KONGUNADU COLLEGE OF ENGINEERING AND TECHNOLOGY (AUTONOMOUS)

DEPATMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT CODE:24EC304

SUBJECT NAME:DIGITAL LOGIC AND COMPUTER ORGANIZATION

UNIT-I
DIGITAL FUNDAMENTALS

- Number Systems – Decimal, Binary, Octal, Hexadecimal, radix conversion ,1's and 2's complements, Codes – Binary, BCD, Excess 3, Gray, Alphanumeric codes, Boolean theorems & Postulates, Logic gates, Universal gates, Sum of products and product of sums, Minterms and Maxterms, Karnaugh map Minimization

UNIT-I
BOOLEAN ALGEBRA AND LOGIC GATES

Number Systems:

Number system is a basis for counting various items

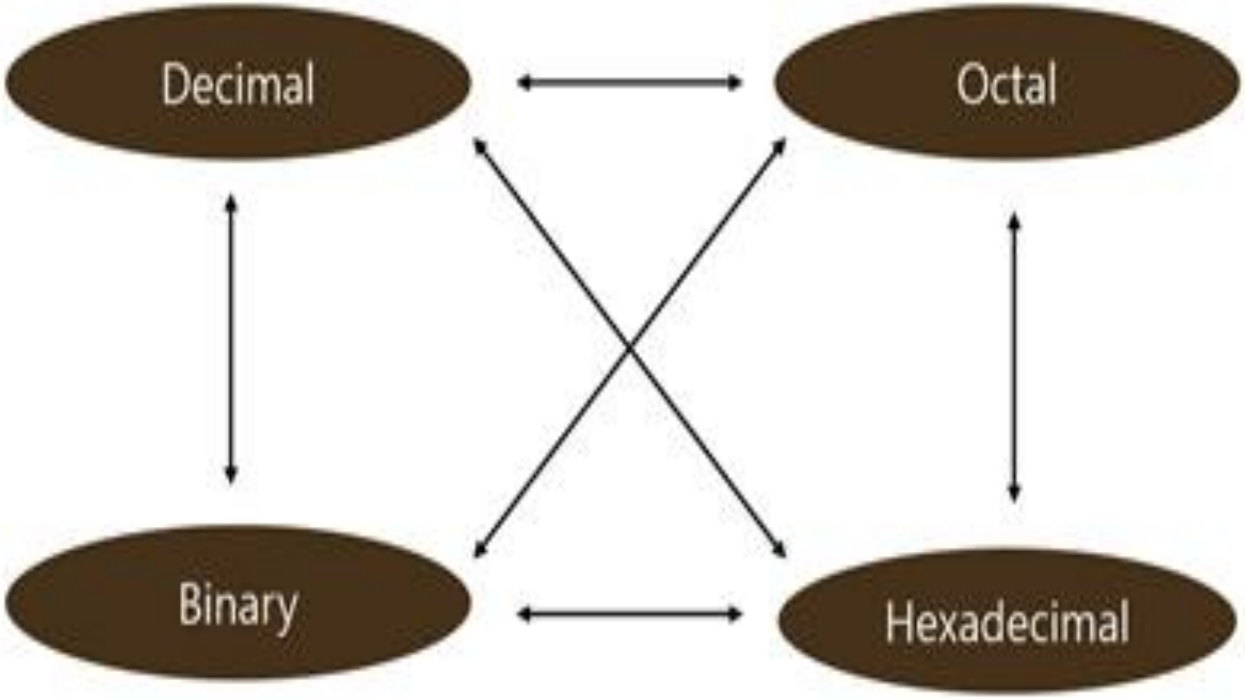
The decimal number system has 10 digits:0,1,2,3,4,5,6,7,8 and 9

Types of Number Systems:

System	Base	Symbols
Decimal	10	0,1,2,….9
Binary	2	0,1
Octal	8	0,1,2,….7
Hexa-decimal	16	0,1,2,….9,A,B,…F

