any particular memory cycle, the cache checks the memory address being issued by the processor. If this address matches the address of one of the few memory locations held in the cache, the cache handles the memory cycle very quickly; this is called a **cache hit.** If the address does not, then the memory cycle must be satisfied far more slowly by the main memory; this is called a **cache miss.**

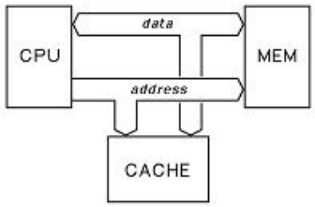


Figure 7:Adding a cache to the naive view

The correspondence between the main memory and cache is specified by a Mapping function. When the cache is full and a memory word that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that constitutes the Replacement algorithm.

Mapping Functions

There are three main mapping techniques which decides the cache organization:

1. Direct-mapping technique

2. Associative mapping Technique

3. Set associative mapping technique

To discuss possible methods for specifying where memory blocks are placed in the cache, we use a specific small example, a cache consisting of 128 blocks of 16 word each, for a total of

2048(2k) word, and assuming that the main memory is addressable by a 16-bit address. The main memory has 64k word, which will be viewed as 4K blocks of 16 word each, the consecutive addresses refer to consecutive word.

Direct Mapping Technique

The cache systems are divided into three categories, to implement cache system. As shown in figure, the lower order 4-bits from 16 words in a block constitute a **word field**. The second field is known as **block field** used to distinguish a block from other blocks. Its length is 7-bits, when a new block enters the cache; the 7-bit cache block field determines the cache position in which this block must be stored. The third field is a **Tag field**, used to store higher order 5-bits of the memory address of the block, and to identify which of the 32blocks are mapped into the cache.

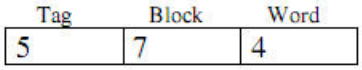


Figure 8:Main Memory Address

It is the simplest mapping technique, in which each block from the main memory has only one possible location in the cache organization. For example, the block I of the main memory maps on to block i module128 of the cache. Therefore, whenever one of the main memory blocks 0, 128, 256, ……. Is loaded in the cache, it is stored in the block 0. Block 1, 129, 257,….. are stored in block 1 of the cache and so on.

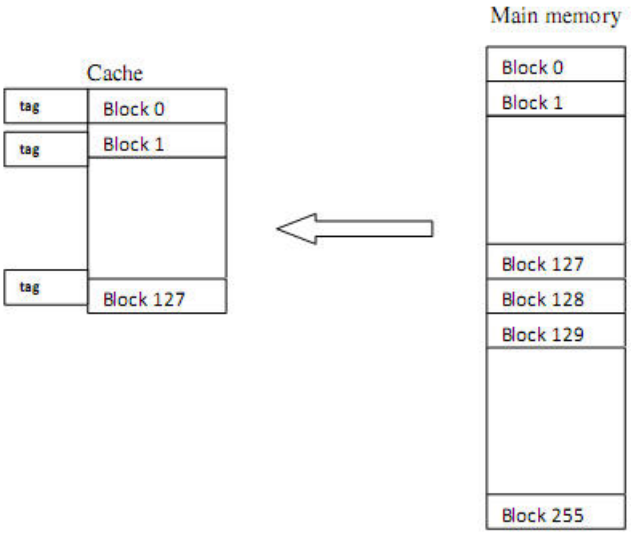


Figure 9 :Direct Mapped Cache

block is present. This is called associative-mapping technique. It gives the complete freedom in choosing the cache location in which to place the memory block.

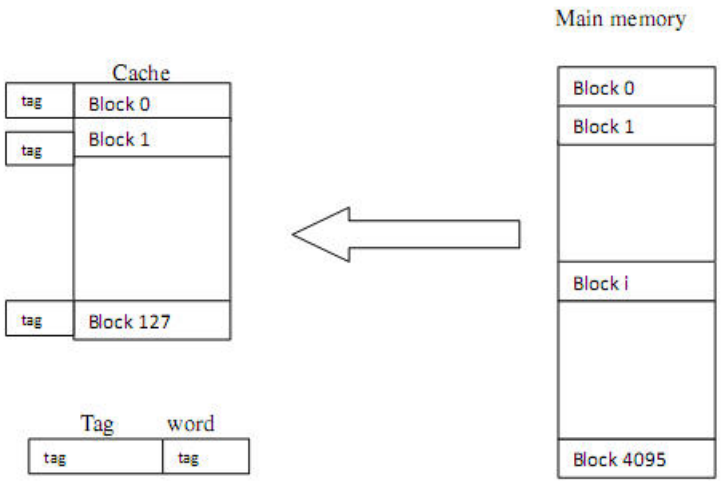


Figure 10: Associative mapped cache

Set-Associative Mapping

It is a combination of the direct and associative-mapping techniques can be used. Blocks of the cache are grouped into sets and the mapping allows a block of main memory to reside in any block of the specific set. In this case memory blocks 0, 64,128……4032 mapped into cache set 0, and they can occupy either of the two block positions within this set. The cache might contain the desired block. The tag field of the address must then be associatively compared to the tags of the two blocks of the set to check if the desired block is present this two associative search is simple to implement.

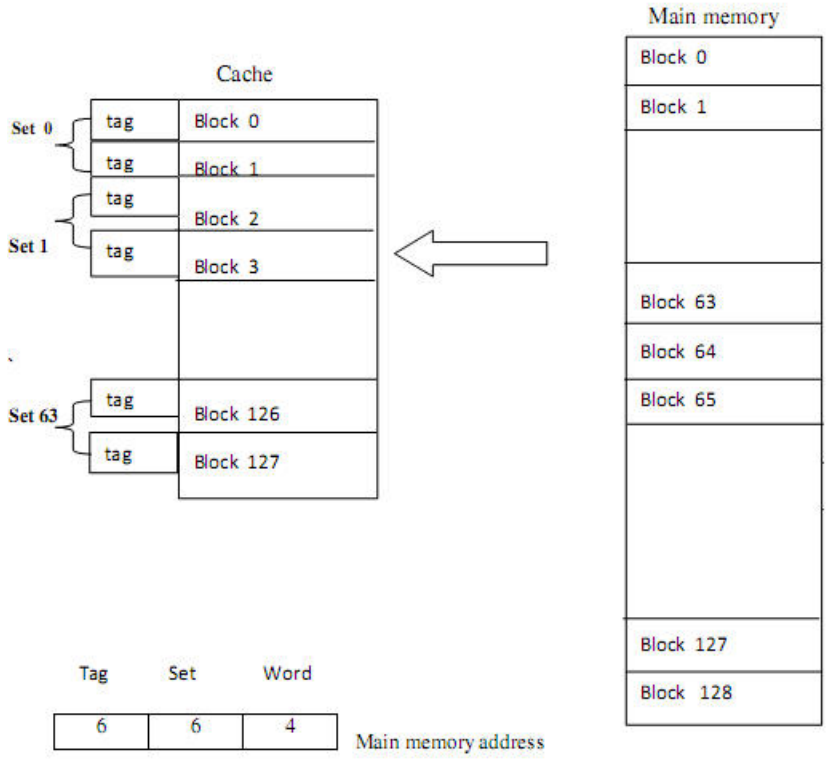


Figure 11: Set-Associative Mapped Cache

Replacement Algorithms

In a direct-mapped cache, the position of each block is fixed, hence no replacement strategy exists. In associative and set-associative caches, when a new block is to be brought into the cache and all the Positions that it may occupy are full, the cache controller must decide which of the old blocks to overwrite. This is important issue because the decision can be factor in system performance.

The objective is to keep blocks in the cache that are likely to be referenced in the near future. Its not easy to determine which blocks are about to be referenced. The property of locality of reference gives a clue to a reasonable strategy. When a block is to be over written, it is sensible to overwrite the one that has gone the longest time without being referenced. This block is called the least recently used(LRU) block, and technique is called the **LRU Replacement algorithm.**

The LRU algorithm has been used extensively for many access patterns, but it can lead to poor performance in some cases. For example, it produces disappointing results when accesses are made to sequential elements of an array that is slightly too large to fit into the cache. Performance of LRU algorithm can be improved by introducing a small amount of randomness in deciding which block to replace.

Example of mapping techniques

We now consider a detailed example to illustrate the effects of different cache mapping techniques. Assume that a processor has separate instruction and data caches. To keep the example simple, assume the data cache has space for only eight blocks of data. Also assume that each block consists of only one 16-bit word of data and the memory is word-addressable with 16-bit addresses. (These parameters are not realistic for actual computers, but they allow us to illustrate mapping techniques clearly.) Finally, assume the LRU replacement algorithm is used for block replacement in the cache.

Let us examine changes in the data cache entries caused by running the following application: A 4 × 10 array of numbers, each occupying one word, is stored in main memory locations 7A00 through 7A27 (hex). The elements of this array, A, are stored in column order, as shown in Figure 5.18. The figure also indicates how tags for different cache mapping techniques are derived from the memory address. Note that no bits are needed to identify a word within a block, as was done in Figures 5.15 through 5.17, because we have assumed that each block contains only one word. The application normalizes the elements of the first row of A with respect to the average value of the elements in the row. Hence, we need to compute the average of the elements in the row

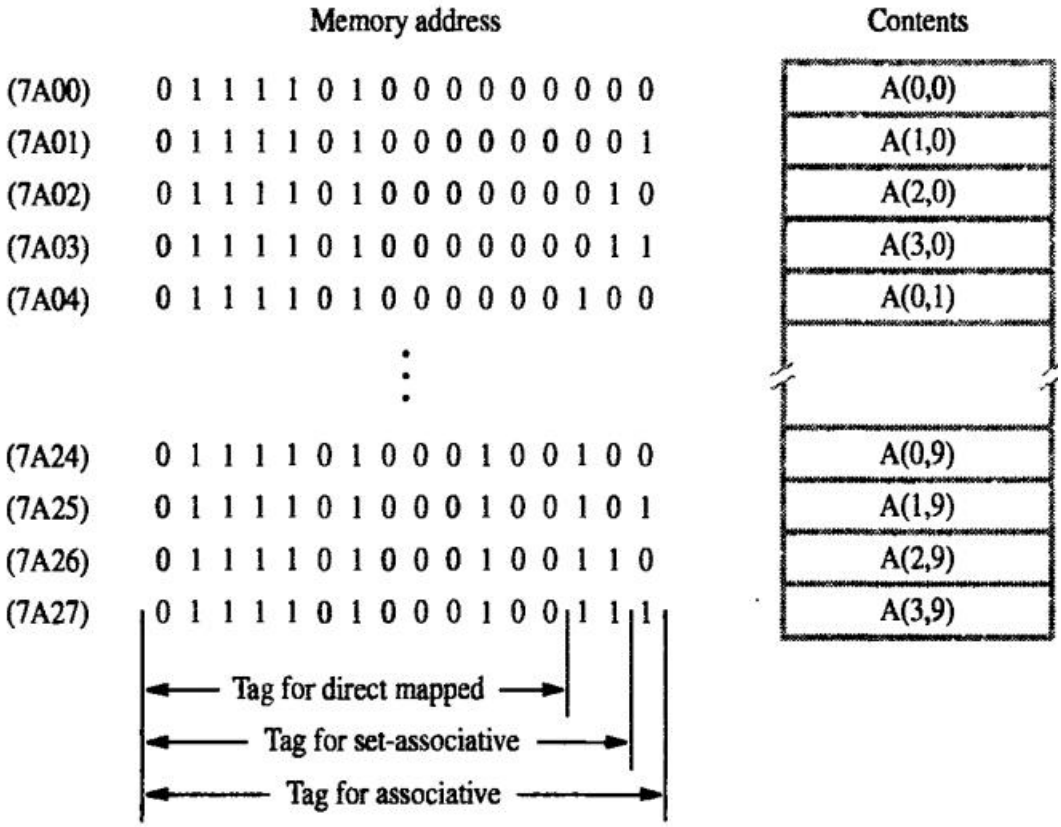


Figure 5.18 An array stored in the main memory.

```
SUM := 0
for j:= 0 to 9 do
        SUM := SUM + A(0,j)
end
AVE := SUM / 10
for i:= 9 downto 0 do
        A(0,i) := A(0,i) / AVE
end
```

Figure 5.19 Task for example in Section 5.5.3.

Block position	Contents of data cache after pass:								
	$j = 1$	$j = 3$	$j = 5$	$j = 7$	$j = 9$	$i = 6$	$i = 4$	$i = 2$	$i = 0$
0	A(0,0)	A(0,2)	A(0,4)	A(0,6)	A(0,8)	A(0,6)	A(0,4)	A(0,2)	A(0,0)
1									
2									
3									
4	A(0,1)	A(0,3)	A(0,5)	A(0,7)	A(0,9)	A(0,7)	A(0,5)	A(0,3)	A(0,1)
5									
6									
7									

Figure 5.20 Contents of a direct-mapped data cache.

and divide each element by that average. The required task can be expressed as

$$A(0,i) \leftarrow \frac{A(0,i)}{\left(\sum_{j=0}^{9} A(0,j)\right)/10} \qquad \text{for } i = 0, 1, \ldots, 9$$

Figure 5.19 gives the structure of a program that corresponds to this task. In a machine language implementation of this program, the array elements will be addressed as memory locations. We use the variables SUM and AVE to hold the sum and average values, respectively. These variables, as well as index variables i and j, will be held in processor registers during the computation.

Direct mapped Cache:

In a direct-mapped data cache, the contents of the cache change as shown in Figure 5.20. The columns in the table indicate the cache contents after various passes through the two program loops in Figure 5.19 are completed. For example, after the second pass through the first loop ($j = 1$), the cache holds the elements $A(0, 0)$ and

$A(0, 1)$. These elements are in block positions 0 and 4, as determined by the three least-significant bits of the address. During the next pass, the $A(0, 0)$ element is replaced by $A(0, 2)$, which maps into the same block position. Note that the desired elements map into only two positions in the cache, thus leaving the contents of the other six positions unchanged from whatever they were before the normalization task was executed.

After the tenth pass through the first loop ($j = 9$), the elements $A(0, 8)$ and $A(0, 9)$ are found in the cache. Since the second loop reverses the order in which the elements are handled, the first two passes through this loop ($i = 9, 8$) will find the required data in the cache. When $i = 7$, the element $A(0, 9)$ is replaced with $A(0, 7)$. When $i = 6$, element $A(0, 8)$ is replaced with $A(0, 6)$, and so on. Thus, eight elements are replaced while the second loop is executed.

Associate mapped cache:

Figure 5.21 presents the changes if the cache is associative-mapped. During the first eight passes through the first loop, the elements are brought into consecutive block positions, assuming that the cache was initially empty. During the ninth pass ($j = 8$), the LRU algorithm chooses $A(0, 0)$ to be overwritten by $A(0, 8)$. The next and last pass through the j loop sees $A(0, 1)$ replaced by $A(0, 9)$. Now, for the first eight passes through the second loop ($i = 9, 8, \ldots, 2$) all required elements are found in the cache. When $i = 1$, the element needed is $A(0, 1)$, so it replaces the least recently used element, $A(0, 9)$. During the last pass, $A(0, 0)$ replaces $A(0, 8)$.

In this case, when the second loop is executed, only two elements are not found in the cache. In the direct-mapped case, eight of the elements had to be reloaded during the second loop. Obviously, the associative-mapped cache benefits from the complete freedom in mapping a memory block into any position in the cache. Good utilization

	Contents of data cache after pass:				
Block position	$j = 7$	$j = 8$	$j = 9$	$i = 1$	$i = 0$
0	A(0,0)	A(0,8)	A(0,8)	A(0,8)	A(0,0)
1	A(0,1)	A(0,1)	A(0,9)	A(0,1)	A(0,1)
2	A(0,2)	A(0,2)	A(0,2)	A(0,2)	A(0,2)
3	A(0,3)	A(0,3)	A(0,3)	A(0,3)	A(0,3)
4	A(0,4)	A(0,4)	A(0,4)	A(0,4)	A(0,4)
5	A(0,5)	A(0,5)	A(0,5)	A(0,5)	A(0,5)
6	A(0,6)	A(0,6)	A(0,6)	A(0,6)	A(0,6)
7	A(0,7)	A(0,7)	A(0,7)	A(0,7)	A(0,7)

Figure 5.21 Contents of an associative-mapped data cache.

	Contents of data cache after pass:					
	$j = 3$	$j = 7$	$j = 9$	$i = 4$	$i = 2$	$i = 0$
Set 0	A(0,0)	A(0,4)	A(0,8)	A(0,4)	A(0,4)	A(0,0)
	A(0,1)	A(0,5)	A(0,9)	A(0,5)	A(0,5)	A(0,1)
	A(0,2)	A(0,6)	A(0,6)	A(0,6)	A(0,2)	A(0,2)
	A(0,3)	A(0,7)	A(0,7)	A(0,7)	A(0,3)	A(0,3)
Set 1						

Figure 5.22 Contents of a set-associative-mapped data cache.

of this cache also occurred because we chose to reverse the order in which the elements are handled in the second loop of the program. It is interesting to consider what would happen if the second loop dealt with the elements in the same order as in the first loop (see Problem 5.12). Using the LRU algorithm, all elements would be overwritten before they are used in the second loop. This degradation in performance would not occur if a random replacement algorithm were used.

Set Associative mapped cache:

For this example, we assume that a set-associative data cache is organized into two sets, each capable of holding four blocks. Thus, the least-significant bit of an address determines which set the corresponding memory block maps into. The high-order 15 bits constitute the tag.

Changes in the cache contents are depicted in Figure 5.22. Since all the desired blocks have even addresses, they map into set 0. Note that, in this case, six elements must be reloaded during execution of the second loop.

Even though this is a simplified example, it illustrates that in general, associative mapping performs best, set-associative mapping is next best, and direct mapping is the worst. However, fully associative mapping is expensive to implement, so set-associative mapping is a good practical compromise.

4.5 Performance Considerations:

 Two Key factors in the commercial success are the performance & cost ie the best possible performance at low cost.

 A common measure of success is called the Price/ Performance ratio. Performance depends on how fast the machine instruction are brought to the processor and how fast they are executed.

 To achieve parallelism(ie. Both the slow and fast units are accessed in the same manner),interleaving is used.

Interleaving:

 If the main memory system is divided into a number of memory modules. Each module has its own address buffer register (ABR) and data buffer register (DBR).

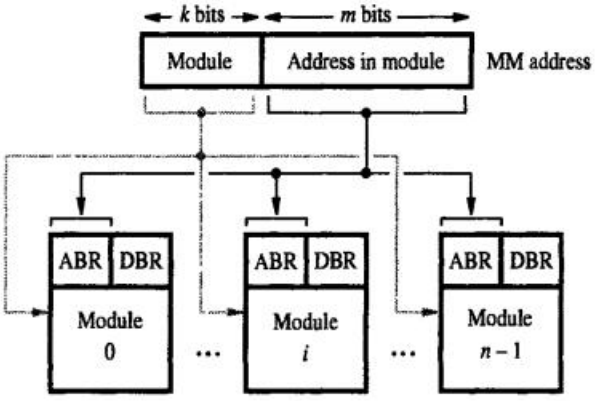
 Memory access operations may proceed in more than one module at the same time. Thus the aggregate rate of transmission of words to and from the main memory system can be increased.

 Two methods of address layout are indicated they are

 Consecutive words in a module

 Consecutive words in a consecutive module

Consecutive words in a module



(a) Consecutive words in a module

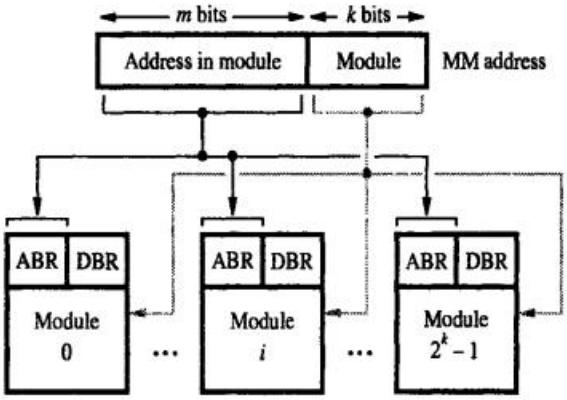
Consecutive words are placed in a module.

 High-order k bits of a memory address determine the module.

 Low-order m bits of a memory address determine the word within a module.

 When a block of words is transferred from main memory to cache, only one module is busy at a time.

Consecutive words in a consecutive module



(b) Consecutive words in consecutive modules

Consecutive words are located in consecutive modules.

 Consecutive addresses can be located in consecutive modules.

 While transferring a block of data, several memory modules can be kept busy at the same time.

 This is called **interleaving**

 When requests for memory access involve consecutive addresses, the access will be to different modules.

 Since parallel access to these modules is possible, the average rate of fetching words from the Main Memory can be increased.

Example:

The effect of interleaving is substantial. Consider the time needed to transfer a block of data from the main memory to the cache when a read miss occurs. Suppose that a cache with 8-word blocks is used, similar to our examples in Section 5.5. On a read miss, the block that contains the desired word must be copied from the memory into the cache. Assume that the hardware has the following properties. It takes one clock cycle to send an address to the main memory. The memory is built with relatively slow DRAM chips that allow the first word to be accessed in 8 cycles, but subsequent words of the block are accessed in 4 clock cycles per word. (Recall from Section 5.2.3 that, when consecutive locations in a DRAM are read from a given row of cells, the row address is decoded only once. Addresses of consecutive columns of the array are then applied to access the desired words, which takes only half the time per access.) Also, one clock cycle is needed to send one word to the cache.

If a single memory module is used, then the time needed to load the desired block into the cache is

$$1 + 8 + (7 \times 4) + 1 = 38 \text{ cycles}$$

Suppose now that the memory is constructed as four interleaved modules, using the scheme in Figure 5.25*b*. When the starting address of the block arrives at the memory, all four modules begin accessing the required data, using the high-order bits of the address. After 8 clock cycles, each module has one word of data in its DBR. These words are transferred to the cache, one word at a time, during the next 4 clock cycles. During this time, the next word in each module is accessed. Then it takes another 4 cycles to transfer these words to the cache. Therefore, the total time needed to load the block from the interleaved memory is

$$1 + 8 + 4 + 4 = 17 \text{ cycles}$$

Thus, interleaving reduces the block transfer time by more than a factor of 2.

Hit Rate and Miss Penalty

An excellent indicator of the effectiveness of a particular implementation of the memory hierarchy is the success rate in accessing information at various level of the hierarchy. A successful access to data in a cache is called a hit.

The number of hits stated as fraction of all attempted access is called the hit rate, and the miss rate is the number of misses stated as a fraction of attempted accesses.

 Hit rate can be improved by increasing block size, while keeping cache size constant.

 Block sizes that are neither very small nor very large give best results.

 Miss penalty can be reduced if load-through approach is used when loading new blocks into cache.

Example:

Consider now the impact of the cache on the overall performance of the computer. Let h be the hit rate, M the miss penalty, that is, the time to access information in the main memory, and C the time to access information in the cache. The average access time experienced by the processor is

$$t_{ave} = hC + (1 - h)M$$

We use the same parameters as in Example 5.1. If the computer has no cache, then, using a fast processor and a typical DRAM main memory, it takes 10 clock cycles for each memory read access. Suppose the computer has a cache that holds 8-word blocks and an interleaved main memory. Then, as we showed in Section 5.6.1, 17 cycles are needed to load a block into the cache. Assume that 30 percent of the instructions in a typical program perform a read or a write operation, which means that there are 130 memory accesses for every 100 instructions executed. Assume that the hit rates in the cache are 0.95 for instructions and 0.9 for data. Let us further assume that the miss penalty is the same for both read and write accesses. Then, a rough estimate of the improvement in performance that results from using the cache can be obtained as follows:

$$\frac{\textit{Time without cache}}{\textit{Time with cache}} = \frac{130 \times 10}{100(0.95 \times 1 + 0.05 \times 17) + 30(0.9 \times 1 + 0.1 \times 17)} = 5.04$$

This result suggests that the computer with the cache performs five times better.