Decimal	Hexadecimal	Binary (421/8421)
0	0	(0)000
1	1	(0)001
2	2	(0)010
3	3	(0)011
4	4	(0)100
5	5	(0)101
6	6	(0)110
7	7	(0)111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Conversion among Bases



Binary to Octal

- Group into 3's starting at least significant bit (if the number of bits is not evenly divisible by 3, then add 0's at the most significant end)
- write 1 octal digit for each group
- e.g.:$(001010101)_2$ to $()_8$

001 010 101

4 2 1

1 2 5

1 0 0

Answer = 125_8

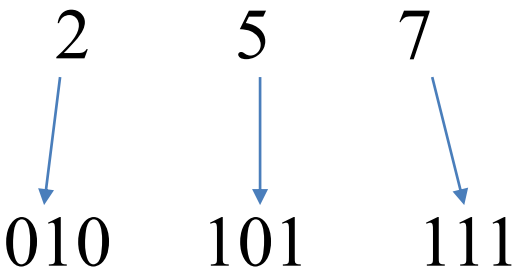
e.g.:$(010101101.011100)_2$ to $()_8$

(0)10 101 101 . 011 1(00)

2 5 5 . 3 4 → $(255.34)_8$

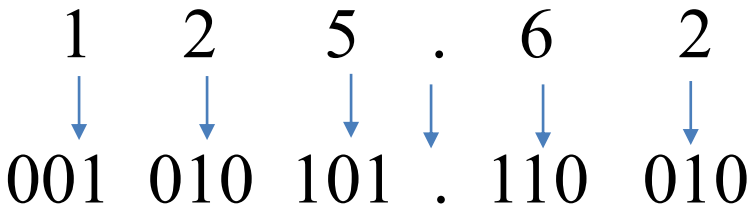
Octal to Binary

- For each of the Octal digit write its binary equivalent
  e.g.: $(257)_8$ to $(\ )_2$



Answer = $(010101111)_2$

e.g.: $(125.62)_8$ to $(\ )_2$



Binary to Hexadecimal

- Group into 4's starting at least significant bit (if the number of bits is not evenly divisible by 4, then add 0's at the most significant end)
- write 1 hex digit for each group.
- e.g.: $(001010111011)_2$ to $()_{16}$

(00)10 1011 1011

2 B B

Answer = $(2BB)_{16}$

e.g.: $(001101101110.10011010)_2$ to $()_{16}$

(00)11 0110 1110 . 1001 101(0)

3 6 E . 9 A → $(36E.9A)_{16}$

Hexadecimal to Binary

- For each of the Hex digit write its binary equivalent(use 4 digits to represent)
- e.g.: $(8A9.B4)_{16}$ to $(\ )_2$

8	A	9	.	B	4
1000	1010	1001	.	1011	0100

→ $(100010101001.101101)_2$

Octal to Hexadecimal

Steps:

1.Convert octal number to its binary equivalent

2.Convert binary number to its hexadecimal equivalent

e.g.: $(615.25)_8$ to $()_{16}$

6	1	5	.	2	5
110	001	101	.	010	101
1	1000	1101	.0101	01	
(000)1	1000	1101	.	0101	01(00)
1	8	D	.	5	4

Step 1

Step 2

$(615.25)_8 \rightarrow (110001101.010101)_2 \rightarrow (18D.54)_{16}$

Hexadecimal to Octal

Steps:

1. Convert hexadecimal number to its binary equivalent
2. Convert binary number to its octal equivalent

e.g.: $(BC66.AF)_{16}$ to $()_8$

B	C	6	6	.	A	F
1011	1100	0110	0110	.	1010	1111
(00)1011110001100110.10101111(0)						
001	011	110	001	100	110	.	101	011	110
1	3	6	1	4	6	.	5	3	6

Step 1

Step 2

$(BC66.AF)_{16} \longrightarrow (1011110001100110)_2 \longrightarrow (136146.536)_8$

Converting any radix to decimal

- Converting from any base to decimal is done by multiplying each digit by its weight and summing.
- Ex: Convert $(3102.12)_4$ to its decimal equivalent

$N=3*4^3+1*4^2+0*4^1+2*4^0+1*4^{-1}+2*4^{-2}$

$=192+16+0+2+0.25+0.125=(210.375)_{10}$

- Ex: Determine the value of base x, if $(193)_x = (623)_8$

Converting octal into decimal : $(623)_8= 6*8^2+2*8^1+3*8^0 =(403)_{10}$

$(193)_x=1*x^2+9*x^1+3*x^0 = (403)_{10}$

$x^2+9x+3=403$ ⟶ $x=16$ or -25

Negative not applicable so $(193)_{16}=(623)_8$

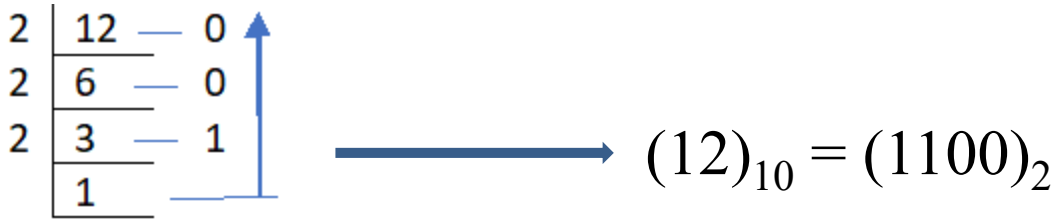
Conversion of Decimal number to any Radix number

Steps:

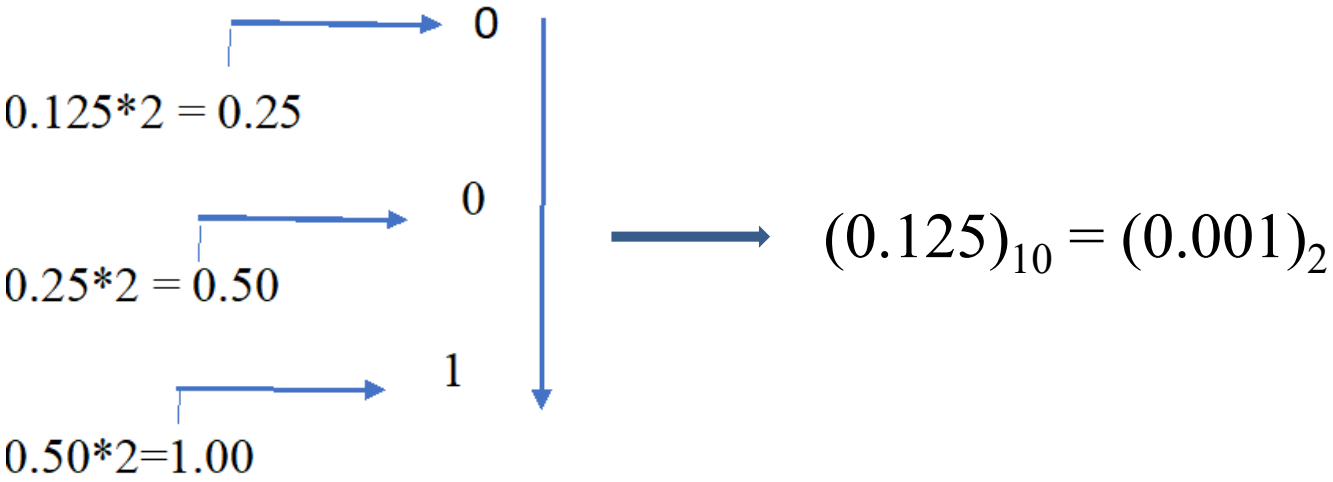
1. Convert integer part (Successive Division Method)
2. Convert fractional part (Successive Multiplication Method)