It is also interesting to consider how effective this cache is compared to an ideal cache that has a hit rate of 100 percent (in which case, all memory references take one cycle). Our rough estimate of relative performance for these caches is

$$\frac{100(0.95 \times 1 + 0.05 \times 17) + 30(0.9 \times 1 + 0.1 \times 17)}{130} = 1.98$$

This means that the actual cache provides an environment in which the processor effectively works with a large DRAM-based main memory that appears to be only two times slower than the circuits in the cache.

In this example, we made a simplifying assumption that the same clock is used to access the on-chip cache and the main memory via the system bus. A high-performance processor is likely to operate under the control of a clock that is much faster than the system bus clock, perhaps up to ten times faster. Let us consider the impact of a cache in a system of this type.

Example 2:

Suppose that there is a single cache that is implemented on the processor chip and that the main memory is realized using SDRAM chips. Assume that the system bus clock is four times slower than the processor clock. As in Example 5.2, assume that a cache block contains 8 words, and that the hit rates in the cache are 0.95 for instructions and 0.9 for data. The SDRAM timing diagram is similar to Figure 5.9. The only difference is that there is a burst of 8 data words rather than four. Thus, according to Figure 5.9, it will take 14 clock cycles from when the RAS signal is asserted to transfer a block of data between the main memory and the cache. Since the RAS and CAS signals are generated by the memory controller, as indicated in Figure 5.11, one more cycle is needed during which the processor sends the address of the first word in a block to the memory controller. Therefore, a total of 15 cycles is needed to transfer a block. The cycles shown in Figure 5.9 are the system bus clock cycles. If the processor clock is four times faster, then it takes 60 processor cycles to transfer an 8-word block to or from the main memory. Note also that Figure 5.9 indicates that the processor can read or write a single word in the main memory in 9 bus clock cycles, consisting of the 8 cycles indicated in Figure 5.9 plus one cycle needed to send an address to the memory controller. Hence, 36 processor cycles are needed to access a single word in the main memory. Yet, the processor accesses a word in the cache in one processor cycle!

Repeating the calculation in Example 5.2 gives:

$$\frac{\textit{Time without cache}}{\textit{Time with cache}} = \frac{130 \times 36}{100(0.95 \times 1 + 0.05 \times 60) + 30(0.9 \times 1 + 0.1 \times 60)} = 7.77$$

Thus, accounting for the differences between processor and system bus clock speeds shows that the cache has an even greater positive effect on the performance.

Caches on processor chip:

In high-performance processors two levels of caches are normally used. The L1 cache(s) is on the processor chip. The L2 cache, which is much larger, may be implemented externally using SRAM chips. But, a somewhat smaller L2 cache may also be implemented on the processor chip,

If both L1 and L2 caches are used, the L1 cache should be designed to allow very fast access by the processor because its access time will have a large effect on the clock rate of the processor. A cache cannot be accessed at the same speed as a register file because the cache is much bigger and, hence, more complex. A practical way to speed up access to the cache is to access more than one word simultaneously and then let the processor use them one at a time. This technique is used in many commercial processors.

The L2 cache can be slower, but it should be much larger to ensure a high hit rate. Its speed is less critical because it only affects the miss penalty of the L1 cache. A workstation computer may include an L1 cache with the capacity of tens of kilobytes and an L2 cache of several megabytes.

Including an L2 cache further reduces the impact of the main memory speed on the performance of a computer. The average access time experienced by the processor in a system with two levels of caches is

$$t_{ave} = h_1C_1 + (1 - h_1)h_2C_2 + (1 - h_1)(1 - h_2)M$$

where

h_1 is the hit rate in the L1 cache.

h_2 is the hit rate in the L2 cache.

C_1 is the time to access information in the L1 cache.

C_2 is the time to access information in the L2 cache.

M is the time to access information in the main memory.

The number of misses in the L2 cache, given by the term $(1 - h_1)(1 - h_2)$, should be low. If both h_1 and h_2 are in the 90 percent range, then the number of misses will be less than 1 percent of the processor's memory accesses. Thus, the miss penalty M will be less critical from a performance point of view.

Other enhancements:

Write buffer

 Write-through:

• Each write operation involves writing to the main memory.

• If the processor has to wait for the write operation to be complete, it slows down the processor.

• Processor does not depend on the results of the write operation.

• Write buffer can be included for temporary storage of write requests.

• Processor places each write request into the buffer and continues execution.

• If a subsequent Read request references data which is still in the write buffer, then this data is referenced in the write buffer.

 Write-back:

• Block is written back to the main memory when it is replaced.

• If the processor waits for this write to complete, before reading the new block, it is slowed down.

• Fast write buffer can hold the block to be written, and the new block can be read first.

Prefetching

• New data are brought into the processor when they are first needed.

• Processor has to wait before the data transfer is complete.

• Prefetch the data into the cache before they are actually needed, or a before a Read miss occurs.

• Prefetching can be accomplished through software by including a special instruction in the machine language of the processor.

 Inclusion of prefetch instructions increases the length of the programs.

• Prefetching can also be accomplished using hardware:

 Circuitry that attempts to discover patterns in memory references and then prefetches according to this pattern.

Lockup-Free Cache

• Prefetching scheme does not work if it stops other accesses to the cache until the prefetch is completed.

• A cache of this type is said to be “locked” while it services a miss.

• Cache structure which supports multiple outstanding misses is called a lockup free cache.

• Since only one miss can be serviced at a time, a lockup free cache must include circuits that keep track of all the outstanding misses.

• Special registers may hold the necessary information about these misses.

4.6 VIRTUAL MEMORY:

 Techniques that automatically move program and data blocks into the physical main memory when they are required for execution is called the **Virtual Memory**.

 The binary address that the processor issues either for instruction or data are called the virtual / Logical address.

 The virtual address is translated into physical address by a combination of hardware and software components. This kind of address translation is done by MMU (Memory Management Unit).

When the desired data are in the main memory, these data are fetched /accessed immediately.

 If the data are not in the main memory, the MMU causes the Operating system to bring the data into memory from the disk.

 Transfer of data between disk and main memory is performed using DMA scheme.

Fig: Virtual Memory Organization

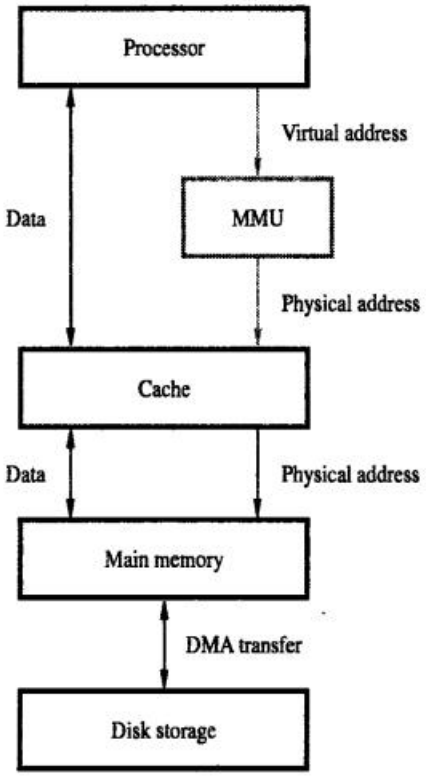


Figure 5.26 Virtual memory organization.

Memory management unit (MMU) translates virtual addresses into physical addresses.

• If the desired data or instructions are in the main memory they are fetched as described previously.

• If the desired data or instructions are not in the main memory, they must be transferred from secondary storage to the main memory.

• MMU causes the operating system to bring the data from the secondary storage into the main memory.

Address Translation:

In address translation, all programs and data are composed of fixed length units called Pages.

The Page consists of a block of words that occupy contiguous locations in the main memory.

The pages are commonly range from 2K to 16K bytes in length. The cache bridge speed up the gap between main memory and secondary storage and it is implemented in software techniques.

Each virtual address generated by the processor contains virtual Page number (Low order bit) and offset(High order bit) Virtual Page number+ Offset Specifies the location of a particular byte (or word) within a page.

Page Table:

It contains the information about the main memory address where the page is stored & the current status of the page.

Page Frame:

An area in the main memory that holds one page is called the page frame.

Page Table Base Register:

 It contains the starting address of the page table.

 Virtual Page Number+Page Table Base register Gives the address of the corresponding entry in the page table.ie)it gives the starting address of the page if that page currently resides in memory.

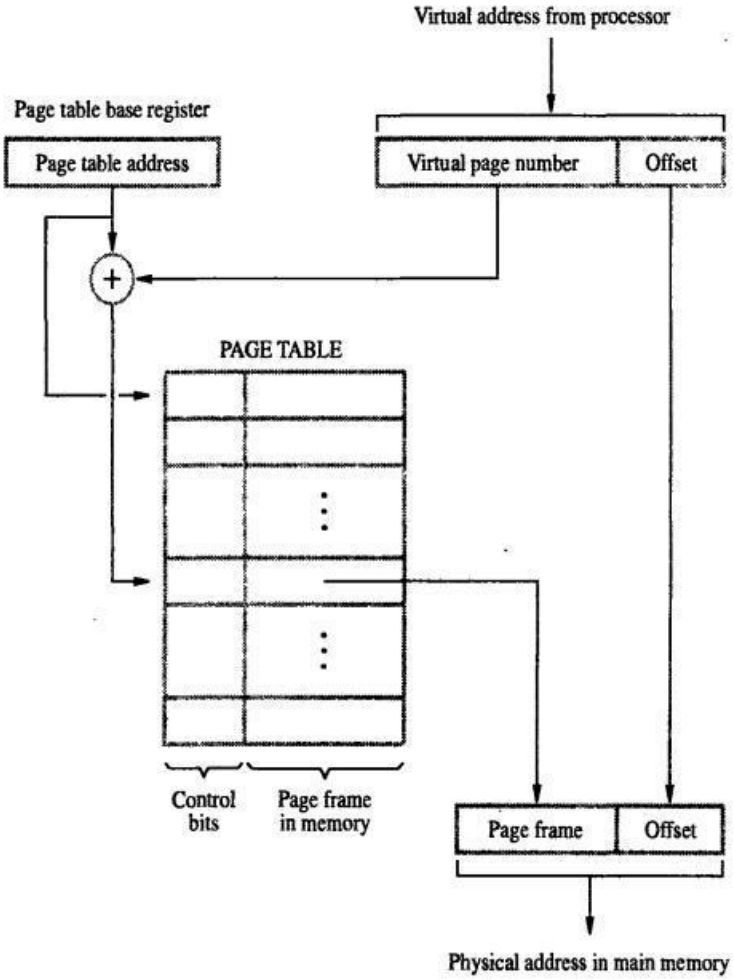
Control Bits in Page Table:

 The Control bit specifies the status of the page while it is in main memory. Function:

 The control bit indicates the validity of the page ie) it checks whether the page is actually loaded in the main memory.

 It also indicates that whether the page has been modified during its residency in the memory; this information is needed to determine whether the page should be written back to the disk before it is removed from the main memory to make room for another page.

Fig: Virtual Memory Address Translation



The Page table information is used by MMU for every read & write access.

 The Page table is placed in the main memory but a copy of the small portion of the page table is located within MMU.

 This small portion or small cache is called Translation Look Aside Buffer (TLB).

 This portion consists of the page table entries that corresponds to the most recently accessed pages and also contains the virtual address of the entry.

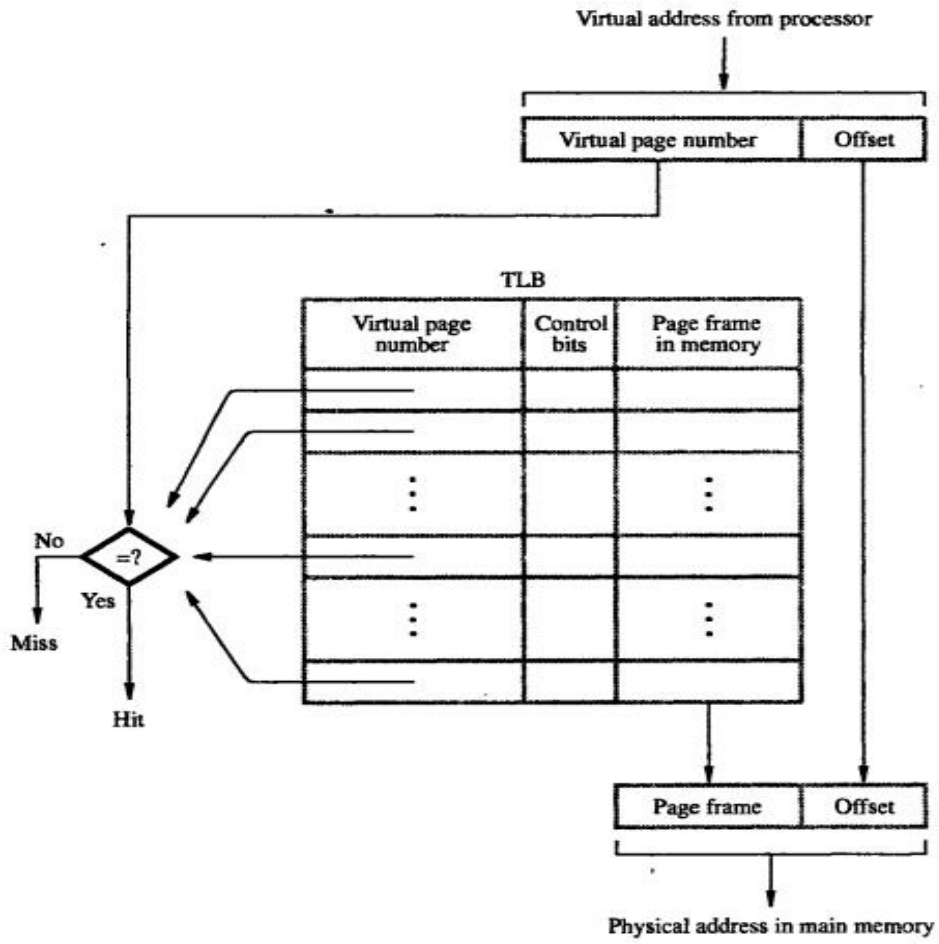


Figure 5.28 Use of an associative-mapped TLB.

When the operating system changes the contents of page table , the control bit in TLB will invalidate the corresponding entry in the TLB. Given a virtual address, the MMU looks in TLB for the referenced page.

 If the page table entry for this page is found in TLB, the physical address is obtained immediately. If there is a miss in TLB, then the required entry is obtained from the page table in the main memory & TLB is updated.

 When a program generates an access request to a page that is not in the main memory, then Page Fault will occur.

The whole page must be brought from disk into memory before an access can proceed. When it detects a page fault, the MMU asks the operating system to generate an interrupt.

 The operating System suspend the execution of the task that caused the page fault and begin execution of another task whose pages are in main memory because the long delay occurs while page transfer takes place.

 When the task resumes,either the interrupted instruction must continue from the point of interruption or the instruction must be restarted.

 If a new page is brought from the disk when the main memory is full,it must replace one of the resident pages.In that case,it uses LRU algorithm which removes the least referenced Page.

 A modified page has to be written back to the disk before it is removed from the main memory. In that case,write – through protocol is used.

MEMORY MANAGEMENT REQUIREMENTS:

Management routines are part of the Operating system. Assembling the OS routine into virtual address sp ace is called “System Space”. The virtual space in which the user application programs reside is called the “User Space”. Each user space has a separate page table. The MMU uses the page table to determine the address of the table to be used in the translation process. Hence by changing the contents of this register, the OS can switch from one space to another. The process has two stages. They are,

 User State

 Supervisor state.

User State: In this state, the processor executes the user program.

Supervisor State: When the processor executes the operating system routines, the processor will be in supervisor state. Privileged Instruction:

In user state, the machine instructions cannot be executed. Hence a user program is prevented from accessing the page table of other user spaces or system spaces.

The control bits in each entry can be set to control the access privileges granted to each program. ie) One program may be allowed to read/write a given page, while the other programs may be given only red access.

4.7 SECONDARY STORAGE:

The Semi-conductor memories do not provide all the storage capability.

The Secondary storage devices provide larger storage requirements. Some of the Secondary Storage devices are,

 Magnetic Disk

 Optical Disk

 Magnetic Tapes.

M.JOSHNA

Magnetic Disk:

 Magnetic Disk system consists o one or more disk mounted on a common spindle.

 A thin magnetic film is deposited on each disk, usually on both sides.

 The disks are placed in a rotary drive so that the magnetized surfaces move in close proximity to read /write heads.

 Each head consists of magnetic yoke & magnetizing coil.

Digital information can be stored on the magnetic film by applying the current pulse of suitable polarity to the magnetizing coil.

 Only changes in the magnetic field under the head can be sensed during the Read operation.

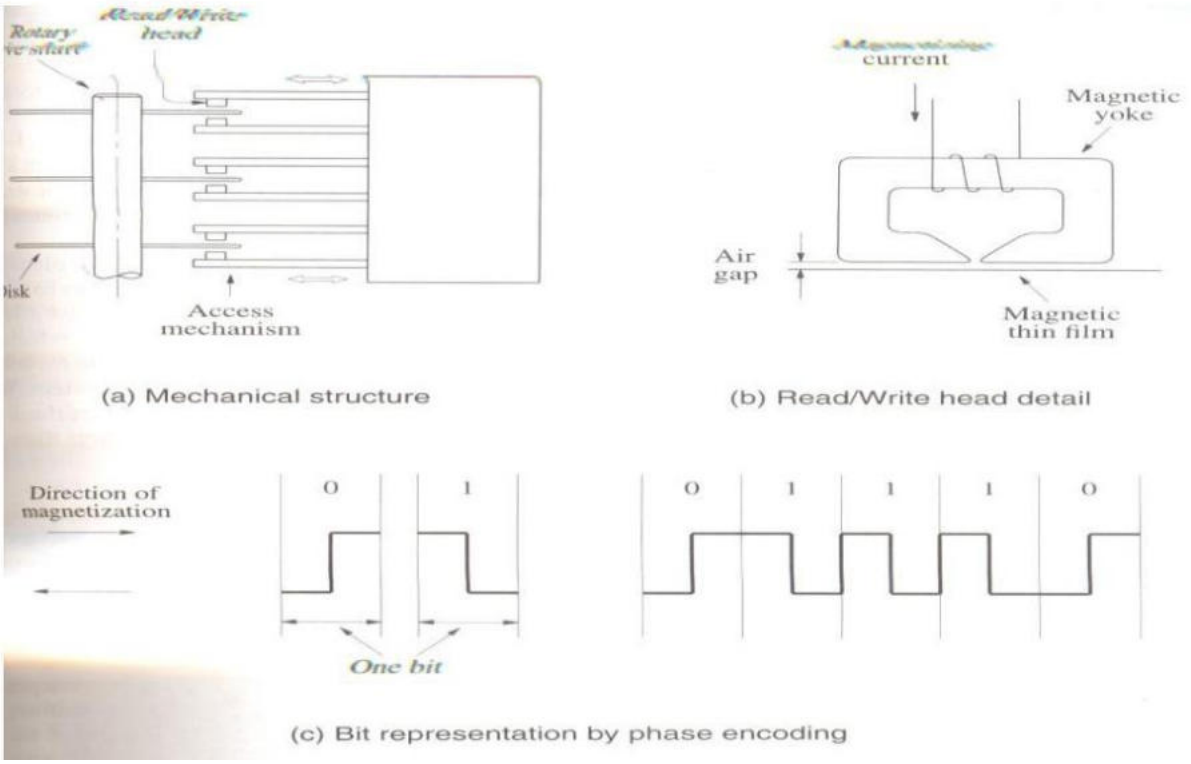
 Therefore if the binary states 0 & 1 are represented by two opposite states of magnetization, a voltage is induced in the head only at 0-1 and at 1-0 transition in the bit stream.

 A consecutive (long string) of 0‟s & 1‟s are determined by using the clock which is mainly used for synchronization.

 Phase Encoding or Manchester Encoding is the technique to combine the clocking information with data.

 The Manchester Encoding describes that how the self-clocking scheme is implemented.

Fig:Mechanical Structure



The Read/Write heads must be maintained at a very small distance from the moving disk surfaces in order to achieve high bit densities.

 When the disks are moving at their steady state, the air pressure develops between the disk surfaces & the head & it forces the head away from the surface.

 The flexible spring connection between head and its arm mounting permits the head to fly at the desired distance away from the surface.

Wanchester Technology:

 Read/Write heads are placed in a sealed, air – filtered enclosure called the Wanchester Technology.

 In such units, the read/write heads can operate closure to magnetic track surfaces because the dust particles which are a problem in unsealed assemblies are absent.

Merits

It have a larger capacity for a given physical size. The data intensity is high because the storage medium is not exposed to contaminating elements.

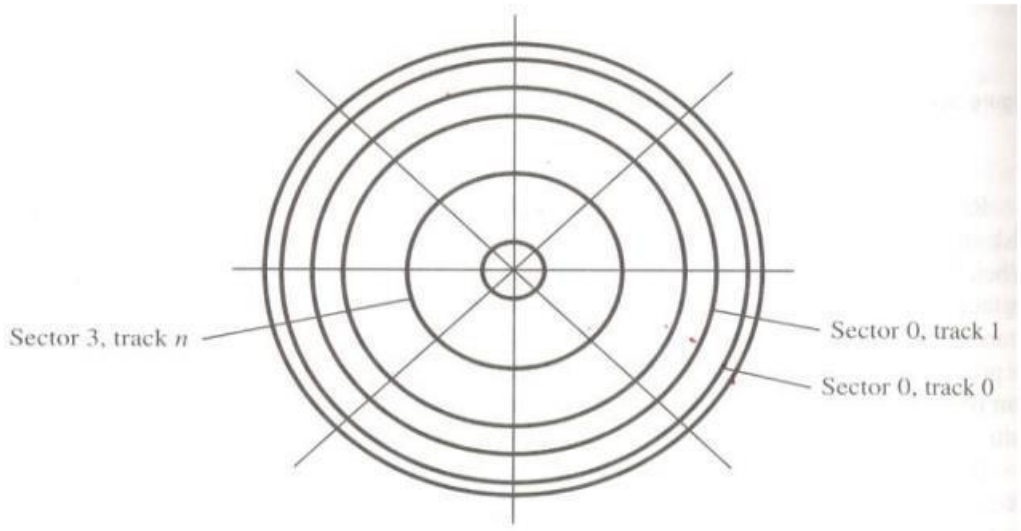
 The read/write heads of a disk system are movable. The disk system has 3 parts.They are,

o Disk Platter(Usually called Disk)

o Disk Drive(spins the disk & moves Read/write heads)

o Disk Controller(controls the operation of the system.)

Fig:Organizing & Accessing the data on disk



Each surface is divided into concentric **tracks.**

Each track is divided into sectors. The set of corresponding tracks on all surfaces of a stack of disk form a logical **cylinder**.

The data are accessed by specifying **the surface number, track number and the**

sector number.

The Read/Write operation start at sector boundaries. Data bits are stored serially on each track.

Each sector usually contains 512 bytes.

Sector header -> contains identification information.

It helps to find the desired sector on the selected track.

ECC (Error checking code)- used to detect and correct errors.

An unformatted disk has no information on its tracks.

The formatting process divides the disk physically into tracks and sectors and this process may discover some defective sectors on all tracks.

The disk controller keeps a record of such defects.

M.JOSHNA

The disk is divided into logical partitions. They are,

 Primary partition

 Secondary partition

In the diag, Each track has same number of sectors.

So all tracks have same storage capacity.

Each surface is divided into concentric **tracks.**

Each track is divided into sectors. The set of corresponding tracks on all surfaces of a stack of disk form a logical **cylinder**.

The data are accessed by specifying **the surface number, track number and the**

sector number.

The Read/Write operation start at sector boundaries. Data bits are stored serially on each track.

Each sector usually contains 512 bytes.

Sector header -> contains identification information.

It helps to find the desired sector on the selected track.

ECC (Error checking code)- used to detect and correct errors.

An unformatted disk has no information on its tracks.