- Steps in Successive Division Method:
  - Divide the integer part of decimal number by desired base number, store quotient (Q) and remainder (R)
  - Consider quotient as a new decimal number and repeat step1 until quotient becomes 0
  - List the remainders in the reverse order
- Steps in Successive Multiplication Method:
  - Multiply the fractional part of decimal number by desired base number
  - Record the integer part of product as carry and fractional part as new fractional part
  - Repeat steps 1 and 2 until fractional part of product becomes 0 or until you have many digits as necessary for your application
  - Read carries downwards to get desired base number

- Convert 12.125 decimal to binary
- Integer Part:

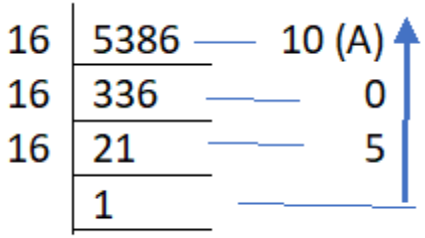


- Fractional Part:



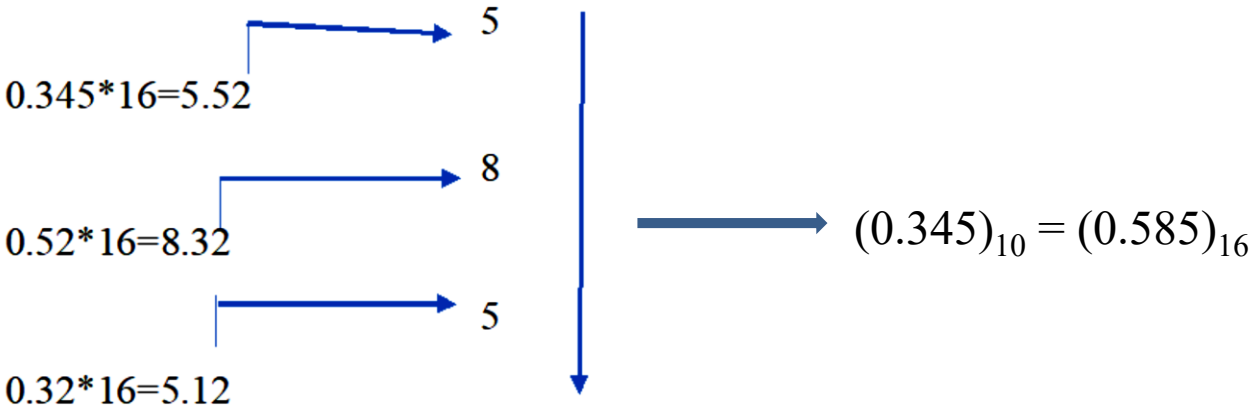
$(12.125)_{10} = (1100.001)_2$

- Convert 5386.345 decimal to hexadecimal
- Integer Part:



$$(5386)_{10} = (150A)_{16}$$

- Fractional Part:



$$(0.345)_{10} = (0.585)_{16}$$

$$(5386.345)_{10} = (150A.585)_{16}$$

1’s Complement

The 1’s complement of a binary number is the number that results when we change all 1’s to zeros and the zeros to ones.

1	1	0	1	0	1	0	0	Number
NOT operation								
0	0	1	0	1	0	1	1	1’s Complement

2’s Complement

The 2’s complement the binary number that results when add 1 to the 1’s complement.

2’s complement = 1’s complement + 1

1	1	0	0	0	1	0	0	Number
NOT operation								
					1	1		Carry
0	0	1	1	1	0	1	1	1’s Complement
							1	Add 1
0	0	1	1	1	1	0	0	2’s Complement

- 9’s Complement

The nines' complement of a decimal digit is the number that must be added to it to produce 9. The complement of 3 is 6, the complement of 7 is 2.