The formatting process divides the disk physically into tracks and sectors and this process may discover some defective sectors on all tracks.

The disk controller keeps a record of such defects.

The disk is divided into logical partitions. They are,

 Primary partition

 Secondary partition

In the diag, Each track has same number of sectors.

So all tracks have same storage capacity.

Each surface is divided into concentric **tracks.**

Each track is divided into sectors. The set of corresponding tracks on all surfaces of a stack of disk form a logical **cylinder**.

The data are accessed by specifying **the surface number, track number and the**

sector number.

The Read/Write operation start at sector boundaries. Data bits are stored serially on each track.

Each sector usually contains 512 bytes.

Sector header -> contains identification information.

It helps to find the desired sector on the selected track.

ECC (Error checking code)- used to detect and correct errors.

An unformatted disk has no information on its tracks.

The formatting process divides the disk physically into tracks and sectors and this process may discover some defective sectors on all tracks.

The disk controller keeps a record of such defects.

The disk is divided into logical partitions. They are,

- Primary partition
- Secondary partition

In the diag, Each track has same number of sectors.

So all tracks have same storage capacity.

Thus the stored information is packed more densely on inner track than on outer track.

Access time

There are 2 components involved in the time delay between receiving an address and the beginning of the actual data transfer. They are,

- Seek time
- Rotational delay / Latency

Seek time – Time required to move the read/write head to the proper track. **Latency** – The amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed sector passes under the read/write head.

M.JOSHNA

Seek time + Latency = Disk access time

Typical disk

One inch disk- weight=1 ounce,

size -> comparable to match book

Capacity -> 1GB

Inch disk has the following parameter

Recording surface=20

Tracks=15000 tracks/surface

Sectors=400.

Each sector stores 512 bytes of data

Capacity of formatted disk=20x15000x400x512=60x109 =60GB

Seek time=3ms

Platter rotation=10000 rev/min

Latency=3ms

Internet transfer rate=34MB/s

Data Buffer / cache

A disk drive that incorporates the required SCSI circuit is referred as SCSI drive. The SCSI can transfer data at higher rate than the disk tracks.

 An efficient method to deal with the possible difference in transfer rate between disk and SCSI bus is accomplished by including a data buffer.

 This buffer is a semiconductor memory.

 The data buffer can also provide cache mechanism for the disk (ie) when a read request arrives at the disk, then controller first check if the data is available in the cache (buffer).

 If the data is available in the cache, it can be accessed and placed on SCSI bus.

 If it is not available then the data will be retrieved from the disk.

Disk Controller:

 The disk controller acts as interface between disk drive and system bus.

 The disk controller uses DMA scheme to transfer data between disk and main memory.

 When the OS initiates the transfer by issuing Read/Write request, the controllers register will load the following information. They are,

Main memory address(address of first main memory location of the block of words involved in the transfer)

 Disk address(The location of the sector containing the beginning of the desired block of words) (number of words in the block to be transferred).

Diskette or Floppy Disk

• spinning platter of special material

• Information stored by magnetically

• read/write head positioned by mechanical arm

• Storage capacity is at a few MBs

• Random access

• seek time from 10 to 40 milliseconds

• Easily portable

Optical Disks

• CD-ROM - read only (books, software releases)

• WORM - write once, read many (archival storage)

• Laser encoding, not magnetic

• 30-50 ms seek times

• 640MB - 17GB storage capacity

• Cheaper than hard disks per MB of storage capacity, but slower

• portable

• Jukeboxes of optical disks are becoming popular for storing really, really large collections of data. The Mercury-20 jukebox (no I'm not selling these, just using it as a typical example) provides access to up to 150 CD-ROMs, or in other words 94GBs of storage capacity. The Mercury jukebox takes a maximum of four seconds to exchange and load a disc into a drive, 2.5 seconds to spin up and access the data and 10 seconds to transfer a 6.0 MB file to the computer or server.

Input/Output Interface:

INPUT–OUTPUT INTERFACES: An interface is a data path between two separate devices in a computer system. Interface to buses can be classified based on the number of bits that are transmitted at a given time to serial versus parallel ports. In a serial port, only 1 bit of data is transferred at a time. Mice and modems are usually connected to serial ports. A parallel port allows more than 1 bit of data to be processed at once. Printers are the most common peripheral devices connected to parallel ports. Table 8.4 shows a summary of the variety of buses and interfaces used in personal computers.

TABLE 8.4 Descriptions of Buses and Interfaces Used in Personal Computers

Bus/Interface	Description
PS/2	A type of port (or interface) that can be used to connect mice and keyboards to the computer. The PS/2 port is sometimes called the mouse port.
Industry standard architecture (ISA)	ISA was originally an 8-bit bus and later expanded to a 16-bit bus in 1984. In 1993, Intel and Microsoft introduced a plug and play ISA bus that allowed the computer to automatically detect and set up computer ISA peripherals such as a modem or sound card.
Extended industry standard architecture (EISA)	EISA is an enhanced form of ISA, which allows for 32-bit data transfers, while maintaining support for 8- and 16-bit expansion boards. However, its bus speed, like ISA, is only 8 MHz. EISA is not widely used, due to its high cost and complicated nature.
Micro channel architecture (MCA)	MCA was introduced by IBM in 1987. It offered several additional features over the ISA such as a 32-bit bus, automatically configured cards and bus mastering for greater efficiency. It is slightly superior to EISA, but not many expansion boards were ever made to fit MCA specifications.
VESA (Video electronics standards association) local bus (VLB)	The VESA, a nonprofit organization founded by NEC, released the VLB in 1992. It is a 32-bit bus that had direct access to the system memory at the speed of the processor, commonly the 486 CPU (33/40 MHz). VLB 2.0 was later released in 1994 and had a 64-bit bus and a bus speed of 50 MHz.
Peripheral component interconnect (PCI)	PCI was introduced by Intel in 1992, revised in 1993 to version 2.0, and later revised in 1995 to PCI 2.1. It is a 32-bit bus that is also available as a 64-bit bus today. Many modern expansion boards are connected to PCI slots.
Advanced graphic port (AGP)	AGP was introduced by Intel in 1997. AGP is a 32-bit bus designed for the high demands of 3D graphics. AGP has a direct line to memory, which allows 3D elements to be stored in the system memory instead of the video memory. AGP is geared towards data-intensive graphics cards, such as 3D accelerators; its design allows for data throughput at rates of 266 MB/s.

TABLE 8.4 *Continued*

Bus/Interface	Description
Universal serial bus (USB)	USB is an external bus developed by Intel, Compaq, DEC, IBM, Microsoft, NEC and Northern Telcom. It was released in 1996 with the Intel 430HX Triton II Mother Board. USB has the capability of transferring 12 Mbps, supporting up to 127 devices. Many devices can be connected to USB ports, which support plug and play.
FireWire (IEEE 1394)	FireWire is a type of external bus, which supports very fast transfer rates: 400 Mbps. Because of this, FireWire is suitable for connecting video devices, such as VCRs, to the computer.
Small computer system interface (SCSI)	SCSI is a type of parallel interface that is commonly used for mass storage devices. SCSI can transfer data at rates of 4 MB/s; in addition, there are several varieties of SCSI that support higher speeds: Fast SCSI (10 MB/s), Ultra SCSI and Fast Wide SCSI (20 MB/s), as well as Ultra Wide SCSI (40 MB/s).
Integrated drive electronics (IDE)	IDE is a commonly used interface for hard disk drives and CD-ROM drives. It is less expensive than SCSI, but offers slightly less in terms of performance.
Enhanced integrated drive electronics (EIDE)	EIDE is an improved version of IDE, which offers better performance than standard SCSI. It offers transfer rates between 4 and 16.6 MB/s.
PCI-X	PCI-X is a high performance bus that is designed to meet the increased I/O demands of technologies such as Fibre Channel, Gigabit Ethernet, and Ultra3 SCSI.
Communication and network riser (CNR)	CNR was introduced by Intel in 2000. It is a specification that supports audio, modem USB and local area networking interfaces of core logic chipsets.

Accessing I/O Devices:

Accessing I/O Devices.: In computing, input/output, or I/O, refers to the communication between an information processing system (computer), and the outside world. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. I/O devices are used by a person (or other system) to communicate with a computer.
Some of the input devices are keyboard, mouse, track ball, joy stick, touch screen, digital camera, webcam, image scanner, fingerprint scanner, barcode reader, microphone and so on. Some of the output devices are speakers, headphones, monitors and printers. Devices for communication between computers, such as modems and network cards, typically serve for both input and output.
I/O devices can be connected to a computer through a single bus which enables the exchange of information. The bus consists of three sets of lines used to carry address, data, and control signals. Each I/O device is assigned a unique set of addresses. When the processor places a particular address on the address lines, the device that recognizes this address responds to the commands issued on the control lines. The processor requests either a read or a write operation, and the

requested data are transferred over the data lines. Figure 5.1 shows the simple arrangement of I/O devices to processor and memory with single bus.

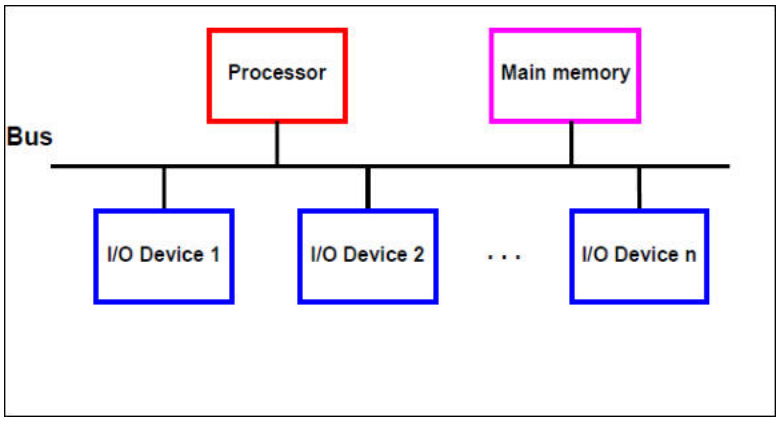


Figure 5.1 A Single bus structure

Memory-mapped I/O: The arrangement of I/O devices and the memory share the same address space is called memory-mapped I/O. With memory-mapped I/O, any machine instruction that can access memory can be used to transfer data to or from an I/O device. For example, if DATAIN is the address of the input buffer associated with the keyboard, the instruction

Move DATAIN,R0

reads the data from DATAIN and stores them into processor register RO. Similarly, the instruction

Move R0,DATAOUT

sends the contents of register R0 to location DATAOUT, which may be the output data buffer of a display unit or a printer. Most computer systems use memory-mapped I/O. Some processors have special In and Out instructions to perform I/O transfers.

Figure 5.2 illustrates the hardware required to connect an I/O device to the bus. The address decoder enables the device to recognize its address when this address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Both the data and status registers are connected to the data bus and assigned unique addresses. The address decoder, the data and status registers, and the control circuitry required to coordinate I/O transfers constitute the device's interface circuit.

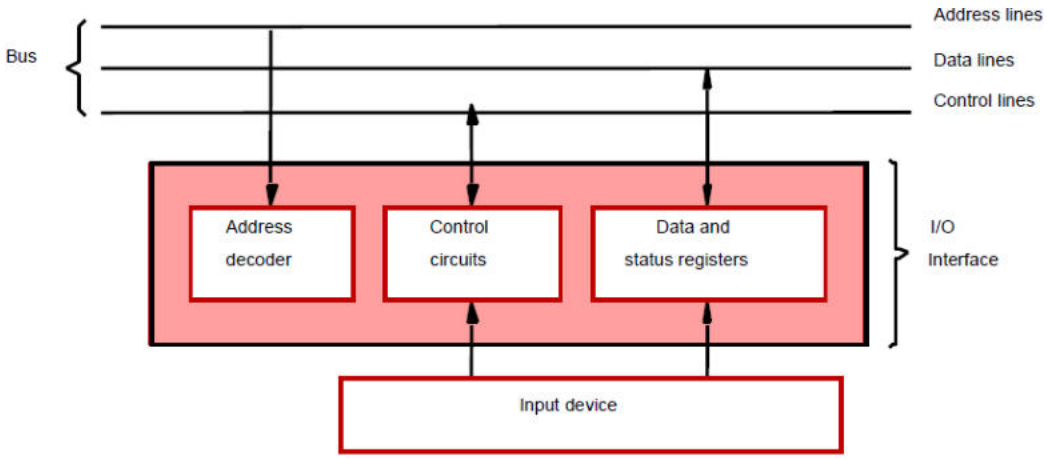


Figure 5.2 I/O interface for an input device

I/O devices operate at speeds that are vastly different from that of the processor. When a human operator is entering characters at a keyboard, the processor is capable of executing millions of instructions between successive character entries. An instruction that reads a character from the keyboard should be executed only when a character is available in the input buffer of the keyboard interface. An input character is read only once.

For an input device such as a keyboard, a status flag, SIN, is included in the interface circuit as part of the status register. This flag is set to 1 when a character is entered at the keyboard and cleared to 0 once this character is read by the processor. Hence, by checking the SIN flag, the software can ensure that it is always reading valid data. This is often accomplished in a program loop that repeatedly reads the status register and checks the state of SIN. When SIN becomes equal to 1, the program reads the input data register. A similar procedure can be used to control output operations using an output status flag, SOUT.

Example:

Program-controlled I/O: Consider a simple example of I/O operations involving a keyboard and a display device in a computer system. The four registers shown in Figure 5.3 are used in the data transfer operations. Register STATUS contains two control flags, SIN and SOUT, which provide status information for the keyboard and the display unit, respectively. The two flags KIRQ and DIRQ in this register are used in conjunction with interrupts. They, and the KEN and DEN bits in register CONTROL, Data from the keyboard are made available in the DATAIN register, and data sent to the display are stored in the DATAOUT register.

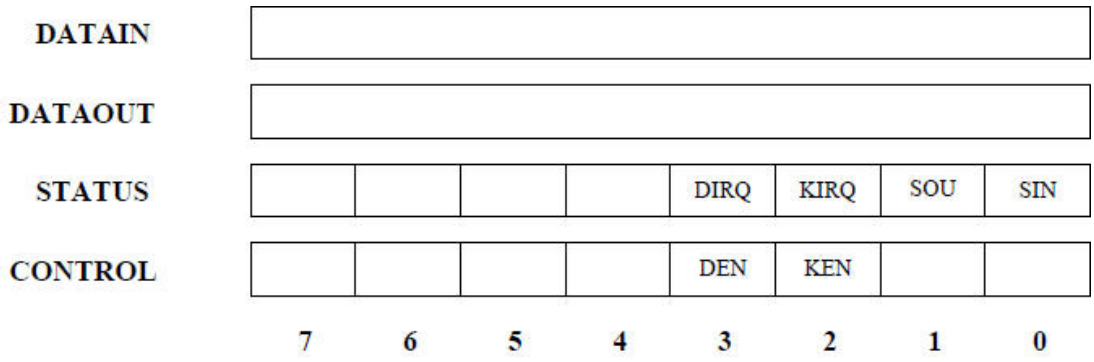


Figure 5.3 Registers in keyboard and display interfaces.

The program in Figure 5.4 reads a line of characters from the keyboard and stores it in a memory buffer starting at location LINE. Then, it calls a subroutine PROCESS to process the input line. As each character is read, it is echoed back to the display. Register R0 is used as a pointer to the memory buffer area. The contents of R0 are updated using the Autoincrement addressing mode so that successive characters are stored in successive memory locations. Each character is checked to see if it is the Carriage Return (CR) character, which has the ASCII code 0D (hex). If it is, a Line Feed character (ASCII code 0A) is sent to move the cursor one line down on the display and subroutine PROCESS is called. Otherwise, the program loops back to wait for another character from the keyboard.

In program-controlled I/O the processor repeatedly checks a status flag to achieve the required synchronization between the processor and an input or output device. The processor polls the device. There are two other commonly used mechanisms for implementing I/O operations: interrupts and direct memory access. In the case of interrupts, synchronization is achieved by having the I/O device send a special signal over the bus whenever it is ready for a data transfer operation. Direct memory access is a technique used for high-speed I/O devices. It involves having the device interface transfer data directly to or from the memory, without continuous involvement by the processor.

```
        Move        #LINE,R0          Initialize memory pointer.
WAITK   TestBit     #0,STATUS         Test SIN.
        Branch=0    WAITK             Wait for character to be entered.
        Move        DATAIN,R1         Read character.
WAITD   TestBit     #1,STATUS         Test SOUT.
        Branch=0    WAITD             Wait for display to become ready.
        Move        R1,DATAOUT        Send character to display.
        Move        R1,(R0)+          Store character and advance pointer.
        Compare     #$0D,R1           Check if Carriage Return.
        Branch!=0   WAITK             If not, get another character.
        Move        #$0A,DATAOUT      Otherwise, send Line Feed.
        Call        PROCESS           Call a subroutine to process the input line.
```

Figure 5.4 A program that reads one line from the keyboard, stores it in memory buffer, and echoes it back to the display.

Interrupts:

Interrupts: Interrupt is a hardware signal to the processor from I/O devices through one of the control line called interrupt-request line. The routine executed in response to an interrupt request is called the interrupt-service routine, Interrupts bear considerable resemblance to subroutine calls. Assume that an interrupt request arrives during execution of instruction i in Figure 4.5. The processor first completes execution of instruction i. Then, it loads the program counter with the address of the first instruction of the interrupt-service routine. For the time being, let us assume that this address is hardwired in the processor. After execution of the interrupt-service routine, the processor has to come back to instruction i 1. Therefore, when an interrupt occurs, the current contents of the PC, which point to instruction i 1, must be put in temporary storage in a known location. A Return from-interrupt instruction at the end of the interrupt-service routine reloads the PC from that temporary storage location, causing execution to resume at instruction i 1. In many processors, the return address is saved on the processor stack. Alternatively, it may be saved in a special location, such as a register provided for this purpose.

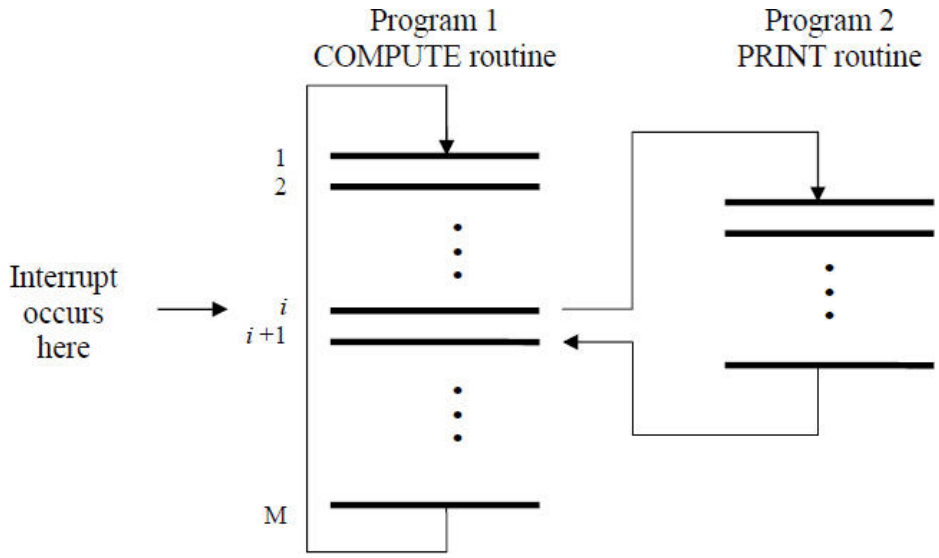
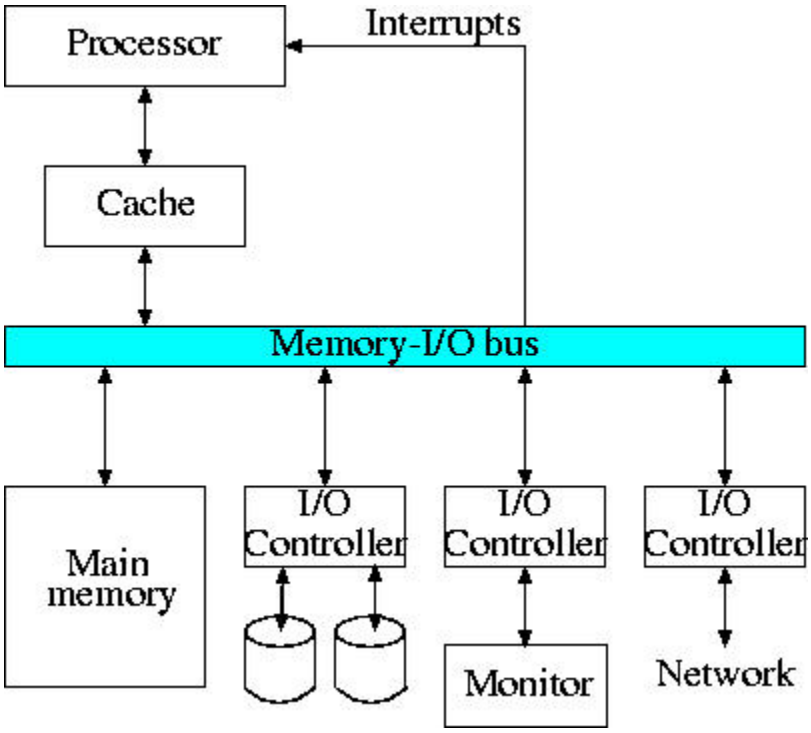


Figure 5.5 Transfer of control through the use of interrupts

The processor must inform the device that its request h been recognized so that it may remove its interrupt-request signal. This may be accomplished by means of a special control signal on the bus called an interrupt-acknowledge signal. The execution of an instruction in the interrupt - service routine that accesses a status or data register in the device interface implicitly informs the device that its interrupt request has been recognized.



Interrupt Hardware:

Interrupt Hardware: In the above discussion, we have assumed that the processor has recognized the occurrence of an interrupt before proceeding to serve it. Computers are provided with interrupt hardware capability in the form of specialized interrupt lines to the processor. These lines are used to send interrupt signals to the processor. In the case of I/O, there exists more than one I/O device. The processor should be provided with a mechanism that enables it to handle simultaneous interrupt requests and to recognize the interrupting device. Two basic schemes can be implemented to achieve this task. The first scheme is called daisy chain bus arbitration (DCBA) and the second is called independent source bus arbitration (ISBA).

According to the DCBA (see Fig. 8.6a), I/O devices present their interrupt requests to the interrupt request line INR (similar to the polling arrangement). Upon recognizing the arrival of an interrupt request, the processor, through a daisy chained grant line (GL), sends its grant to the requesting device to start communication with the processor. The GL goes through all devices starting from the first device nearer to the processor and going to the next device and so on until it reaches the last device (Device #N). If Device #1 has put a request, then it will hold the grant signal and start communication with the processor. If, on the other hand, Device #1 has no interrupt request, it will pass the grant signal to device #2, which will repeat the same procedure, and so on. In the case of multiple requests, the DCBA arrangement gives highest priority to the device physically nearer to the processor. The furthest device from the processor has the lowest priority.

According to the ISBA (see Fig. 8.6b), each I/O device has its own interrupt request line, through which it can send its interrupt request, independent of the other devices. Similarly, each I/O device has its own grant line, through which it receives the grant signal for its request such that it can start communicating with the processor. I/O device priority in the ISBA does not depend on the device location. A priority arbitration circuitry is needed in order to deal with simultaneous interrupt requests. 8.3.2.

Interrupt in Operating Systems When an interrupt occurs, the operating system gains control. The operating system saves the state of the interrupted process, analyzes the interrupt, and passes control to the appropriate routine to handle the interrupt. There are several

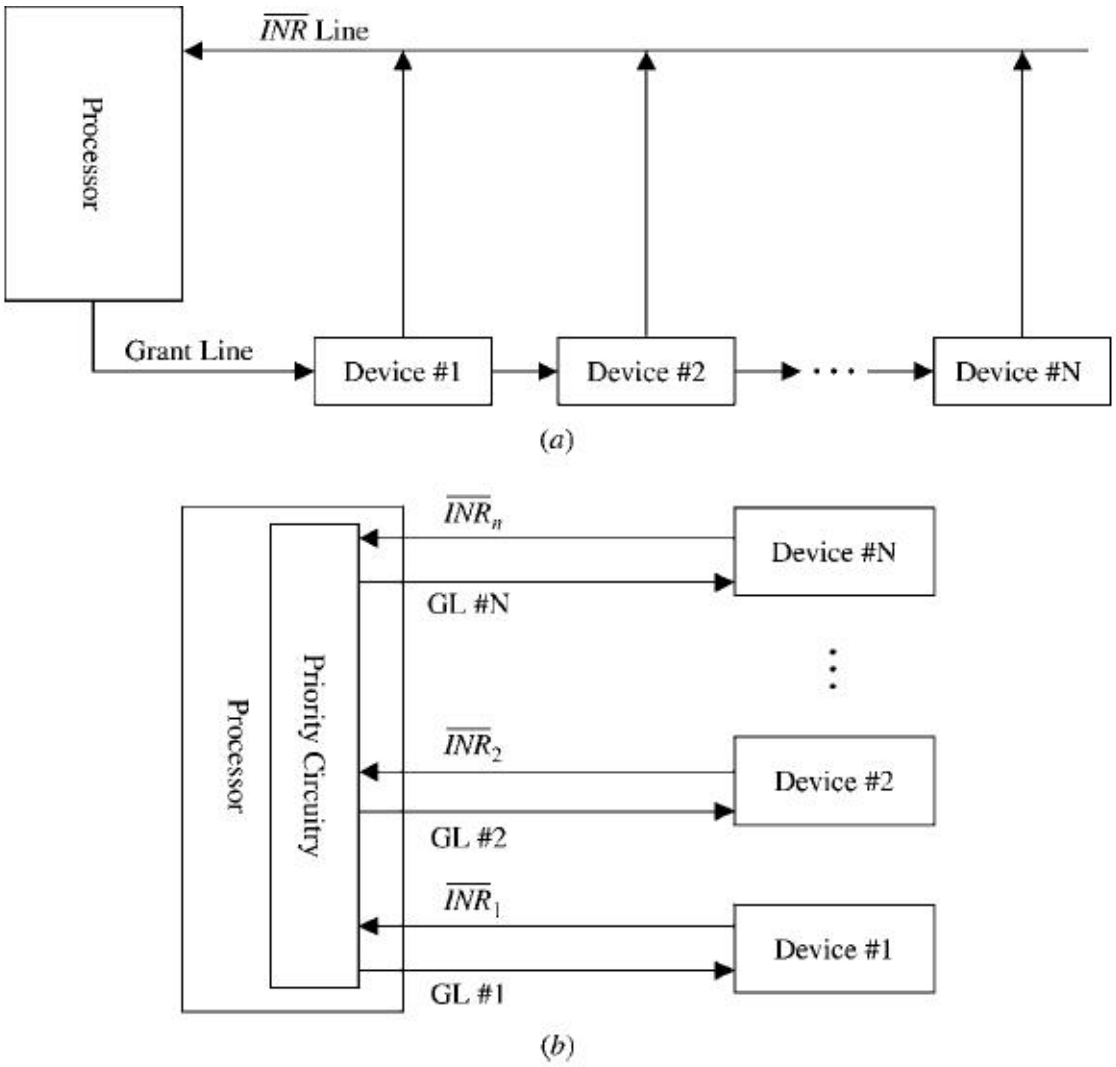


Figure 8.6 Interrupt hardware schemes. (*a*) Daisy chain interrupt arrangement (*b*) Independent interrupt arrangement

types of interrupts, including I/O interrupts. An I/O interrupt notifies the operating system that an I/O device has completed or suspended its operation and needs some service from the CPU. To

process an interrupt, the context of the current process must be saved and the interrupt handling routine must be invoked. This process is called context switching. A process context has two parts: processor context and memory context. The processor context is the state of the CPU's registers including program counter (PC), program status words (PSWs), and other registers. The memory context is the state of the program's memory including the program and data. The interrupt handler is a routine that processes each different type of interrupt.

The operating system must provide programs with save area for their contexts. It also must provide an organized way for allocating and deallocating memory for the interrupted process.When the interrupt handling routine finishes processing the interrupt, the CPU is dispatched to either the interrupted process, or to the highest priority ready process. This will depend on whether the interrupted process is preemptive or nonpreemptive. If the process is nonpreemptive, it gets the CPU again. First the context must be restored, then control is returned to the interrupts process.

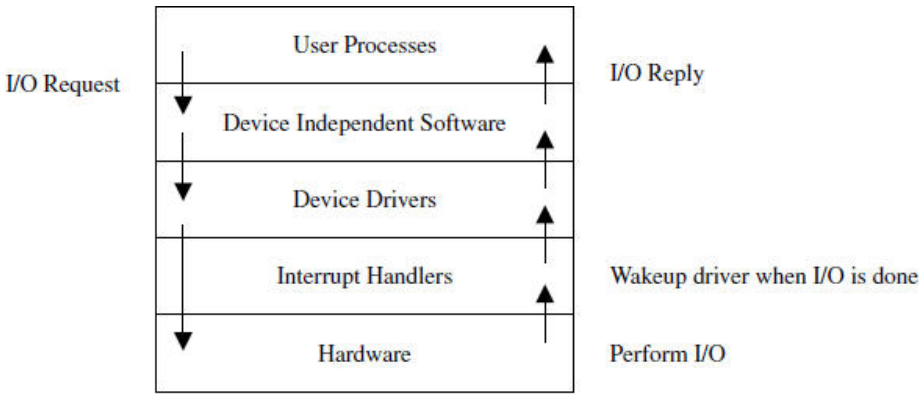


Figure 8.7 Layered I/O software

Figure 8.7 shows the layers of software involved in I/O operations. First, the program issues an I/O request via an I/O call. The request is passed through to the I/O device.When the device completes the I/O, an interrupt is sent and the interrupt handler is invoked. Eventually, control is relinquished back to the process that initiated the I/O.

Examples of interrupt I/O

Example 1: 80386 Interrupt Architecture The 8086 processors have just two hardware interrupt pins. These are labeled INTR andNMI.NMI is a nonmaskable interrupt, which means it cannot be blocked and the processor must respond to it. The NMI input is usually reserved for critical system functions. The INTR input is a maskable interrupt request line between the CPU and the programmable interrupt controller (8259A PIC). Interrupts on INTR can be enabled and disabled using the instructions STI (set interrupt flag) and CLI (clear interrupt flag), respectively. Interrupt handlers are called interrupt service routines (ISR). The address of each interrupt service routine is stored in four consecutive memory locations in the interrupt vector table (IVT). The IVT stores pointers to ISR for each type of interrupt. When an interrupt occurs, an 8-bit type number is supplied to the processor, which identifies the appropriate entry in this table.

When an interrupt is generated by a device, it goes to the PIC. Multiple interrupts may be generated simultaneously. However, they are all buffered by the PIC. The PIC decides which one of these interrupts should be forwarded to the CPU. To inform the CPU that an outstanding interrupt is waiting to be processed, the PIC sends an interrupt request (INTR) to the CPU, which then, at the appropriate time, responds with an interrupt acknowledgment (INTA). At this time, PIC will put an 8-bit interrupt type number associated with the device on the bus so that the CPU can identify which interrupt handler to invoke. In the case when several interrupts are pending, PIC will send next interrupt request to the CPU only after it receives an end of interrupt command from the current ISR. Figure 8.8 shows the simple protocol that is used to determine which ISR is to be invoked. In the computer designs that used a single PIC (PC and XT), eight different interrupt requests are allowed (IRQ0–IRQ7). Table 8.1 shows a list of standard interrupt type numbers for typical devices. When AT was designed, a second PIC was added,

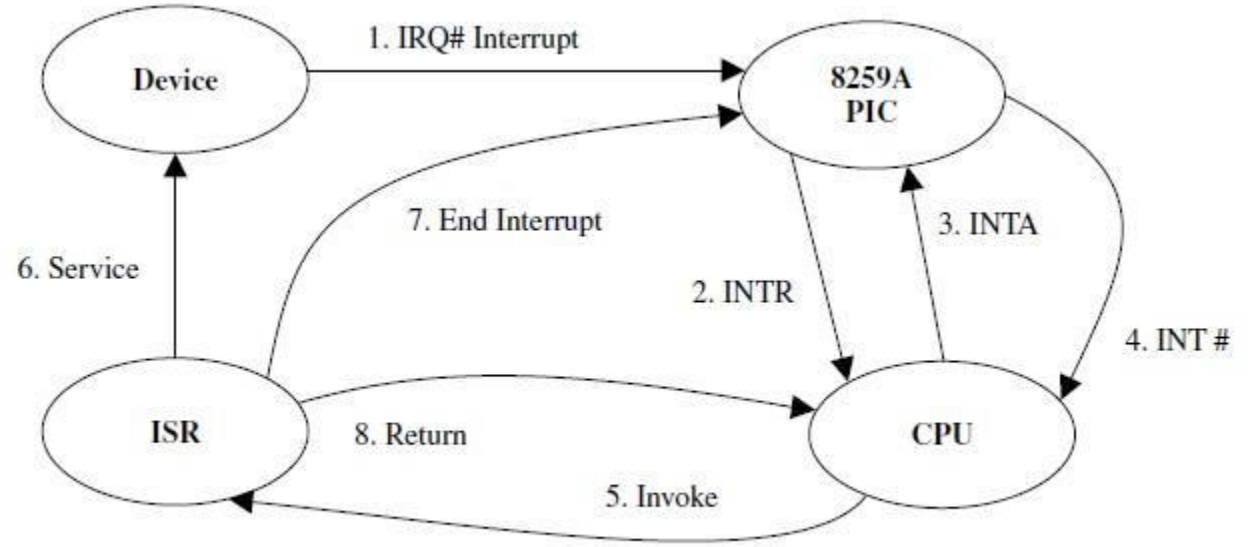


Figure 8.8 Interrupt handling in 80×86

increasing the number of interrupt inputs to 15. Figure 8.9 shows two PICS wired in cascade. One PIC is designated as master and the other becomes the slave. As shown in the figure, all slave interrupts are input via IRQ1 of the master. In general, eight different slaves can be accommodated by a single PIC.

Example 2: ARM Interrupt Architecture ARM stands for Advanced RISC Machines. ARM is a 16/32-bit architecture that is used for portable devices because of its low power consumption and reasonable performance. Interrupt requests to the ARM core are collected and controlled by the interrupt controller, which is called ATIC. The interrupt controller provides an interface to the core and can collect up to 64 interrupt requests.

The usual sequence of events for interrupts is as follows. Interrupts would be enabled at the source (such as a peripheral), then enabled in the interrupt controller, and finally, enabled to the core. When an interrupt occurs at the source, its signal is routed to the interrupt controller then to the ARM core. In the interrupt controller, the interrupt can be enabled or disabled to the core and can be assigned a priority

TABLE 8.1 Standard IBM-PC Interrupt Type Numbers for Typical Devices

Device	IRQ no.	Interrupt type number
Programmable interval timer	0	08H
Keyboard	1	09H
Cascading to the second PICs	2	Reserved
Serial communication port (COM2)	3	0BH
Serial communication port (COM1)	4	0CH
Fixed disk controller	5	0DH
Floppy disk controller	6	0EH
Parallel printer controller	7	0FH

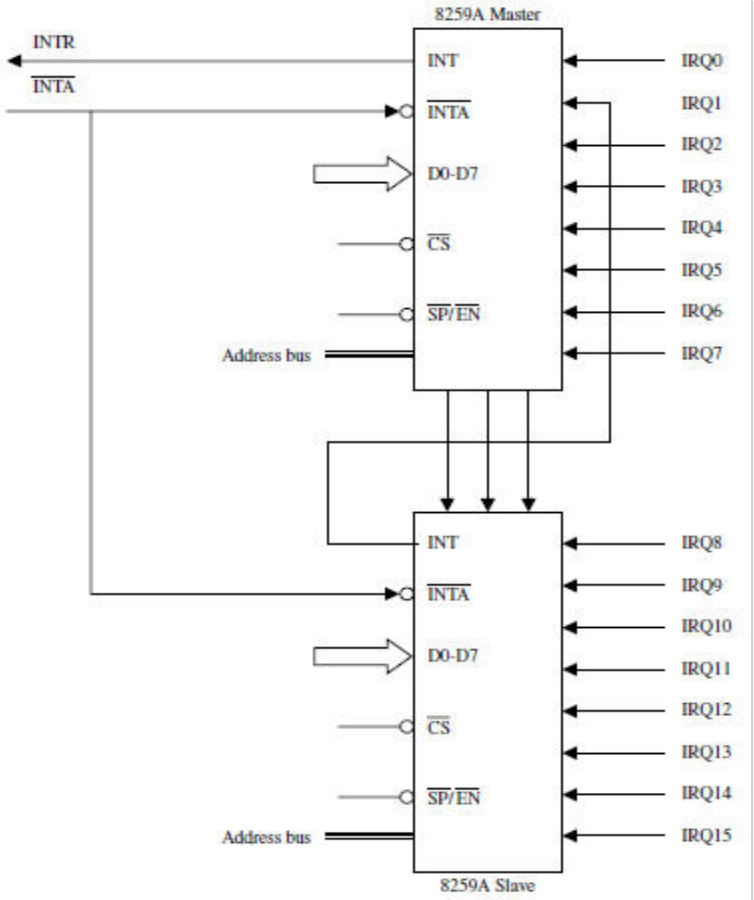


Figure 8.9 Fifteen different interrupts are supported by two PICs wired in cascade

level. Once the interrupt request reaches the core, it will halt the core from its normal processing routines to allow the interrupt request to be serviced. Among the different interrupt requests that the ARM core can handle are IRQ and FIQ requests. The IRQ (normal interrupt request) is used for general-purpose interrupt handling. It has a lower priority than an FIQ (fast interrupt request) and is masked out when an FIQ sequence is entered. The FIQ is used to support highspeed data transfer or channel processes.

TABLE 8.2 Interrupt Vector Table

Exception type	Mode	Address
Reset	Supervisor	0×00000000
Undefined instructions	Undefined	0×00000004
Software interrupts (SWI)	Supervisor	0×00000008
Prefetch abort	Abort	0×0000000C
Data abort	Abort	0×00000010
IRQ (Normal interrupt)	IRQ	0×00000018
FIQ (Fast interrupt)	FIQ	0×0000001C

Similar to the 8086, the addresses of the interrupt handlers are stored in a vector table, which is shown in Table 8.2. For example, when an IRQ is detected by the core, it accesses address 018 of the vector table and executes the instruction loaded in that address. Normally, the instruction found at 018 of the vector table is of the form: LDR PC, IRQ_Handler (load the address of the IRQ interrupt handler in the PC). When an FIQ is detected by the core, it accesses address 01C of the vector table and executes the instruction loaded in that address. Normally, the instruction found at 01C of the vector table is of the form: LDR PC, FIQ_Handler.

Enabling and Disabling Interrupts:

ENABLING AND DISABLING INTERRUPTS: There are many situations in which the processor should ignore interrupt requests. For example, in the case of the Compute-Print program of Figure 5.5, an interrupt request from the printer should be accepted only if there are output lines to be printed. After printing the last line of a set of n lines, interrupts should be disabled until another set becomes available for printing.

A simple way is to provide machine instructions, such as Interrupt-enable and Interruptdisable. The processor hardware ignores the interrupt-request line until the execution of the first instruction of the interrupt-service routine has been completed. Then, by using an Interrupt-disable instruction as the first instruction in the interrupt-service routine, the programmer can ensure that no further interruptions will occur until an Interrupt-enable instruction is executed. Typically, the Interrupt-enable instruction will be the last instruction in the interrupt-service routine before the Return-from-interrupt instruction. The processor must guarantee that execution of the Returnfrom-interrupt instruction is completed before ¬further interruption can occur.

The second option, which is suitable for a simple processor with only one interrupt- request line, is to have the processor automatically disable interrupts before starting the execution of the interrupt-service routine. After saving the contents of the PC and the processor status register (PS) on the stack, the processor performs the equivalent of executing an Interrupt -disable instruction. It is often the case that one bit in the PS register, called Interrupt-enable, indicates whether

interrupts are enabled. An interrupt request received while this bit is equal to 1 will be accepted. After saving the contents of the PS on the stack, with the Interrupt-enable bit equal to 1, the processor clears the Interrupt-enable bit in its PS register, thus disabling further interrupts. When a Return-frominterrupt instruction is executed, the contents of the PS are restored from the stack, setting the Interrupt-enable bit back to 1. Hence, interrupts are again enabled.

In the third option, the processor has a special interrupt-request line for which the interrupthandling circuit responds only to the leading edge of the signal. Such a line is said to be edgetriggered. In this case, the processor will receive only one request, regardless of how long the line is activated. Hence, there is no danger of multiple interruptions and no need to explicitly disable interrupt requests from this line. Before proceeding to study more complex aspects of interrupts, let us summarize the sequence of events involved in handling an interrupt request from a single device.

Assuming that interrupts are enabled, the following is a typical scenario:

1. The device raises an interrupt request.
2. The processor interrupts the program currently being executed.
3. Interrupts are disabled by changing the control bits in the PS (except in the case of edge triggered interrupts).
4. The device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.
5. The action requested by the interrupt is performed by the interrupt-service routine.
6. Interrupts are enabled and execution of the interrupted program is resumed.

Handling Multiple Devices:

HANDLING MULTIPLE DEVICES: Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Because these devices are operationally independent, there is no definite order in which they will generate interrupts. For example, device X may request an interrupt while an interrupt caused by device Y is being serviced, or several devices may request interrupts at exactly the same time. This gives rise to a number of questions:

1. How can the processor recognize the device requesting an interrupt?
2. Given that different devices are likely to require different interrupt -service routines, how can the processor obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled? If two devices have activated the line at the same time, it must be possible to break the tie and elect one of the two requests for service. When the interrupt-service routine for the selected device has been completed, the second request can be serviced.

Vectored interrupts: To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine. The term vectored interrupts refers to all interrupt - handling schemes based on this approach.

A device requesting an interrupt can identify itself by sending a special code to the processor over the bus. This enables the processor to identify individual devices even if they share a single interrupt-request line. The code supplied by the device may represent the starting address of the interrupt-service routine for that device. The code length is typically in the range of 4 to 8 bits. The remainder of the address is supplied by the processor based on the area in its memory where the addresses for interrupt-service routines are located.

This arrangement implies that the interrupt-service routine for a given device must always start at the same location. The programmer can gain some flexibility by storing in this location an instruction that causes a branch to the appropriate routine. In many computers, this is done automatically by the interrupt-handling mechanism. The location pointed to by the interrupting device is used to store the starting address of the interrupt-service routine. The processor reads this address, called the interrupt vector, and loads it into the PC. The interrupt vector may also include a new value for the processor status register.

In most computers, I/O devices send the interrupt-vector code over the data bus, using the bus control signals to ensure that devices do not interfere with each other. When a device sends an interrupt request, the processor may not be ready to receive the interrupt-vector code immediately. For example, it must first complete the execution of the current instruction, which may require the use of the bus. There may be further delays if interrupts happen to be disabled at the time the request is raised. The interrupting device must wait to put data on the bus only when the processor is ready to receive it. When the processor is ready to receive the interrupt-vector code, it activates the interrupt-acknowledge line, INTA. The I/O device responds by sending its interrupt- vector code and turning off the INTR signal.

Interrupt nesting: I/O devices should be organized in a priority structure. An interrupt request from a highpriority device should be accepted while the processor is servicing another request from a lowerpriority device. A multiple-level priority organization means that during execution of an interruptservice routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. To implement this scheme, priority level can be assigned to the processor that can be changed by the program.
The processor's priority is usually encoded in a few bits of the processor status word. It can be changed by program instructions that write into the PS. These are privileged instructions, which can be executed only while the processor is running in the supervisor mode. The processor is in the supervisor mode only when executing operating system routines. It switches to the user mode before beginning to execute application programs. Thus, a user program cannot accidentally, or intentionally, change the priority of the processor and disrupt the system's operation. An attempt to execute a privileged instruction while in the user mode leads to a special type of interrupt called a privilege exception.

A multiple-priority scheme can be implemented easily by using separate interrupt- request and interrupt-acknowledge lines for each device, as shown in Figure 5.7. Each of the interruptrequest lines is assigned a different priority level. Interrupt requests received over these lines are sent to a

priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

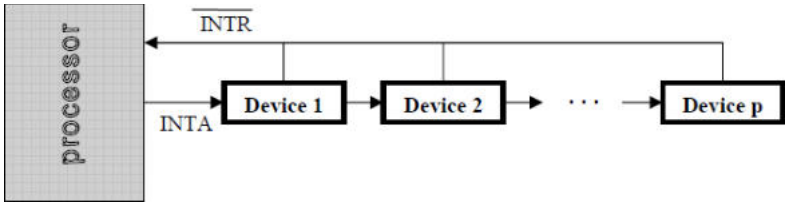


Figure 5.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

Simultaneous request: When simultaneous interrupt requests are arrived from two or more I/O devices to the processor, the processor must have some means of deciding which request to service first.

Using a priority scheme such as that of Figure 5.7, the solution is straightforward. The processor simply accepts the request having the highest priority. If several devices share one interrupt-request line, as in Figure 5.6, some other mechanism is needed. Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. A widely used scheme is to connect the devices to form a daisy chain, as shown in Figure 5.8. The interruptrequest line INTR is common to all devices. The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices. When several devices raise an interrupt request and the INTR line is activated, the processor responds by setting the INTA line to 1. This signal is received by device 1. Device 1 passes the signal on to device 2 only if it does not require any service. If device 1 has a pending request for interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines. Therefore, in the daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority. The second device along the chain has second highest priority, and so on.

Controlling Device Requests:

CONTROLLING DEVICE REQUESTS: The control needed is usually provided in the form of an interrupt-enable bit in the device's interface circuit. The keyboard interrupt-enable, KEN, and display interrupt- enable, DEN, flags in register CONTROL in Figure 5.3 perform this function. If either of these flags is set, the interface circuit generates an interrupt request whenever the corresponding status flag in register STATUS is set. At the same time, the interface circuit sets bit KIRQ or DIRQ to indicate that the keyboard or display unit, respectively, is requesting an interrupt. If an interrupt-enable bit is equal to 0, the interface circuit will not generate an interrupt request, regardless of the state of the status flag.
There are two independent mechanisms for controlling interrupt requests. At the device end, an interrupt-enable bit in a control register determines whether the device is allowed to generate an

interrupt request. At the processor end, either an interrupt enable bit in the PS register or a priority structure determines whether a given interrupt request will be accepted.

EXCEPTIONS: The term exception is often used to refer to any event that causes an interruption. Hence, I/O interrupts are one example of an exception.

Recovery from errors: Computers use a variety of techniques to ensure that all hardware components are operating properly. For example, many computers include an error-checking code in the main memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it and informs the processor by raising an interrupt. The processor may also interrupt a program if it detects an error or an unusual condition while executing the instructions of this program. For example, the OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero. When exception processing is initiated as a result of such errors, the processor proceeds in exactly the same manner as in the case of an I/O interrupt request. It suspends the program being executed and starts an exception-service routine. This routine takes appropriate action to recover from the error, if possible, or to inform the user about it. Recall that in the case of an I/O interrupt, the processor completes execution of the instruction in progress before accepting the interrupt. However, when an interrupt is caused by an error, execution of the interrupted instruction cannot usually be completed, and the processor begins exception processing immediately.