- Example: Obtain 9’s complement of 7493

```
  9 9 9 9
- 7 4 9 3
---------
  2 5 0 6   →9’s complement
```

10’s Complement

The 10’s complement of the given number is obtained by adding 1 to the 9’s complement.

10’s complement = 9’s complement + 1

Example: Obtain 10’s complement of 7493

```
  9 9 9 9          2 5 0 6
- 7 4 9 3            + 1
--------          ---------
  2 5 0 6          2 5 0 7   ⟶   10’s complement
```

Arithmetic Operations

Binary Addition

- The addition consists of four possible elementary operations:

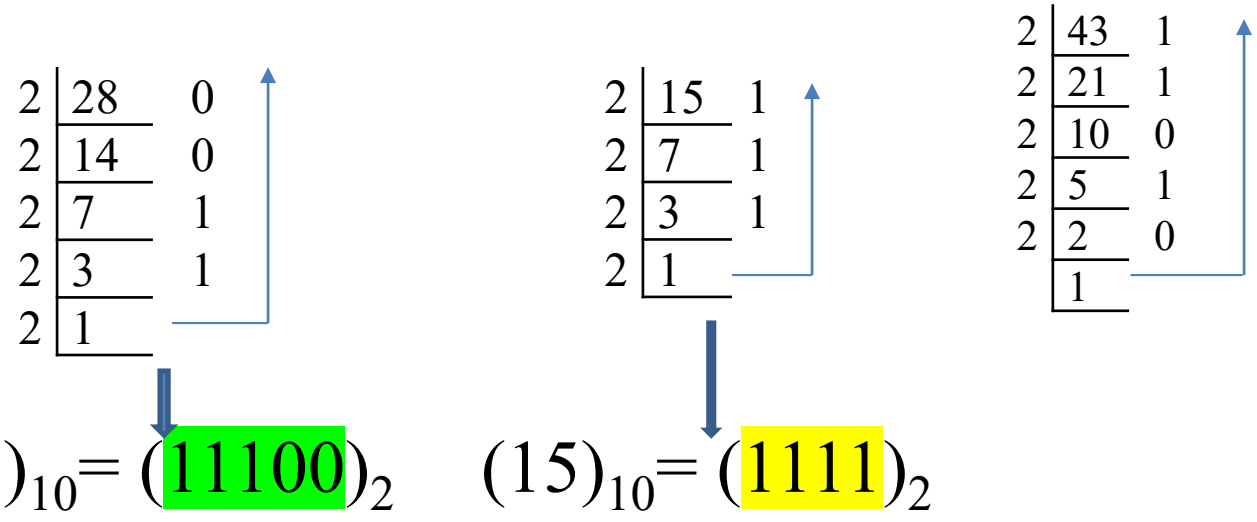
S.No	Operations
1	0+0=0
2	0+1=1
3	1+0=1
4	1+1=10(0 with carry 1)

In the last case, sum is of two digits: Higher Significant bit is called Carry and lower significant bit is called Sum.

- Perform addition of $(11001100)_2$ and $(11011010)_2$

1	1		1	1				Carry
	1	1	0	0	1	1	0	0 Number 1
(+)	1	1	0	1	1	0	1	0 Number 2
1	1	0	1	0	0	1	1	0 Result

- Add $(28)_{10}$ and $(15)_{10}$ by converting them to binary



$(28)_{10} = (11100)_2$ $(15)_{10} = (1111)_2$

1	1	1				Carry
	1	1	1	0	0	$(28)_{10}$
(+)	0	1	1	1	1	$(15)_{10}$
1	0	1	0	1	1	$(43)_{10}$

Binary Subtraction

- The subtraction consists of four possible elementary operations:

S.No	Operations
1	0-0=0
2	0-1=1 (borrow 1)
3	1-0=1
4	1-1=0

In case of second operation the minuend bit is smaller than the subtrahend bit, hence 1 is borrowed.