Debugging: Another important type of exception is used as an aid in debugging programs. System software usually includes a program called a debugger, which helps the programmer find errors in a program. The debugger uses exceptions to provide two important facilities called trace and breakpoints.
When a processor is operating in the trace mode, an exception occurs after execution of every instruction, using the debugging program as the exception-service routine. The debugging program enables the user to examine the contents of registers, memory locations, and so on. On return from the debugging program, the next instruction in the program being debugged is executed, then the debugging program is activated again. The trace exception is disabled during the execution of the debugging program. Breakpoints provide a similar facility, except that the program being debugged is interrupted only at specific points selected by the user. An instruction called Trap or Software interrupt is usually provided for this purpose. Execution of this instruction results in exactly the same actions as when a hardware interrupt request is received. While debugging a program, the user may wish to interrupt program execution after instruction i. The debugging routine saves instruction i 1 and replaces it with a software interrupt instruction. When the program is executed and reaches that point, it is interrupted and the debugging routine is activated. This gives the user a chance to examine memory and register contents. When the user is ready to continue executing the program being debugged, the debugging routine restores the saved instruction that was at location i 1 and executes a Return-from-interrupt instruction.

Privilege exception: To protect the operating system of a computer from being corrupted by user programs, certain instructions can be executed only while the processor is in the supervisor mode. These are called privileged instructions. For example, when the processor is running in the user mode, it will not execute an instruction that changes the priority level of the processor or that enables a user program to access areas in the computer memory that have been allocated to other

users. An attempt to execute such an instruction will produce a privilege exception, causing the processor to switch to the supervisor mode and begin executing an appropriate routine in the operating system.

Use of Interrupts in Operating Systems:

USE OF INTERRUPTS IN OPERATING SYSTEMS: The operating system (OS) is responsible for coordinating all activities within a computer. It makes extensive use of interrupts to perform I/O operations and communicate with and control the execution of user programs. The operating system incorporates the interrupt-service routines (ISR) for all devices connected to a computer. Application programs do not perform I/O operations themselves. An operating system provides a variety of services to application programs. To facilitate the implementation of these services, most processors have several different software interrupt instructions, each with its own interrupt vector. They can be used to call different parts of the OS.

- In a computer that has both a supervisor and a user mode, the processor switches its operation to supervisor mode at the time it accepts an interrupt request. It does so by setting a bit in the processor status register after saving the old contents of that register on the stack. Thus, when an application program calls the as by a software interrupt instruction, the processor automatically switches to supervisor mode, giving the as complete access to the computer's resources. When the as executes a Return-from-interrupt instruction, the processor status word belonging to the application program is restored from the stack. As a result, the processor switches back to the user mode.
- Multitasking is a mode of operation in which a processor executes several user programs at the same time. A common as technique that makes this possible is called time slicing. when operating system is started, an initialization routine OSINIT called for initialization. OSINIT loads the starting address of a routine called SCHEDULER in the interrupt vector corresponding to the timer interrupt. Hence, at the end of each time slice, the timer interrupt causes this routine to be executed.
- A program, together with any information that describes its current state of execution, is regarded by the as an entity called a process. A process can be in one of three states: Running, Runnable, or Blocked. The Running state means that the program is currently being executed. Assume that program A is in the Running state during a given time slice. At the end of that time slice, the timer interrupts the execution of this program and starts the execution of SCHEDULER. This is an operating system routine whose function is to determine which user program should run in the next time slice. It starts by saving all the information that will be needed later when execution of program A is resumed. The information saved, which is called

the program state, includes register contents, the program counter, and the processor status word.

- SCHEDULER selects for execution some other program, B, that was suspended earlier and is in the Runnable state. It restores all information saved at the time program R was suspended, including the contents of PS and PC, and executes a Retum-from-interrupt instruction. As a result, program B resumes execution for T seconds, at the end of which the timer clock raises an interrupt again, and a context switch to another runnable process takes place. Suppose that program A needs to read an input line from the keyboard. Instead of performing the operation itself, it requests I/O service from the operating system. It uses the stack or the processor registers to pass information to the OS describing the required operation, the I/O device, and the address of a buffer in the program data area where the line should be placed. Then it executes a software interrupt instruction. The interrupt vector for s instruction points to the OS SERVICES. This routine examines the information on the stack and initiates the requested operation by calling an appropriate OS routine. In our example, it calls, which is a routine responsible for starting I/O operations.
- While an I/O operation is in progress, the program that requested it cannot continue execution. Hence, the 10INIT routine sets the process associated with program A into the Blocked state, indicating to the scheduler that the program cannot resume execution at this time. The 10INIT routine carries out any preparations needed for the I/O operation, such as initializing address pointers and byte count, then calls a routine that performs the I/O transfers. It is common practice in operating system design to encapsulate all software pertaining to a particular device into a elf-contained module called the device driver. Such a module can be easily added to or deleted from the OS. We have assumed that the device driver for the keyboard consists of two routines, KBDINIT and KBDDATA, as shown n Figure 4.1 Oc. The 10INIT routine calls KBDINIT, which performs any initialization operations needed by the device or its interface circuit. KBDINIT also enables interrupts in the interface circuit by setting the appropriate bit in its control register, d then it returns to IOINIT, which returns to OSSERVICES. The keyboard is now ready to participate in a data transfer operation. It will generate an interrupt request whenever a key is pressed.
- Following the return to OSSERVICES, the SCHEDULER routine selects another user program to run. Of course, the scheduler will not select program A, because that program is now in the Blocked state. The Return-from-interrupt instruction that causes the selected user program to begin execution will also enable interrupts in the processor by loading new contents into the

processor status register. Thus, an interrupt request generated by the keyboard's interface will be accepted. The interrupt vector for this interrupt points to an OS routine called 10DATA. Because there could be several devices connected to the same interrupt request line, 10DATA begins by polling these devices to determine the one requesting service. Then, it calls the appropriate device driver to service the request. In our example, the driver called will be KBDDATA, which will transfer one character of data. If the character is a Carriage Return, it will also set to 1 a flag called END, to inform 10DATA that the requested I/O operation has been completed. At this point, the 10DATA routine changes the state of process A from Blocked to Runnable, so that the scheduler may select it for execution in some future time slice.

Handling Multiple Devices:

HANDLING MULTIPLE DEVICES: Let us now consider the situation where a number of devices capable of initiating interrupts are connected to the processor. Because these devices are operationally independent, there is no definite order in which they will generate interrupts. For example, device X may request an interrupt while an interrupt caused by device Y is being serviced, or several devices may request interrupts at exactly the same time. This gives rise to a number of questions:
1. How can the processor recognize the device requesting an interrupt?
2. Given that different devices are likely to require different interrupt -service routines, how can the processor obtain the starting address of the appropriate routine in each case?
3. Should a device be allowed to interrupt the processor while another interrupt is being serviced?
4. How should two or more simultaneous interrupt requests be handled? If two devices have activated the line at the same time, it must be possible to break the tie and elect one of the two requests for service. When the interrupt-service routine for the selected device has been completed, the second request can be serviced.

Vectored interrupts: To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding interrupt-service routine. The term vectored interrupts refers to all interrupt - handling schemes based on this approach.

A device requesting an interrupt can identify itself by sending a special code to the processor over the bus. This enables the processor to identify individual devices even if they share a single interrupt-request line. The code supplied by the device may represent the starting address of the interrupt-service routine for that device. The code length is typically in the range of 4 to 8 bits. The remainder of the address is supplied by the processor based on the area in its memory where the addresses for interrupt-service routines are located.

This arrangement implies that the interrupt-service routine for a given device must always start at the same location. The programmer can gain some flexibility by storing in this location an instruction that causes a branch to the appropriate routine. In many computers, this is done automatically by the interrupt-handling mechanism. The location pointed to by the interrupting device is used to store the starting address of the interrupt-service routine. The processor reads this address, called the interrupt vector, and loads it into the PC. The interrupt vector may also include a new value for the processor status register.

In most computers, I/O devices send the interrupt-vector code over the data bus, using the bus control signals to ensure that devices do not interfere with each other. When a device sends an interrupt request, the processor may not be ready to receive the interrupt-vector code immediately. For example, it must first complete the execution of the current instruction, which may require the use of the bus. There may be further delays if interrupts happen to be disabled at the time the request is raised. The interrupting device must wait to put data on the bus only when the processor is ready to receive it. When the processor is ready to receive the interrupt-vector code, it activates the interrupt-acknowledge line, INTA. The I/O device responds by sending its interrupt- vector code and turning off the INTR signal.

Interrupt nesting: I/O devices should be organized in a priority structure. An interrupt request from a high priority device should be accepted while the processor is servicing another request from a lower priority device. A multiple-level priority organization means that during execution of an interrupt service routine, interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. To implement this scheme, priority level can be assigned to the processor that can be changed by the program.
The processor's priority is usually encoded in a few bits of the processor status word. It can be changed by program instructions that write into the PS. These are privileged instructions, which can be executed only while the processor is running in the supervisor mode. The processor is in the supervisor mode only when executing operating system routines. It switches to the user mode before beginning to execute application programs. Thus, a user program cannot accidentally, or intentionally, change the priority of the processor and disrupt the system's operation. An attempt to execute a privileged instruction while in the user mode leads to a special type of interrupt called a privilege exception.
A multiple-priority scheme can be implemented easily by using separate interrupt- request and interrupt-acknowledge lines for each device, as shown in Figure 5.7. Each of the interrupt request lines is assigned a different priority level. Interrupt requests received over these lines are sent to a priority arbitration circuit in the processor. A request is accepted only if it has a higher priority level than that currently assigned to the processor.

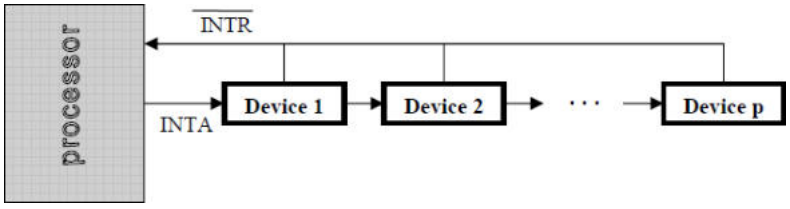


Figure 5.7 Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

Simultaneous request: When simultaneous interrupt requests are arrived from two or more I/O devices to the processor, the processor must have some means of deciding which request to service first.

Using a priority scheme such as that of Figure 5.7, the solution is straightforward. The processor simply accepts the request having the highest priority. If several devices share one interrupt-request line, as in Figure 5.6, some other mechanism is needed. Polling the status registers of the I/O devices is the simplest such mechanism. In this case, priority is determined by the order in which the devices are polled. When vectored interrupts are used, we must ensure that only one device is selected to send its interrupt vector code. A widely used scheme is to connect the devices to form a daisy chain, as shown in Figure 5.8. The interruptrequest line INTR is common to all devices. The interrupt-acknowledge line, INTA, is connected in a daisy-chain fashion, such that the INTA signal propagates serially through the devices. When several devices raise an interrupt request and the INTR line is activated, the processor responds by setting the INTA line to 1. This signal is received by device 1. Device 1 passes the signal on to device 2 only if it does not require any service. If device 1 has a pending request for interrupt, it blocks the INTA signal and proceeds to put its identifying code on the data lines. Therefore, in the daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority. The second device along the chain has second highest priority, and so on.

Controlling Device Requests:

CONTROLLING DEVICE REQUESTS: The control needed is usually provided in the form of an interrupt-enable bit in the device's interface circuit. The keyboard interrupt-enable, KEN, and display interrupt- enable, DEN, flags in register CONTROL in Figure 5.3 perform this function. If either of these flags is set, the interface circuit generates an interrupt request whenever the corresponding status flag in register STATUS is set. At the same time, the interface circuit sets bit KIRQ or DIRQ to indicate that the keyboard or display unit, respectively, is requesting an interrupt. If an interrupt-enable bit is equal to 0, the interface circuit will not generate an interrupt request, regardless of the state of the status flag.
There are two independent mechanisms for controlling interrupt requests. At the device end, an interrupt-enable bit in a control register determines whether the device is allowed to generate an interrupt request. At the processor end, either an interrupt enable bit in the PS register or a priority structure determines whether a given interrupt request will be accepted.

EXCEPTIONS: The term exception is often used to refer to any event that causes an interruption. Hence, I/O interrupts are one example of an exception.

Recovery from errors: Computers use a variety of techniques to ensure that all hardware components are operating properly. For example, many computers include an error-checking code in the main memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it and informs the processor by raising an interrupt. The processor may

also interrupt a program if it detects an error or an unusual condition while executing the instructions of this program. For example, the OP-code field of an instruction may not correspond to any legal instruction, or an arithmetic instruction may attempt a division by zero. When exception processing is initiated as a result of such errors, the processor proceeds in exactly the same manner as in the case of an I/O interrupt request. It suspends the program being executed and starts an exception-service routine. This routine takes appropriate action to recover from the error, if possible, or to inform the user about it. Recall that in the case of an I/O interrupt, the processor completes execution of the instruction in progress before accepting the interrupt. However, when an interrupt is caused by an error, execution of the interrupted instruction cannot usually be completed, and the processor begins exception processing immediately.

Debugging: Another important type of exception is used as an aid in debugging programs. System software usually includes a program called a debugger, which helps the programmer find errors in a program. The debugger uses exceptions to provide two important facilities called trace and breakpoints.
When a processor is operating in the trace mode, an exception occurs after execution of every instruction, using the debugging program as the exception-service routine. The debugging program enables the user to examine the contents of registers, memory locations, and so on. On return from the debugging program, the next instruction in the program being debugged is executed, then the debugging program is activated again. The trace exception is disabled during the execution of the debugging program. Breakpoints provide a similar facility, except that the program being debugged is interrupted only at specific points selected by the user. An instruction called Trap or Software interrupt is usually provided for this purpose. Execution of this instruction results in exactly the same actions as when a hardware interrupt request is received. While debugging a program, the user may wish to interrupt program execution after instruction i. The debugging routine saves instruction i 1 and replaces it with a software interrupt instruction. When the program is executed and reaches that point, it is interrupted and the debugging routine is activated. This gives the user a chance to examine memory and register contents. When the user is ready to continue executing the program being debugged, the debugging routine restores the saved instruction that was at location i 1 and executes a Return-from-interrupt instruction.

Privilege exception: To protect the operating system of a computer from being corrupted by user programs, certain instructions can be executed only while the processor is in the supervisor mode. These are called privileged instructions. For example, when the processor is running in the user mode, it will not execute an instruction that changes the priority level of the processor or that enables a user program to access areas in the computer memory that have been allocated to other users. An attempt to execute such an instruction will produce a privilege exception, causing the processor to switch to the supervisor mode and begin executing an appropriate routine in the operating system.

Use of Interrupts in Operating System:

USE OF INTERRUPTS IN OPERATING SYSTEMS: The operating system (OS) is responsible for coordinating all activities within a computer. It makes extensive use of interrupts to perform I/O operations and communicate with and control the execution of user programs. The operating system incorporates the interrupt-service routines (ISR) for all devices connected to a

computer. Application programs do not perform I/O operations themselves. An operating system provides a variety of services to application programs. To facilitate the implementation of these services, most processors have several different software interrupt instructions, each with its own interrupt vector. They can be used to call different parts of the OS.

- In a computer that has both a supervisor and a user mode, the processor switches its operation to supervisor mode at the time it accepts an interrupt request. It does so by setting a bit in the processor status register after saving the old contents of that register on the stack. Thus, when an application program calls the as by a software interrupt instruction, the processor automatically switches to supervisor mode, giving the as complete access to the computer's resources. When the as executes a Return-from-interrupt instruction, the processor status word belonging to the application program is restored from the stack. As a result, the processor switches back to the user mode.
- Multitasking is a mode of operation in which a processor executes several user programs at the same time. A common as technique that makes this possible is called time slicing. when operating system is started, an initialization routine OSINIT called for initialization. OSINIT loads the starting address of a routine called SCHEDULER in the interrupt vector corresponding to the timer interrupt. Hence, at the end of each time slice, the timer interrupt causes this routine to be executed.
- A program, together with any information that describes its current state of execution, is regarded by the as an entity called a process. A process can be in one of three states: Running, Runnable, or Blocked. The Running state means that the program is currently being executed. Assume that program A is in the Running state during a given time slice. At the end of that time slice, the timer interrupts the execution of this program and starts the execution of SCHEDULER. This is an operating system routine whose function is to determine which user program should run in the next time slice. It starts by saving all the information that will be needed later when execution of program A is resumed. The information saved, which is called the program state, includes register contents, the program counter, and the processor status word.
- SCHEDULER selects for execution some other program, B, that was suspended earlier and is in the Runnable state. It restores all information saved at the time program R was suspended, including the contents of PS and PC, and executes a Retum-from-interrupt instruction. As a result, program B resumes execution for T seconds, at the end of which the timer clock raises an interrupt again, and a context switch to another runnable process takes place. Suppose that program A needs to read an input line from the keyboard. Instead of performing the

operation itself, it requests I/O service from the operating system. It uses the stack or the processor registers to pass information to the OS describing the required operation, the I/O device, and the address of a buffer in the program data area where the line should be placed. Then it executes a software interrupt instruction. The interrupt vector for s instruction points to the OS SERVICES. This routine examines the information on the stack and initiates the requested operation by calling an appropriate OS routine. In our example, it calls, which is a routine responsible for starting I/O operations.

- While an I/O operation is in progress, the program that requested it cannot continue execution. Hence, the 10INIT routine sets the process associated with program A into the Blocked state, indicating to the scheduler that the program cannot resume execution at this time. The 10INIT routine carries out any preparations needed for the I/O operation, such as initializing address pointers and byte count, then calls a routine that performs the I/O transfers. It is common practice in operating system design to encapsulate all software pertaining to a particular device into a elf-contained module called the device driver. Such a module can be easily added to or deleted from the OS. We have assumed that the device driver for the keyboard consists of two routines, KBDINIT and KBDDATA, as shown n Figure 4.1 Oc. The 10INIT routine calls KBDINIT, which performs any initialization operations needed by the device or its interface circuit. KBDINIT also enables interrupts in the interface circuit by setting the appropriate bit in its control register, d then it returns to IOINIT, which returns to OSSERVICES. The keyboard is now ready to participate in a data transfer operation. It will generate an interrupt request whenever a key is pressed.
- Following the return to OSSERVICES, the SCHEDULER routine selects another user program to run. Of course, the scheduler will not select program A, because that program is now in the Blocked state. The Return-from-interrupt instruction that causes the selected user program to begin execution will also enable interrupts in the processor by loading new contents into the processor status register. Thus, an interrupt request generated by the keyboard's interface will be accepted. The interrupt vector for this interrupt points to an OS routine called 10DATA. Because there could be several devices connected to the same interrupt request line, 10DATA begins by polling these devices to determine the one requesting service. Then, it calls the appropriate device driver to service the request. In our example, the driver called will be KBDDATA, which will transfer one character of data. If the character is a Carriage Return, it will also set to 1 a flag called END, to inform 10DATA that the requested I/O operation has been completed. At this

point, the 10DATA routine changes the state of process A from Blocked to Runnable, so that the scheduler may select it for execution in some future time slice.

Direct Memory Access:

DIRECT MEMORY ACCESS (DMA): The main idea of direct memory access (DMA) is to enable peripheral devices to cut out the "middle man" role of the CPU in data transfer. It allows peripheral devices to transfer data directly from and to memory without the intervention of the CPU. Having peripheral devices access memory directly would allow the CPU to do other work, which would lead to improved performance, especially in the cases of large transfers. The DMA controller is a piece of hardware that controls one or more peripheral devices. It allows devices to transfer data to or from the system's memory without the help of the processor. In a typical DMA transfer, some event notifies the DMA controller that data needs to be transferred to or from memory. Both the DMA and CPU use memory bus and only one or the other can use the memory at the same time. The DMA controller then sends a request to the CPU asking its permission to use the bus. The CPU returns an acknowledgment to the DMA controller granting it bus access. The DMA can now take control of the bus to independently conduct memory transfer. When the transfer is complete the DMA relinquishes its control of the bus to the CPU. Processors that support DMA provide one or more input signals that the bus requester can assert to gain control of the bus and one or more output signals that the CPU asserts to indicate it has relinquished the bus. Figure 8.10 shows how the DMA controller shares the CPU's memory bus.

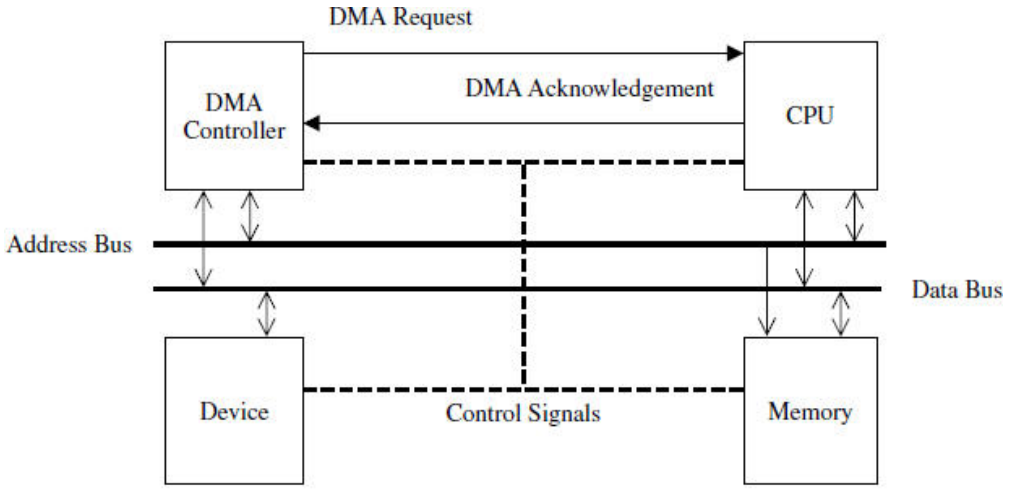


Figure 8.10 DMA controller shares the CPU's memory bus

Direct memory access controllers require initialization by the CPU. Typical setup parameters include the address of the source area, the address of the destination area, the length of the block, and whether the DMA controller should generate a processor interrupt once the block transfer is complete. A DMA controller has an address register, a word count register, and a control register. The address register contains an address that specifies the memory location of the data to be transferred. It is typically possible to have the DMA controller automatically increment the address register after each word transfer, so that the next transfer will be from the next memory location.

The word count register holds the number of words to be transferred. The word count is decremented by one after each word transfer. The control register specifies the transfer mode.

Direct memory access data transfer can be performed in burst mode or singlecycle mode. In burst mode, the DMA controller keeps control of the bus until all the data has been transferred to (from) memory from (to) the peripheral device. This mode of transfer is needed for fast devices where data transfer cannot be stopped until the entire transfer is done. In single-cycle mode (cycle stealing), the DMA controller relinquishes the bus after each transfer of one data word. This minimizes the amount of time that the DMA controller keeps the CPU from controlling the bus, but it requires that the bus request/acknowledge sequence be performed for every single transfer. This overhead can result in a degradation of the performance. The single-cycle mode is preferred if the system cannot tolerate more than a few cycles of added interrupt latency or if the peripheral devices can buffer very large amounts of data, causing the DMA controller to tie up the bus for an excessive amount of time.

The following steps summarize the DMA operations:

1. DMA controller initiates data transfer.
2. Data is moved (increasing the address in memory, and reducing the count of words to be moved).
3. When word count reaches zero, the DMA informs the CPU of the termination by means of an interrupt.
4. The CPU regains access to the memory bus.

A DMA controller may have multiple channels. Each channel has associated with it an address register and a count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. While the transfer is taking place, the CPU is free to do other things. When the transfer is complete, the CPU is interrupted. Direct memory access channels cannot be shared between device drivers. A device driver must be able to determine which DMA channel to use. Some devices have a fixed DMA channel, while others are more flexible, where the device driver can simply pick a free DMA channel to use.

Linux tracks the usage of the DMA channels using a vector of dma_chan data structures (one per DMA channel). The dma_chan data structure contains just two fields, a pointer to a string describing the owner of the DMA channel and a flag indicating if the DMA channel is allocated or not.

Buses:

BUSES: A bus in computer terminology represents a physical connection used to carry a signal from one point to another. The signal carried by a bus may represent address, data, control signal, or power. Typically, a bus consists of a number of connections running together. Each connection is called a bus line. A bus line is normally identified by a number. Related groups of bus lines are usually identified by a name. For example, the group of bus lines 1 to 16 in a given computer system may be used to carry the address of memory locations, and therefore are identified as address lines.

Depending on the signal carried, there exist at least four types of buses: address, data, control, and power buses. Data buses carry data, control buses carry control signals, and power buses carry the power-supply/ground voltage. The size (number of lines) of the address, data, and control bus varies from one system to another. Consider, for example, the bus connecting a CPU and memory in a given system, called the CPU bus. The size of the memory in that system is 512Mword and each word is 32 bits. In such system, the size of the address bus should be $\log2(5122^{20}) = 29$ lines, the size of the data bus should be 32 lines, and at least one control line (RR=W) should exist in that system. In addition to carrying control signals, a control bus can carry timing signals. These are signals used to determine the exact timing for data transfer to and from a bus; that is, they determine when a given computer system component, such as the processor, memory, or I/O devices, can place data on the bus and when they can receive data from the bus. A bus can be synchronous if data transfer over the bus is controlled by a bus clock. The clock acts as the timing reference for all bus signals. A bus is asynchronous if data transfer over the bus is based on the availability of the data and not on a clock signal. Data is transferred over an asynchronous bus using a technique called handshaking. The operations of synchronous and asynchronous buses are explained below.

To understand the difference between synchronous and asynchronous, let us consider the case when a master such as a CPU or DMA is the source of data to be transferred to a slave such as an I/O device. The following is a sequence of events involving the master and slave:

1. Master: send request to use the bus
2. Master: request is granted and bus is allocated to master
3. Master: place address/data on bus
4. Slave: slave is selected
5. Master: signal data transfer
6. Slave: take data
7. Master: free the bus

Synchronous and Asynchronous Buses:

Bus Protocols: A bus is a communication channel shared by many devices and hence rules need to be established in order for the communication to happen correctly. These rules are called bus protocols. Design of a bus architecture involves several tradeoffs related to the width of the data bus, data transfer size, bus protocols, clocking, etc. Depending on whether the bus transactions are controlled by a clock or not, buses are classified into synchronous and asynchronous buses. Depending on whether the data bits are sent on parallel wires or multiplexed onto one single wire, there are parallel and serial buses. Control of the bus communication in the presence of multiple devices necessitates defined procedures called arbitration schemes. In this section, different kinds of buses and arbitration schemes are described.

Synchronous Buses: In synchronous buses, the steps of data transfer take place at fixed clock cycles. Everything is synchronized to bus clock and clock signals are made available to both master and slave. The bus clock is a square wave signal. A cycle starts at one rising edge of the clock and ends at the next rising edge, which is the beginning of the next cycle. A transfer may take multiple bus cycles depending on the speed parameters of the bus and the two ends of the transfer.One scenario would be that on the first clock cycle, the master puts an address on the address bus, puts data on the data bus, and asserts the appropriate control lines. Slave recognizes its address on the address bus on the first cycle and reads the new value from the bus in the second cycle. Synchronous buses are simple and easily implemented. However, when connecting devices with varying speeds to a synchronous bus, the slowest device will determine the speed of the bus. Also, the synchronous bus length could be limited to avoid clock-skewing problems.

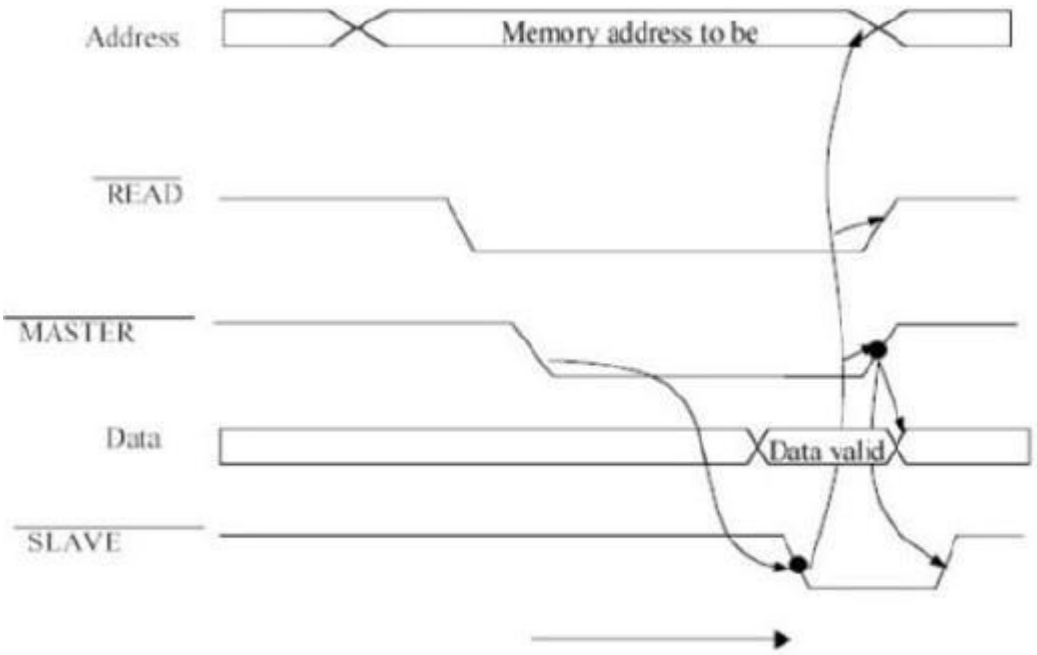


Figure 6. Read Operations on an Asynchronous Bus

A memory read transaction on the synchronous bus typically proceeds as illustrated in Fig. 5. During the first clock cycle the CPU places the address of the location it wants to read, on the address lines of the bus. Later during the same clock cycle, once the address lines have stabilized,

the READ request is asserted by the CPU. Many times, some of these control signals are active low and asserting the signal means that they are pulled low. A few clock cycles are needed for the memory to perform accessing of the requested location. In a simple non-pipelined bus, these appear as wait states and the data is placed on the bus by the memory after the tow or three wait cycles. The CPU then releases the bus by deasserting the READ control signal. The write transaction is similar except that the processor is the data source and the WRITE signal is the one that is asserted. Different bus architectures synchronize bus operations with respect to the rising edge or falling edge or level of the clock signal.

Asynchronous Buses: There are no fixed clock cycles in asynchronous buses. Handshaking is used instead. Figure 8.11 shows the handshaking protocol. The master asserts the data-ready line

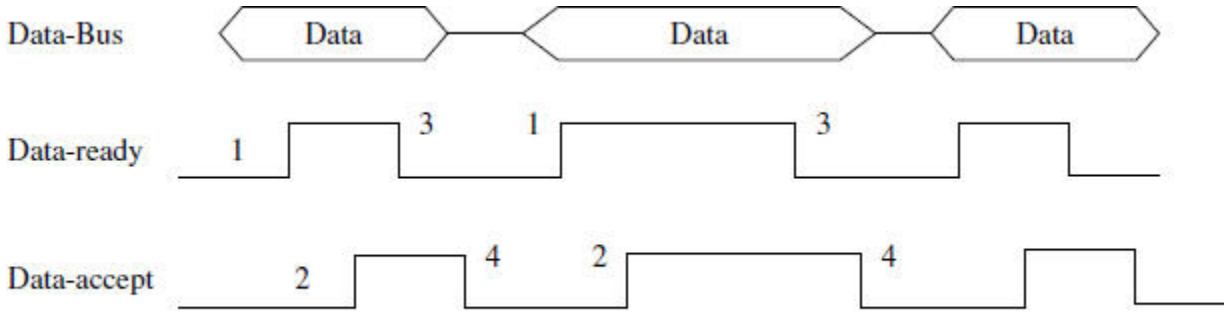


Figure 8.11 Asynchronous bus timing using handshaking protocol

(point 1 in the figure) until it sees a data-accept signal. When the slave sees a dataready signal, it will assert the data-accept line (point 2 in the figure). The rising of the data-accept line will trigger the falling of the data-ready line and the removal of data from the bus. The falling of the data-ready line (point 3 in the figure) will trigger the falling of the data-accept line (point 4 in the figure). This handshaking, which is called fully interlocked, is repeated until the data is completely transferred. Asynchronous bus is appropriate for different speed devices.

An asynchronous bus has no system clock. Handshaking is done to properly conduct the transmission of data between the sender and the receiver. The process is illustrated in Fig. 6. For example, in an asynchronous read operation, the bus master puts the address and control signals on the bus and then asserts a synchronization signal. The synchronization signal from the master prompts the slave to get synchronized and once it has accessed the data, it asserts its own synchronization signal. The slave's synchronization signal indicates to the processor that there is valid data on the bus, and it reads the data. The master then deasserts its synchronization signal, which indicates to the slave that the master has read the data. The slave then deasserts its synchronization signal. This method of synchronization is referred to as a full handshake. Note

that there is no clock and that starting and ending of the data transfer are indicated by special synchronization signals. An asynchronous communication protocol can be considered as a pair of Finite State machines (FSMs) that operate in such a way that one FSM does not proceed until the other FSM has reached a certain state.

Synchronous buses are typically faster than asynchronous buses because there is no overhead to establish a time reference for each transaction. Another reason that helps the synchronous bus to operate fast is that the bus protocol is predetermined and very little logic is involved in implementing the Finite State machine. However, synchronous buses are affected by clock skew and they cannot be very long. But asynchronous buses work well even when they are long because clock skew problems do not affect them. Thus asynchronous buses can handle longer physical distances and higher number of devices. Processor-memory buses are typically synchronous because the devices connected to the bus are fast, are small in number and are located in close proximity. I/O buses are typically asynchronous because many peripherals need only slow data rates and are physically situated far away.

Bus Arbitration:

Bus Arbitration: Bus arbitration is needed to resolve conflicts when two or more devices want to become the bus master at the same time. In short, arbitration is the process of selecting the next bus master from among multiple candidates. Conflicts can be resolved based on fairness or priority in a centralized or distributed mechanisms. Centralized Arbitration In centralized arbitration schemes, a single arbiter is used to select the next master. A simple form of centralized arbitration uses a bus request line, a bus grant line, and a bus busy line. Each of these lines is shared by potential masters, which are daisy-chained in a cascade. Figure 8.12 shows this simple centralized arbitration scheme. In the figure, each of the potential masters can submit a bus request at any time.

A fixed priority is set among the masters from left to right. When a bus request is received at the central bus arbiter, it issues a bus grant by asserting the bus grant line. When the potential master that is closest to the arbiter (potential master 1) sees the bus grant signal, it checks to see if it had made a bus request. If yes, it takes over the bus and stops propagation of the bus grant signal any further. If it has not made a request, it will simple turn the bus grant signal to the next master to the right (potential master 2), and so on. When the transaction is complete, the busy line is deasserted.

Instead of using shared request and grant lines, multiple bus request and bus grant lines can be used. In one scheme, each master will have its own independent request and grant line as shown in Figure 8.13. The central arbiter can employ any prioritybased or fairness-based tiebreaker. Another scheme allows the masters to have multiple priority levels. For each priority level, there is a bus request and a bus grant line. Within each priority level, daisy chain is used. In this scheme, each device is attached to the daisy chain of one priority level. If the arbiter receives multiple

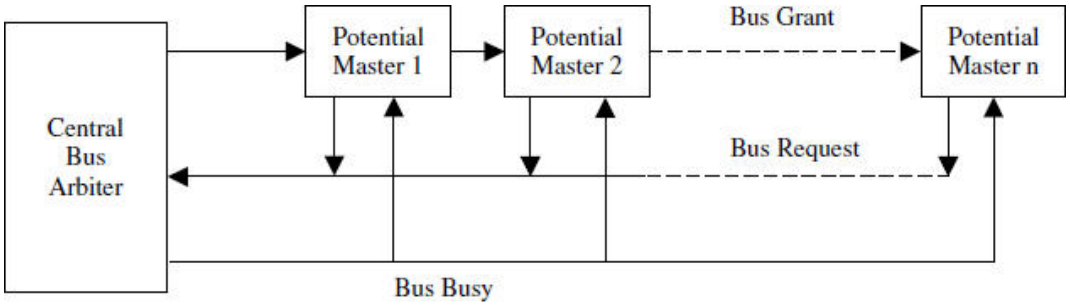


Figure 8.12 Centralized arbiter in a daisy-chain scheme

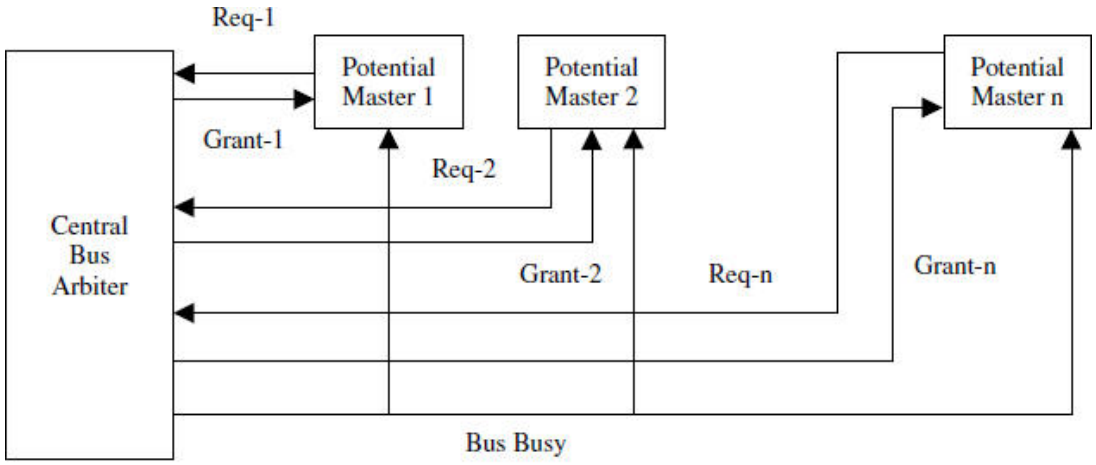


Figure 8.13 Centralized arbiter with independent request and grant lines

bus requests from different levels, it grants the bus to the level with the highest priority. Daisy chaining is used among the devices of that level. Figure 8.14 shows an example of four devices included in two priority levels. Potential master 1 and potential master 3 are daisy-chained in level 1 and potential master 2 and potential master 4 are daisy-chained in level 2.

Decentralized Arbitration In decentralized arbitration schemes, priority-based arbitration is usually used in a distributed fashion. Each potential master has a unique arbitration number, which is used in resolving conflicts when multiple requests are submitted. For example, a conflict can always be resolved in favor of the device with the highest arbitration number. The question now is how to determine which device has the highest arbitration number? One method is that a requesting device would make its unique arbitration number available to all other devices. Each device compares that number with its own arbitration number. The device with the smaller number is always dismissed. Eventually, the requester with the highest arbitration number will survive and be granted bus access.

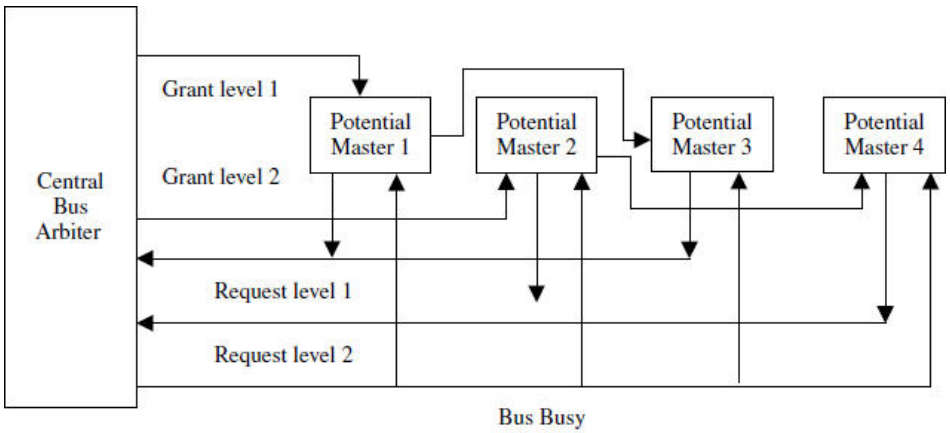


Figure 8.14 Centralized arbiter with two priority levels (four devices)

Interface Circuits:

Interface circuits: An Input/output (I/O) interface consists of the circuitry required to connect an I/O device to a computer bus. On one side of the interface we have the bus signals for address, data, and control. On the other side we have a data path with its associated controls to transfer data between the interface and the I/O device. This side is called a port, and it can be classified as either a parallel or a serial port. A parallel port transfers data in the form of a number of bits, typically 8 or 16, simultaneously to or from the device. A serial port transmits and receives data one bit at a time. Communication with the bus is the same for both formats; the conversion from the parallel to the serial format, and vice versa, takes place inside the interface circuit. I/O interface does the following:

1. Provides a storage buffer for at least one word of data (or one byte, in the case of byte-oriented devices)
2. Contains status flags that can be accessed by the processor to determine whether the buffer is full (for input) or empty (for output)
3. Contains address-decoding circuitry to determine when it is being addressed by the processor
4. Generates the appropriate timing signals required by the bus control scheme
5. Performs any format conversion that may be necessary to transfer data between the bus and the I/O device, such as parallel-serial conversion in the case of a serial port

Parallel port: Figure 5.20 shows the hardware components needed for connecting a keyboard to a processor. A typical keyboard consists of mechanical switches that are normally open. When a key is pressed, its switch closes and establishes a path for an electrical signal. This signal is

detected by an encoder circuit that generates the ASCII code for the corresponding character. A difficulty with such push-button switches is that the contacts bounce when a key is pressed. Although bouncing may last only one or two milliseconds, this is long enough for the computer to observe a single pressing of a key as several distinct electrical events; this single pressing could be erroneously interpreted as the key being pressed and released rapidly several times. The effect of bouncing must be eliminated. We can do this in two ways: A simple de-bouncing circuit can be included, or a software approach can be used. When debouncing is implemented in software, the I/O routine that reads a character from the keyboard waits long enough to ensure that bouncing has subsided. Figure 5.20 illustrates the hardware approach; debouncing circuits are included as a part of the encoder block.

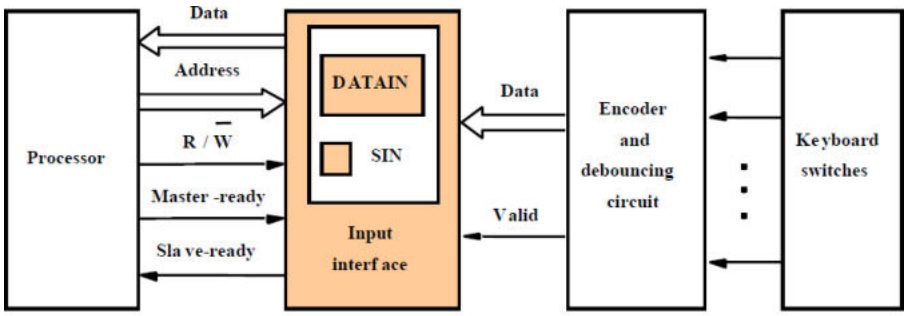


Figure 5.20 Keyboard to processor connection.

The output of the encoder consists of the bits that represent the encoded character and one control signal called Valid, which indicates that a key is being pressed. This information is sent to the interface circuit, which contains a data register, DATAIN, and a status flag, SIN. When a key is pressed, the valid signal changes from 0 to 1,, causing the ASCII code to be loaded into DATAIN and SIN to be set to 1. The status flag SIN is cleared to 0 when the processor reads the contents of the DATAIN register. The interface circuit is connected to an asynchronous bus on which transfers are controlled using the handshake signals Master-ready and Slave-ready. The third control line, R/W distinguishes read and write transfers.

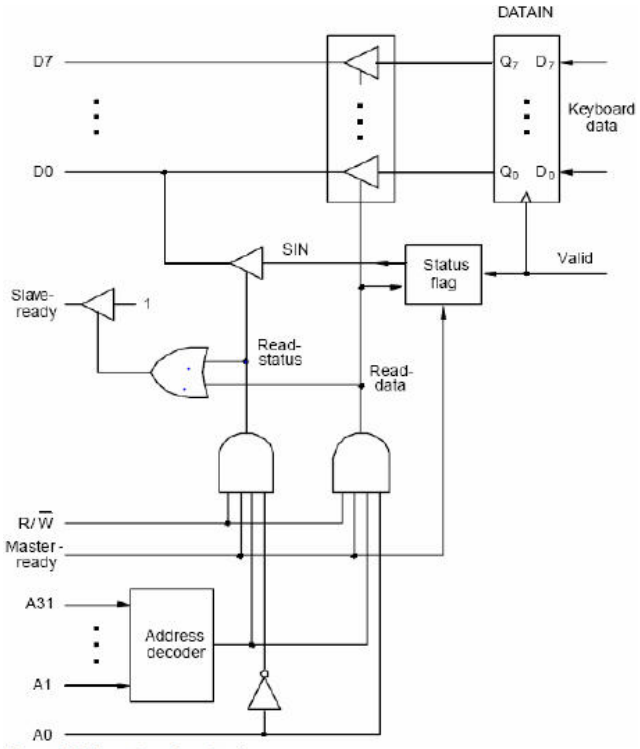


Figure 5.21 Input interface circuit.

Figure 5.21 shows a suitable circuit for an input interface.

The output lines of the DATAIN register are connected to the data lines of the bus by means of three-state drivers, which are turned on when the processor issues a read instruction with the address that selects this register. The SIN signal is generated by a status flag circuit. This signal is also sent to the bus through a three-state driver. It is connected to bit DO, which means it will appear as bit 0 of the status register. Other bits of this register do not contain valid information. An address decoder is used to select the input interface when the high-order 31 bits of an address correspond to any of the addresses assigned to this interface. Address bit AO determines whether the status or the data registers is to be read when the Master-ready signal is active. The control handshake is accomplished by activating the Slaveready signal when either Read-status or Read-data is equal to 1.

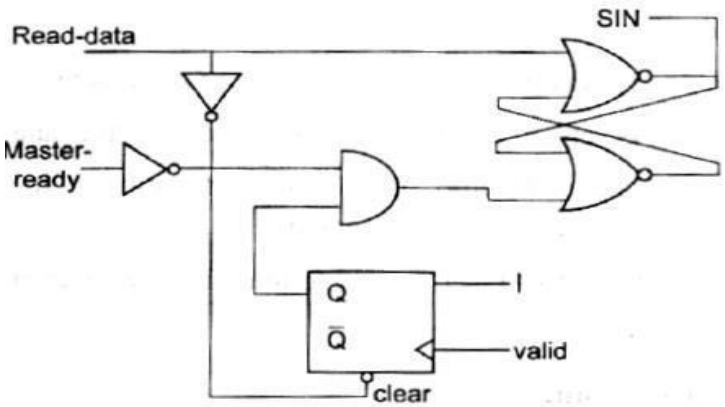


Figure 5.22 Circuit for the status Rag block in Figure 5.21.

A possible implementation of the status flag circuit is shown in Figure 5.21. An edgetriggered D flip-flop is set to 1 by a rising edge on the Valid signal line. this event changes the state of the NOR latch such that SIN is set to 1. The state of this latch must not change while SIN is being read by the processor. Hence, the circuit ensures that SIN can be set only while Masterready is equal to 0. Both the flip- flop and the latch are reset to 0 when Read-data is set to 1 to read the DATAIN register.

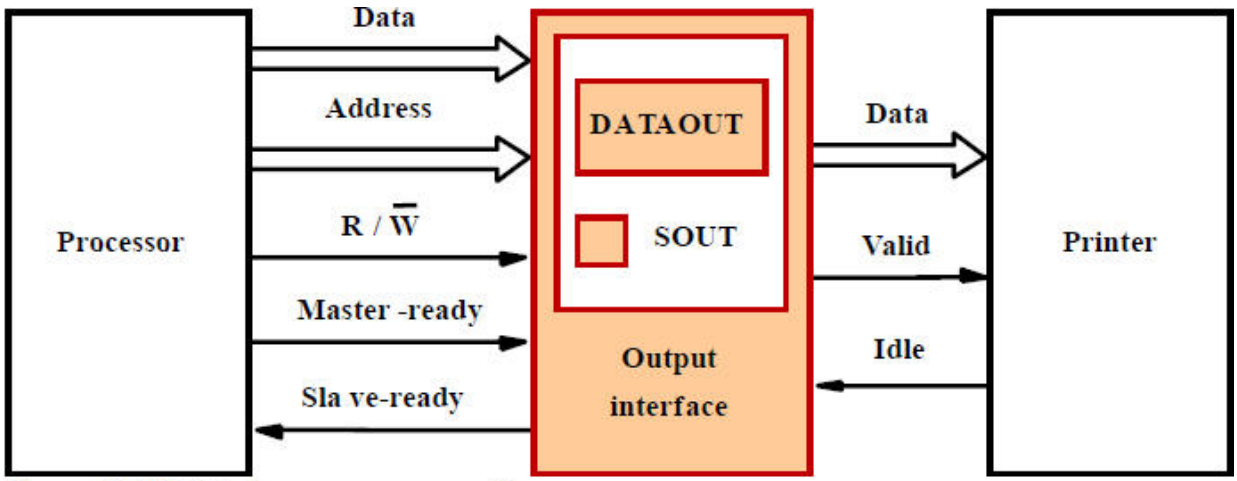


Figure 5.23 Printer to processor connection.

Let us now consider an output interface that can be used to connect an output device, such as a printer, to a processor, as shown in Figure 5.23. The printer operates under control of the handshake signals Valid and Idle in a manner similar to the handshake used on the bus with the Master-ready and Slave-ready signals. When it is ready to accept a character, the printer asserts its Idle signal. The interface circuit can then place a new character on the data lines and activate the Valid signal. In response, the printer starts printing the new character and negates the Idle signal, which in turn causes the interface to deactivate the Valid signal.

Serial port: A serial port is used to connect the processor to I/O devices that require transmission of data one bit at a time. The key feature of an interface circuit for a serial port is that it is capable of communicating in a bit-serial fashion on the device side and in a bit-parallel fashion on the bus side. The transformation between the parallel and serial formats is achieved with shift registers that have parallel access capability. A block diagram of a typical serial interface is shown in Figure 5.27. It includes the familiar DATAIN and DATAOUT registers. The input shift register accepts bit-serial input from the I/O device. When all 8 bits of data have been received, the contents of this shift register are loaded in parallel into the DATAIN register. Similarly, output data in the DATAOUT register are loaded into the output shift register, from which e bits are shifted out and sent to the I/O device.

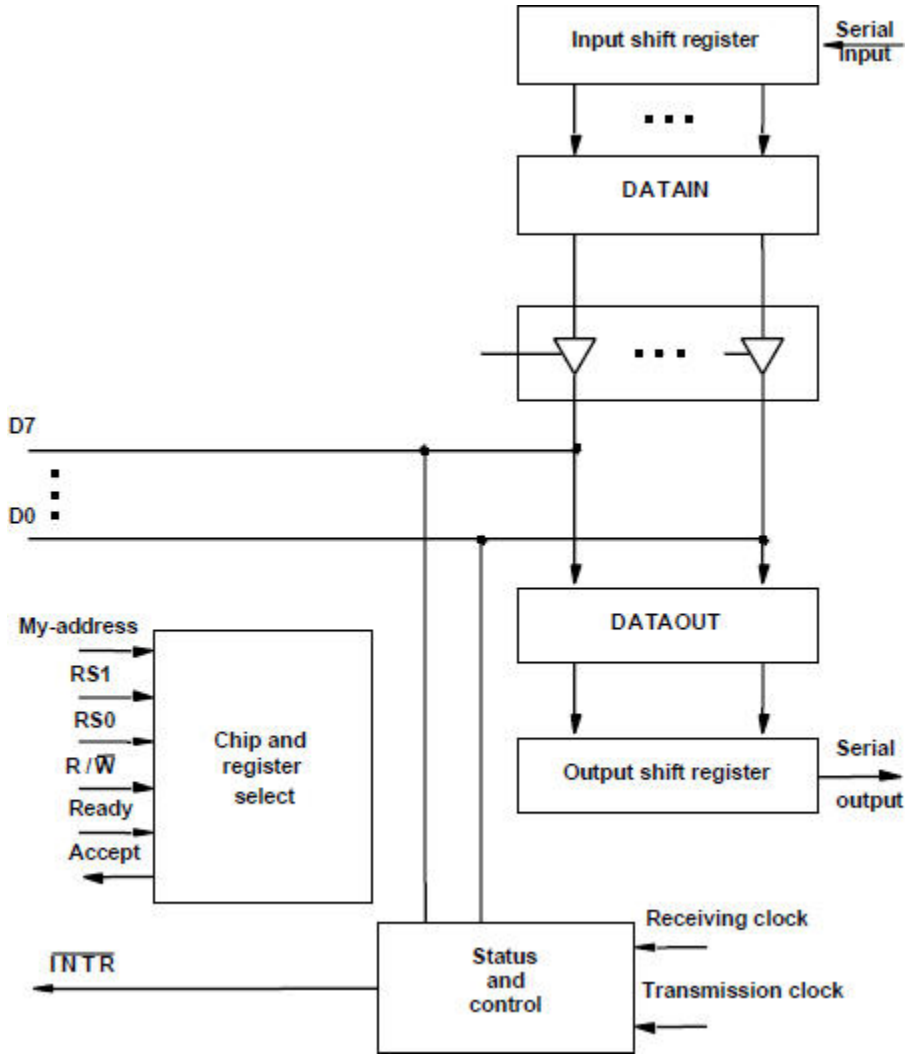


Figure 5.27 A serial interface.
The part of the interface that deals with the bus is the same as in the parallel interface described earlier. The status flags SIN and SOUT serve similar functions. The SIN flag is set to 1

- when new data are loaded in DATAIN; it is cleared to 0 when the processor reads the contents of DATAIN. As soon as the data are transferred from the input shift register into the DATAIN register, the shift register can start accepting the next 8-bit character from the I/O device. The SOUT flag indicates whether the output buffer is available. It is cleared to 0 when the processor writes new data into the DATAOUT register and set to 1 when data are transferred from DATAOUT into the output shift register.
- The double buffering used in the input and output paths are important. A simpler interface could be implemented by turning DATAIN and DATA OUT into shift registers and eliminating

the shift registers in Figure 5.27. However, this would impose awkward restrictions on the operation of the I/O device; after receiving one character from the serial line, the device cannot start receiving the next character until the processor reads the contents of DATAIN. Thus, a pause would be needed between two characters to allow the processor to read the input data. With the double buffer, the transfer of the second character can begin as soon as the first character is loaded from the shift register into the DATAIN register. Thus, provided the processor reads the contents of DATAIN before the serial transfer of the second character is completed, the interface can receive a continuous stream of serial data. An analogous situation occurs in the output path of the interface.

Because it requires fewer wires, serial transmission is convenient for connecting devices that are physically far away from the computer. The speed of transmission, often given as a bit rate, depends on the nature of the devices connected. To accommodate a range of devices, a serial interface must be able to use a range of clock speeds. The circuit in Figure 5.27 allows separate clock signals to be used for input and output operations for increased flexibility. Because serial interfaces play a vital role in connecting I/O devices, several widely used standards have been developed. A standard circuit that includes the features of our example in Figure 5.27 is known as a Universal Asynchronous Receiver Transmitter (UART). It is intended for use with low-speed serial devices. Data transmission is performed using the asynchronous start-stop format. To facilitate connection to communication links, a popular standard known as RS-232-C was developed.

Standard I/O Interfaces:

Introduction: A number of standards have been developed for I/O Interface. IBM developed a they called ISA (Industry Standard Architecture) for their personal computer known at the time as PC AT. The popularity of that computer led to other manufacturers producing ISA -compatible interfaces for their 110 devices, thus making ISA into a de facto standard. Some standards have been developed through industrial cooperative efforts, even among competing companies driven by their common self-interest in having compatible products. In some cases, organizations such as the IEEE (Institute of Electrical and Electronics Engineers), ANSI (American Nationa1 Standards Institute), or international bodies such as ISO (International Standards Organization) have blessed these standards and given them an official status.
There are three widely used bus standards, PCI (Peripheral Component Interconnect), SCSI (Small Computer System Interface), and USB (Universal Serial Bus). The way these standards are used in a typical computer system is illustrated in Figure 5.28. The PCI standard defines an expansion bus on the motherboard. SCSI and USB are used for connecting additional devices, both inside and outside the computer box. The SCSI bus is a high-speed parallel bus intended for devices such as disks and video displays. The USB bus uses serial transmission to suit the needs of equipment ranging from keyboards to game controls to internet connections. The figure shows an interface circuit that enables devices compatible with the earlier ISA standard, such as the popular IDE (Integrated Device Electronics) disk, to be connected. It also shows a connection to an Ethernet.

The Ethernet is a widely used local area network, providing a high-speed connection among computers in a building or a university campus. A given computer may use more than one bus standard. A typical Pentium computer has both a PCI bus and an ISA bus, thus providing the user with a wide range of devices to choose from.

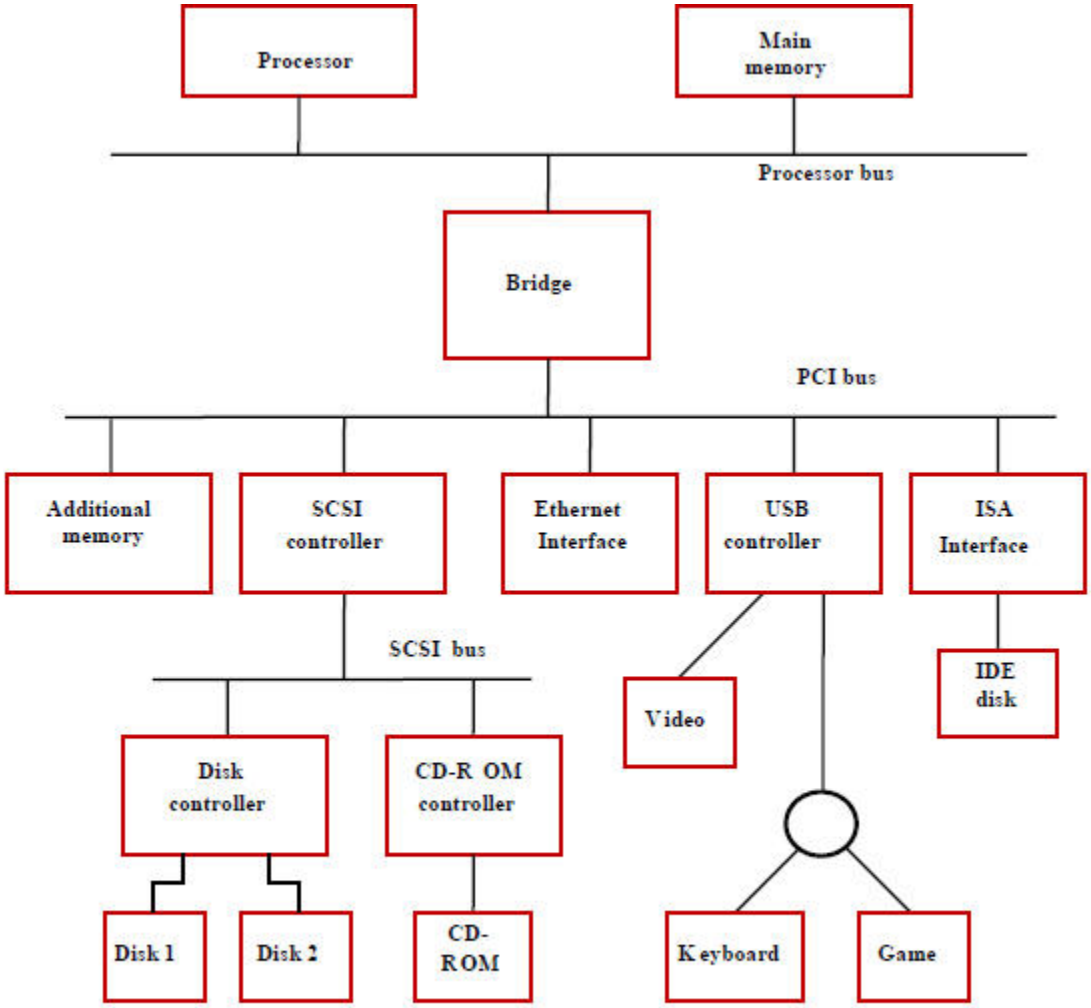


Figure 5.28 An example of a computer system using different interface standards.

5.7.1 PCI (Peripheral Component Interconnect)

The PCI bus is a good example of a system bus. It supports the functions found on a processor bus but in a standardized format that is independent of any particular processor. Devices connected to the PCI bus appear to the processor as if they were connected directly to the processor bus. They are assigned addresses in the memory address space of the processor. Early PCs used the 8-bit XT bus, whose signals closely mimicked those of Intel's 80x86 processors. Later, the 16-bit bus used on the PC AT computers became known as the ISA bus. Its extended 32- bit version is known as the EISA bus. Other buses developed in the eighties with similar capabilities are the Microchannel used in IBM PCs and the NuBus used in Macintosh computers.

The PCI was developed as a low-cost bus that is truly processor independent. Its design anticipated a rapidly growing demand for bus bandwidth to support high-speed disks and graphic and video devices, as well as the specialized needs of multiprocessor systems. As a result, the PCI is still popular as an industry standard almost a decade after it was first introduced in 1992. An important feature that the PCI pioneered is a plug-and-play capability for connecting I/O devices. To connect a new device, the user simply connects the device interface board to the bus. The software takes care of the rest.

Data Transfer: Most memory transfers involve a burst of data rather than just one word. The reason is that modem processors include a cache memory. The PCI is designed primarily to support this mode of operation. A read or a write operation involving a single word is simply treated as a burst of length one. The bus supports three independent address spaces: memory, I/O, and configuration. The first two are self-explanatory. The I/O address space is intended for use with processors, such as Pentium, that have a separate I/O address space. Figure 5.28 shows the main memory of the computer connected directly to the processor bus. An alternative arrangement that is used often with the PCI bus is shown in Figure 5.29. The PCI Bridge provides a separate physical connection for the main memory. For electrical reasons, the bus may be further divided into segments connected via bridges. However, regardless of which bus segment a device is connected to, it may still be mapped into the processor's memory address space.

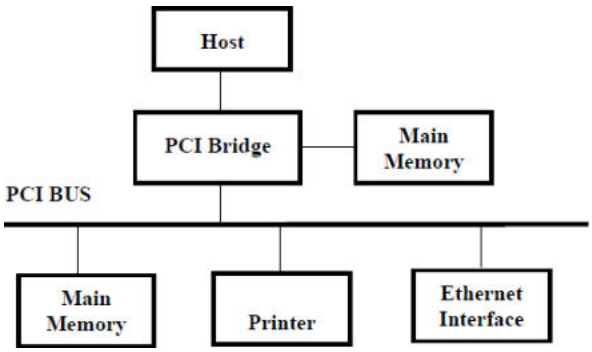


Figure 5.29 Use of a PCI bus in a computer system.: At any given time, one device is the bus master. It has the right to initiate data transfers by issuing read and write commands. A master is called an initiator in PCI terminology. This is either a processor or a DMA controller. The addressed device that responds to read and write commands is called a target. The main bus signals used for transferring data are listed in Table 5.1. Signals whose name ends with the symbol # are asserted when in the low- voltage state. The main difference between the PCI protocol with others is that in addition to a Target-ready signal, PCI also uses an Initiatorready signal, IRDY #. The latter is needed to support burst transfers.

SCSI Bus:

SCSI (Small Computer System Interface): The acronym SCSI stands for Small Computer System Interface. It refers to a standard bus defined by the American National Standards Institute (ANSI) under the designation X3.131. In the original specifications of the standard, devices such as disks are connected to a computer via a 50- wire cable, which can be up to 25 meters in length

and can transfer data at rates up to 5 megabytes/s. The SCSI bus standard has undergone many revisions, and its data transfer capability has increased very rapidly, almost doubling every two years. SCSI-2 and SCSI-3 have been defined, and each has several options. A SCSI bus may have eight data lines, in which case it is called a narrow bus and transfers data one byte at a time. Because of these various options, the SCSI connector may have 50, 68, or 80 pins. The maximum transfer rate in commercial devices that are currently available varies from 5 megabytes/s to 160 megabytes/so The most recent version of the standard is intended to support transfer rates up to 320 megabytes/s, and 640 megabytes/s is anticipated a little later. Devices connected to the SCSI bus are not part of the address space of the processor in the same way as devices connected to the processor bus. The SCSI bus is connected to the processor bus through a SCSI controller, as shown in Figure 5.28. This controller uses DMA to transfer data packets from the main memory to the device, or vice versa. A packet may contain a block of data, commands from the processor to the device, or status information about the device. A controller connected to a SCSI bus is one of two types - an initiator or a target. An initiator has the ability to select a particular target and to send commands specifying the operations to be performed. The disk controller operates as a target. It carries out the commands it receives from the initiator. The initiator establishes a logical connection with the intended target. Once this connection has been established, it can be suspended and restored as needed to transfer commands and bursts of data. While a particular connection is suspended, other devices can use the bus to transfer information. This ability to overlap data transfer requests is one of the key features of the SCSI bus that leads to its high performance.

Data transfers on the SCSI bus are always controlled by the target controller. To send a command to a target, an initiator requests control of the bus and, after winning arbitration, selects the controller it wants to communicate with and hands control of the bus over to it. Then the controller starts a data transfer operation to receive a command from the initiator. Assume that the processor wishes to read a block of data from a disk drive and that these data are stored in two disk sectors that are not contiguous. The processor sends a command to the SCSI controller, which causes the following sequence of events to take place:

1. The SCSI controller, acting as an initiator, contends for control of the bus.
2. When the 'initiator wins the arbitration process, it selects the target controller and hands over control of the bus to it.
3. The target starts an output operation (from initiator to target); in response to this, the initiator sends a command specifying the required read operation.
4. The target, realizing that it first needs to perform a disk seek operation, sends a message to the initiator indicating that it will temporarily suspend the connection between them. Then it releases the bus.
5. The target controller sends a command to the disk drive to move the read head to the first sector involved in the requested read operation. Then, it reads the data stored in that sector and stores them in a data buffer. When it is ready to begin transferring data to the initiator,

the target requests control of the bus. After it wins arbitration, it res elects the initiator controller, thus restoring the suspended connection.

6. The target transfers the contents of the data buffer to the initiator and then suspends the connection again. Data are transferred either 8 or 16 bits in parallel, depending on the width of the bus.
7. The target controller sends a command to the disk drive to perform another seek operation. Then, it transfers the contents of the second disk sector to the initiator, as before. At the end of this transfer, the logical connection between the two controllers is terminated.
8. As the initiator controller receives the data, it stores them into the main memory using the DMA approach.
9. The SCSI controller sends an interrupt to the processor to inform it that the requested operation has been completed.

This scenario shows that the messages exchanged over the SCSI bus are at a higher level than those exchanged over the processor bus. The SCSI bus standard defines a wide range of control messages that can be ex- changed between the controllers to handle different types of I/O devices. Messages are also defined to deal with various error or failure conditions that might arise during device operation or data transfer.

Table 5.1 Data transfer si9na on the PCI bus.

Name	Function
CLK	A 33-MHz or 66-MHz clock.
FRAME#	Sent by the initiator to indicate the duration of a transaction.
AD	32 address/data lines, which may be OJtiona1ly increased to 64.
C/BE#	4 command/byte-enable lines (8 for a 64-bit bus).
IRDY# , TRDY#	Initiator-ready and Target-ready signals.
DEVSEL#	A response from the device indicating that it has recognized its Address and is ready for a data transfer transaction.
IDSEL#	Initialization Device Select,

Consider a bus transaction in which the processor reads four 32-bit words from the memory. In this case, the initiator is the processor and the target is the memory. A complete transfer operation on the bus, involving an address and a burst of data, is called a transaction. Individual word transfers within a transaction are called phases. The sequence of events on the bus is shown in Figure 5.30. A clock signal provides the timing reference used to coordinate different phases of a transaction. All signal transitions are triggered by the rising edge of the clock. The signals changing later in the clock cycle to indicate the delays they encounter.

Bus Signal: The bus signals are summarized in Table 5.2. For simplicity we show the signals for a narrow bus (8 data lines). Note that all signal names are preceded by a minus sign. This indicates that the signals are active, or that a data line is equal to 1, when they are in the low-voltage state. The bus has no address lines. Instead, the data lines are used to identify the bus controllers involved during the selection or reselection process and during bus arbitration. For a

narrow bus, there are eight possible controllers, numbered o through 7, and each i, associated with the data line that has the same number. A wide bus accommodates up to 16 controllers. A controller places its own address or the address of another controller on the bus by activating the corresponding data line. Thus, it is possible to have more than one address on the bus at the same time, as in the arbitration process we describe next. Once a connection is established between two controllers, there is no further need for addressing, and the data lines are used to carry data.

Table 5.2 The SCSI bus signal

Category	Name	Function
Data	-DB(O) to -DB(7)	Data lines: Carry one byte of information during the information transfer phase and identify device during arbitration, selection and reselection phases
	-DB(P)	Parity bit for the data bus
Phase	-BSY	Busy: Asserted when the bus is not free
	-SEL	Selection: Asserted during selection and reelection
Information type	-C/D	Control/Data: Asserted during transfer of control information (command, status or message)
	-MSG	ssage: indicates that the information being transferred is a message
Handshake	-REQ	quest Asserted by a target to request a data transfer cycle
	-ACK	Acknowledge: Asserted by the initiator when it has completed a data transfer operation
Direction of transfer	-I/O	Input/Output. Asserted to indicate an input operation
Other	-ATN	ention: Asserted by an initiator when it wishes to send a message to a target
	-RST	set: Causes all Device controls to disconnect from the bus and their start-up state

The main phases involved in the operation of the SCSI bus are arbitration, selection, information transfer, and reselection.

Arbitration: The bus is free when the -BSY signal is in the inactive (high-voltage) state. Any controller can request the use of the bus while it is in this state. Since two or more controllers may generate such a request at the same time, an arbitration scheme must be implemented. A controller requests the bus by asserting the - BSY signal and by asserting its associated data line to identify itself. The SCSI bus uses a simple distributed arbitration scheme. It is illustrated by the example in Figure 5.32, in which controllers 2 and 6 request the use of the bus simultaneously. Each controller on the bus is assigned a fixed priority, with controller 7 having the highest priority. When -BSY becomes active, all controllers that are requesting the bus examine the data lines and determine whether a higher-priority device is requesting the bus at the same time. The controller using the highest-numbered line realizes that it has won the arbitration process. All other controllers disconnect from the bus and wait for -BSY to become inactive again.

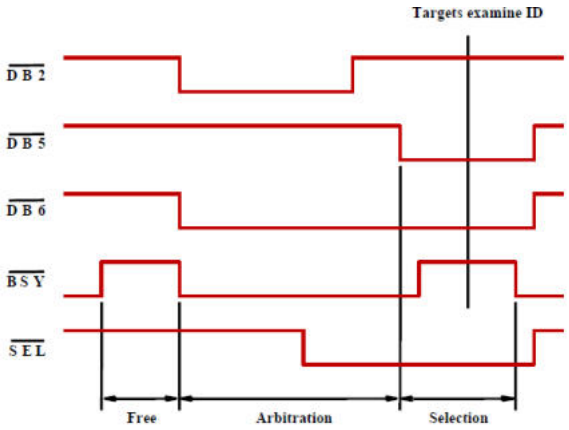


Figure 5.32 Arbitration and selection on the SCSI bus. Device6 wins arbitration and selects device2.

In Figure 5.32, we have assumed that controller 6 is an initiator that wishes to establish a connection to controller 5. After winning arbitration, controller 6 proceeds to the selection phase, in which it identifies the target.

Selection: Having won arbitration, controller 6 continues to assert -BSY and -DB6 (its address). It indicates that it wishes to select controller 5 by asserting the -SEL and then the - DB5 lines. Any other controller that may have been involved in the arbitration phase, such as controller 2 in the figure, must stop driving the data lines once the -SEL line becomes active, if it has not already done so. After placing the address of the target controller on the bus, the initiator releases the - BSY line. The selected target controller responds by asserting - BSY. This informs the initiator that the connection it is requesting has been established, so that it may remove the address information from the data lines. The selection process is now complete, and the target controller (controller 5) is asserting - BSY. From this point on, controller 5 has control of the bus, as required for the information transfer phase.

Information Transfer: The information transferred between two controllers may consist of commands from the initiator to the target, status responses from the target to the initiator, or data being transferred to or from the I/O device. Handshake signaling is used to control information transfers. The - REQ and - ACK signals replace the Master-ready and Slave-ready signals. The target asserts - I/O during an input operation (target to initiator), and it asserts -CID to indicate that the information being transferred is a command or a status response rather than data. High-speed versions of the SCSI bus use a technique known as double-edge clocking or Double Transitions (DT). Each data transfer requires a high-to-low transition followed by a lowto- high transition on the two handshake signals. Double-edge clocking means that data are transferred on both the rising and falling edges of these signals, thus doubling the transfer rate. At the end of the transfer, the target controller releases the - BSY signal, thus freeing the bus for use by other devices.

Reselection: When a logical connection is suspended and the target is ready to restore it, the target must first gain control of the bus. It starts an arbitration cycle, and after winning arbitration, it selects the initiator controller in exactly the same manner as described above. But with the roles of the target and initiator reversed, the initiator is now asserting - BSY. Before data transfer can begin, the initiator must hand control over to the target. This is achieved by having the target

controller assert - BSY r selecting the initiator. Meanwhile, the initiator waits for a short period after being selected to make sure that the target has asserted -BS and then it releases the -BSY line. The connection between the two controllers has now been reestablished, with the target in control of the bus as required for data transfer to proceed.

The bus signaling scheme described above provides the mechanisms needed for two controllers to establish a logical connection and exchange messages. The connection may be suspended and reestablished at any time. The SCSI standard defines the structure and contents of various types of packets that the controllers exchange to handle different situations. The initiator uses these packets to send the commands it receives from the processor to the target. The target responds with status information and data transfer operations. The latter are controlled by the target, because it is the target that knows when data are available, when to suspend and reestablish connections, etc.

Universal Serial Bus (USB):

USB (Universal Serial Bus).: Universal Serial Bus (USB) is an industry standard developed through a collaborative effort of several computer and communications companies, including Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, Nortel Networks, and Philips. USB is a simple and low cost mechanism to connect the devices such as keyboards, mouse, cameras, speakers, printer and display devices to the computer.

The USB supports two speeds of operation, called low-speed (1.5 megabits/s) and fullspeed (12 megabits/s). The most recent revision of the bus specification (USB 2.0) introduced a third speed of operation, called high-speed (480 megabits/s). The USB is quickly gaining acceptance in the market place, and with the addition of the high-speed capability it may well become the interconnection method of choice for most computer devices. The USB has been designed to meet several key objectives:

- Provide a simple, low-cost, and easy to use interconnection system that overcomes the difficulties due to the limited number of I/O ports available on a computer
- Accommodate a wide range of data transfer characteristics for I/O devices, including telephone and Internet connections
- Enhance user convenience through a "plug-and-play" mode of operation

Port limitation: Only a few ports are provided in a typical computer. To add new ports, a user must open the computer box to gain access to the internal expansion bus and install a new interface card. The user may also need to know how to configure the device and the software. An objective of the USB is to make it possible to add many devices to a computer system at any time, without opening the computer box.

Device Characteristics: The different kinds of devices may be connected to a computer cover a wide range of functionality. The speed, volume, and timing constraints associated with data transfers to and from such devices vary significantly. In the case of a keyboard, one byte of data

is generated every time a key is pressed, which may happen at any time. These data should be transferred to the computer promptly. Since the event of pressing a key is not synchronized to any other event in -8 computer system, the data generated by the keyboard are called asynchronous. Furthermore, the rate at which the data are generated is quite low. It is limited by the speed of the human operator to about 100 bytes per second, which is less than 1000 bits per second.

Let us consider a different source of data. Many computers have a microphone either externally attached or built in. The sound picked up by the microphone produces an analog electrical signal, which must be converted into a digital form before it can be handled by the computer. This is accomplished by sampling the analog signal periodically. For each sample, an analog-to-digital (A/D) converter generates an n-bit number representing the magnitude of the sample. The number of bits, n, is selected based on, the desired precision with which to represent each sample. Later, when these data are sent to a speaker, a digital-to-analog (D/A) converter is used to restore the original analog signal from the digital format. The sampling process yields a continuous stream of digitized samples that arrive at regular intervals, synchronized with the sampling clock. Such a data stream is called isochronous, meaning that successive events are separated by equal periods of time.

Plug-and-play: The plug-and-play feature means that a new device, such as an additional speaker, can be connected at any time while the system is operating. The system should detect the existence of this new device automatically, identify the appropriate device-driver soft- ware and any other facilities needed to service that device, and establish the appropriate addresses and logical connections to enable them to communicate. The plug-and-play requirement has many implications at all levels in the system, from the hardware to the operating system and the applications software. One of the primary objectives of the design of the USB has been to provide a plug-and-play capability.

USB Architecture: A serial transmission format has been chosen for the USB because a serial bus satisfies the low-cost and flexibility requirements. Clock and data information are encoded together and transmitted as a single signal. Hence, there are no limitations on clock frequency or distance arising from data skew. Therefore, it is possible to provide a high data transfer bandwidth by using a high clock frequency. As pointed out earlier, the USB offers three bit rates, ranging from 1.5 to 480 megabits/s, to suit the needs of different I/O devices.
To accommodate a large number of devices that can be added or removed at any time, the USB has the tree structure shown in Figure 5.33. Each node of the tree has a device called a hub, which acts as an intermediate control point between the host and the I/O devices. At the root of the tree, a root hub connects the entire tree to the host computer. The leaves of the tree are the I/O devices being served (for example, keyboard, speaker, or digital TV), which are called functions in USB terminology.

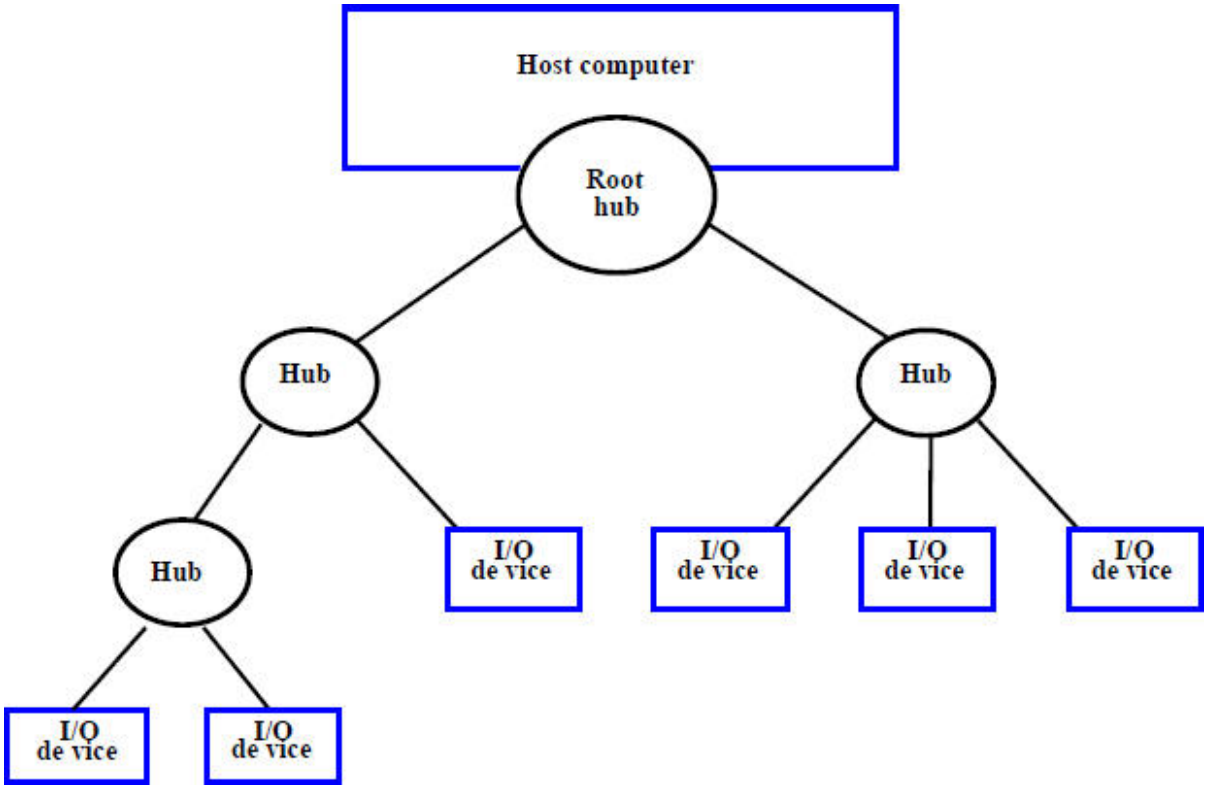


Figure 5.33 Universal Serial Bus tree structure.

The tree structure enables many devices to be connected while using only simple point -topoint serial links. Each hub has a number of ports where devices may be connected, including other hubs. In normal operation, a hub copies a message that it receives from its upstream connection to all its downstream ports. As a result, a message sent by the host computer is broadcast to ill VO devices, but only the addressed device will respond to that message. A message from an I/O device is sent only upstream towards the root of the tree and is not seen by other devices. Hence, the USB enables the host to communicate with the I/O devices, but it does not enable these devices to communicate with each other.

The USB operates strictly on the basis of polling. A device may send a message only in response to a poll message from the host. Hence, upstream messages do not encounter conflicts or interfere with each other, as no two devices can send messages at the same time. This restriction allows hubs to be simple, Low-cost devices.

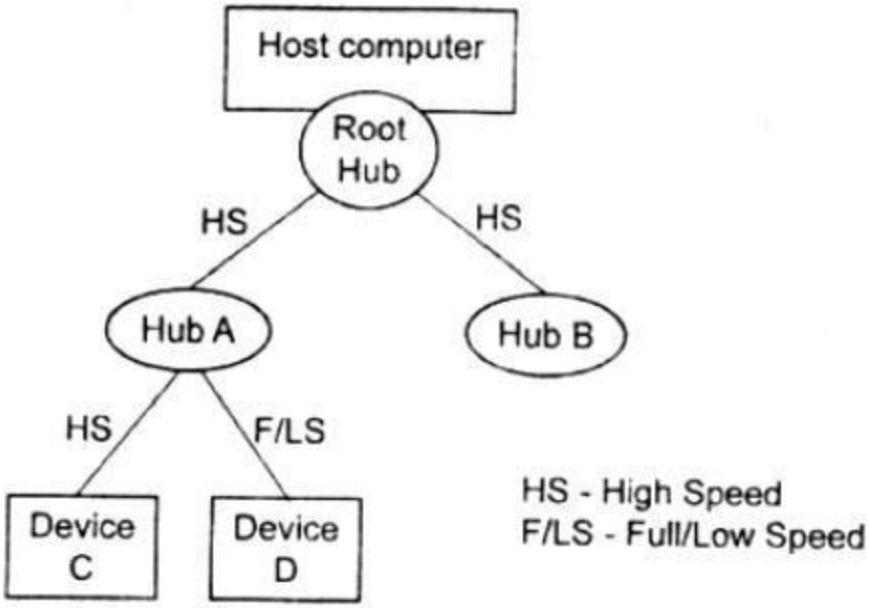


Figure 5.34 USB Split bus operations.

The mode of operation described above is observed for all devices operating at either low speed or full speed. However, one exception has been necessitated by the introduction of highpeed operation in USB version 2.0. Consider the situation in Figure 5.34. Hub A is connected to the root hub by a high-speed link. This hub serves one high-speed device, C, and one low-speed device, D. Normally, a message to device D would be sent at low speed from the root hub. At 1.5 megabits/s, even a short message takes several tens of microseconds. For the duration of this message, no other data transfers can take place, thus reducing the effectiveness of the high-speed links and introducing unacceptable delays for high-speed devices. To mitigate this problem, the USB protocol requires that a message transmitted on a high-speed link is always transmitted at high speed, even when the ultimate receiver is a low-speed device. Hence, a message intended for device D is sent at high speed from the root hub to hub A, then forwarded at low speed to device D. The latter transfer will take a long time, during which high-speed traffic to other nodes is allowed to continue. For example, the root hub may exchange several messages with device C while the low-speed message is being sent from hub A to device D. During this period, the bus is said to be split between high-speed and low-speed traffic. The message to device D is preceded and followed by special commands to hub A to start and end the split-traffic mode of operation, respectively.

The USB standard specifies the hardware details of USB interconnections as well as the organization and requirements of the host software. The purpose of the USB software is to provide bidirectional communication links between application software and I/O devices. These links are called pipes. Any data entering at one end of a pipe is delivered at the other end. Issues such as addressing, timing, or error detection and recovery are handled by the USB protocols. The software that transfers data to or from a given IJO device is called the device driver for that device. The device drivers depend on the characteristics of the devices they support. Hence, a more precise description of the USB pipe is that it connects an VO device to its device driver. It is

established when a device is connected and assigned a unique address by the USB software. Once established, data may flow through the pipe at any time.

Addressing: I/O devices are normally identified by assigning them a unique memory address. In fact, a device usually has several addressable locations to enable the software to send and receive control and status information and to transfer data. When a USB is connected to a host computer, its root hub is attached to the processor bus, where it appears as a single device. The host software communicates with individual devices attached to the USB by sending packets of information, which the root hub forwards to the appropriate device in the USB tree. Each device on the USB, whether it is a hub or an IJO device, is assigned a 7 -bit address. This address is local to the USB tree and is not related in any way to the addresses used on the processor bus. A hub may have any number of devices or other hubs connected to it, and addresses are assigned arbitrarily. When a device is first connected to a hub, or when it is powered on, it has the address 0. The hardware of the hub to which this device is connected is capable of detecting that the device has been connected, and it records this fact as part of its own status information. Periodically, the host polls each hub to collect status information and learn about new devices that may have been added or disconnected.

USB Protocol: All information transferred over the USB is organized in packets, where a packet consists of one or more bytes of information. There are many types of packets that perform a variety of control functions. The information transferred on the USB can be divided into two broad categories: control and data. Control packets perform such tasks as addressing a device to initiate data transfer, acknowledging that data have been received correctly, or indicating an error. Data packets carry information that is delivered to a device. For example, put and output data are transferred inside data packets.

A packet consists of one or more fields containing different kinds of information. The first field of any packet is called the packet identifier, PID, which identifies the type of that packet. There are four bits of information in this field, but they are transmitted twice. The first time they are sent with their true values, and the second time with each bit complemented, as shown in Figure 5.35(a). This enables the receiving device to verify that the PID byte has been received correctly.

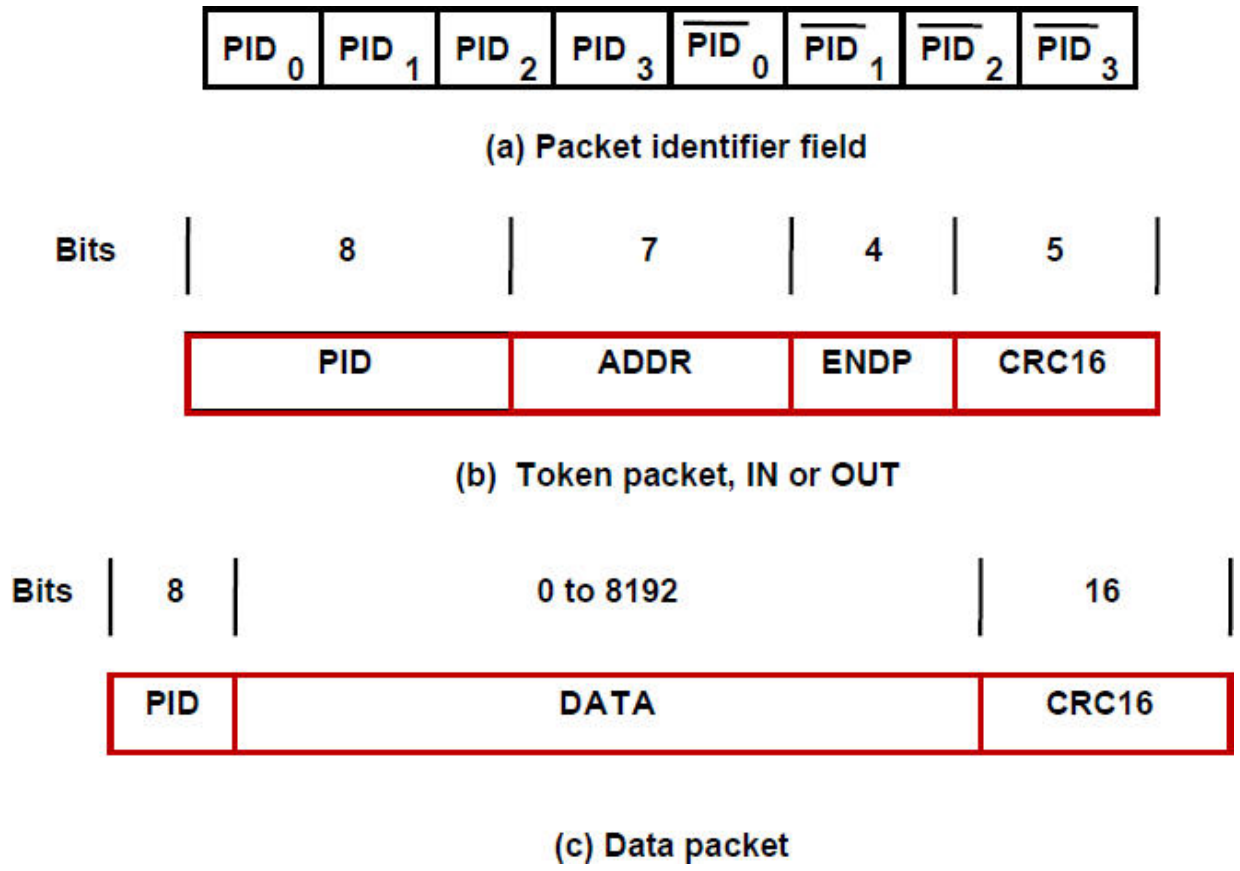


Figure 5.35. USB packet format.

The four PID bits identify one of 16 different packet types. Some control packets, such as ACK (Acknowledge), consist only of the PID byte. Control packets used for controlling data transfer operations are called token packets. They have the format shown in Figure 5.35(b). A token packet starts with the PID field, using one of two PID values to distinguish between an IN packet and an OUT packet, which control input and output transfers, respectively. The PID field is followed by the 7 -bit address of a device and the 4-bit endpoint number within that device. The packet ends with 5 bits for error checking, using a method called cyclic redundancy check (CRC). The CRC bits are computed based on the contents of the address and endpoint fields. By performing an inverse computation, the receiving device can determine whether the packet has been received correctly.

Data packets, which carry input and output data, have the format shown in Figure 4.45c. The packet identifier field is followed by up to 8192 bits of data, then 16 error-checking bits. Three different PID patterns are used to identify data packets, so that data packets may be numbered 0, 1, or 2. Note that data packets do not carry a device address or an endpoint number. This information is included in the IN or OUT token packet that initiates the transfer. Consider an output device connected to a USB hub, which in turn is connected to a host computer. An example of an output operation is shown in Figure 5.36. The host computer sends a token packet of type OUT to the hub, followed by a data packet containing the output data. The PID field of the data packet identifies it as data packet number o. The hub verifies that the transmission has been error free by

checking the error control bits, and then sends an acknowledgment packet (ACK) back to the host. The hub forwards the token and data packets downstream. All I/O devices receive this sequence of packets, but only the device that recognizes its address in the token packet accepts the data in the packet that follows. After verifying that transmission has been error free, it sends an ACK packet to the hub.

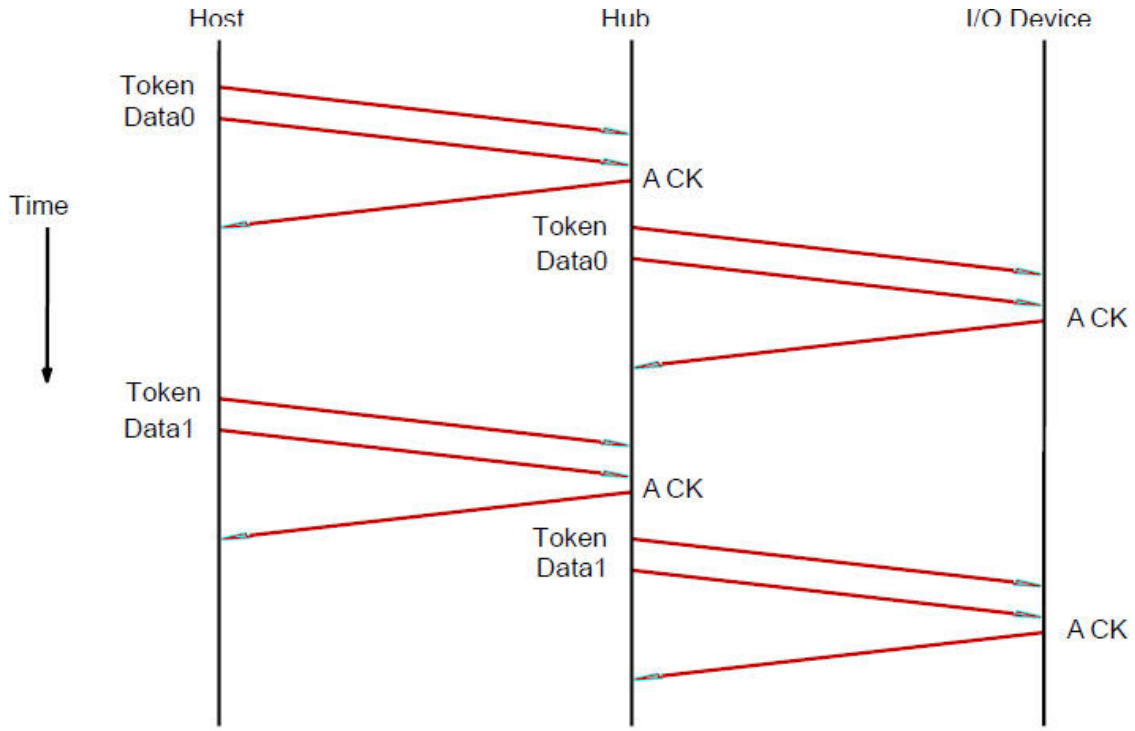


Figure 5.36 An output transfer

Successive data packets on a full-speed or low-speed pipe carry the numbers 0 and 1, alternately. This simplifies recovery from transmission errors. If a token, data, or acknowledgment packet is lost as a result of a transmission error, the sender resends the entire sequence. By checking the data packet number in the PID field, the receiver can detect and discard duplicate packets. High-speed data packets are sequentially numbered 0, 1, 2, 0, and so on. Input operations follow a similar procedure. The host sends a token packet of type IN containing the device address. In effect, this packet is a poll asking the device to send any input data it may have. The device responds by sending a data packet followed by an ACK. If it has no data ready, it responds by sending a negative acknowledgment (NAK) instead.

Electrical characteristics: The cables used for USB connections consist of four wires. Two are used to carry power, 5 V and Ground. Thus, a hub or an I/O device may be powered directly from the bus, or it may have its own external power connection. The other two wires are used to carry data. Different signaling schemes are used for different speeds of transmission. At low speed, 1s and 0s are transmitted by sending a high voltage state (5 V) on one or the other of the two signal wires. For high-speed links, differential transmission is used.