- Perform $(11101100)_2 - (00110010)_2$

			(10) 0	10		0	10		
	1	1	1	0	1	1	0	0	Number 1
(-)	0	0	1	1	0	0	1	0	Number 2
	1	0	1	1	1	0	1	0	Result

Binary Subtraction using 1's complement

- Perform subtraction using 1's complement $(11010)_2$ - $(10000)_2$

Step 1: 1's complement negative number

$(10000)_2$ ⟶ $(01111)_2$

Step 2: Add $(11010)_2$ and $(01111)_2$

1	1	1	1		
	1	1	0	1	0
	0	1	1	1	1
1	0	1	0	0	1
					1
	0	1	0	1	0

Add end-around carry

Note: If carry is generated then the result is positive and in the true form so aa carry to the result to get final result

- Perform subtraction using 1's complement $(15)_{10}$ -$(28)_{10}$

Binary equivalent:$(1111)_2$ -$(11100)_2$

Step 1: 1's complement negative number

$(11100)_2$ ⟶ $(00011)_2$

Step 2: Add $(1111)_2$ and $(00011)_2$

	1	1	1	1		
	0	1	1	1	1	$(15)_{10}$
(+)	0	0	0	1	1	1,s complement of $(28)_{10}$
	1	0	0	1	0	Result
In this case the carry is not generated then the result is negative and in the 1's complement form						
	0	1	1	0	1	Verification (1's complement form of result) $(13)_{10}$

Binary Subtraction using 2's complement

- Perform subtraction using 2's complement binary arithmetic $(147)_{10}$ $-(89)_{10}$
- Step 1: Binary equivalent $(010010011)_2$ $-(01011001)_2$
- Step 2: Find 2's complement of $(89)_{10}$

0	1	0	1	1	0	0	1	Binary equivalent of $(89)_{10}$
1	0	1	0	0	1	1	0	1's complement of $(89)_{10}$
							1	Add 1
1	0	1	0	0	1	1	1	2's complement of $(89)_{10}$

1	1					1	1	1	
	0	1	0	0	1	0	0	1	1	Binary equivalent of $(147)_{10}$
(+)	1	1	0	1	0	0	1	1	1	2's complement of $(89)_{10}$
1	0	0	0	1	1	1	0	1	0	Result

Note: If carry is generated then the result is positive and in the true form so the carry is ignored.

Binary Subtraction using 2's complement

- Perform subtraction using 2's complement $(42)_{10}$ -$(68)_{10}$
- Step 1: Binary equivalent $(101010)_2$ -$(1000100)_2$
- Step 2: Find 2's complement of $(68)_{10}$

1	0	0	0	1	0	0	Binary equivalent of $(68)_{10}$
0	1	1	1	0	1	1	1's complement of $(68)_{10}$
						1	Add 1
0	1	1	1	1	0	0	2's complement of $(68)_{10}$

		1	1				
	0	1	0	1	0	1	0	Binary equivalent of $(42)_{10}$
(+)	0	1	1	1	1	0	0	2's complement of $(68)_{10}$
	1	1	0	0	1	1	0	Result

Note: If carry is not generated then the result is negative and in the 2's complement form

1	1	0	0	1	1	0	Result
0	0	1	1	0	0	1	1's complement
						1	2's complement
0	0	1	1	0	1	0	$(26)_{10}$

Binary Multiplication

Rules for Binary Multiplication are:

S.No	Operations
1	0*0=0
2	0*1=0
3	1*0=0
4	1*1=1

Multiply $(101.11)_2$ and $(110.01)_2$ using binary multiplication method

				1	0	1	.	1	1	Multiplicand
			x	1	1	0	.	0	1	Multiplier
					1	0	1	1	1	
				0	0	0	0	0	0	
			0	0	0	0	0	0	0	
		1	0	1	1	1	0	0	0	
	1	0	1	1	1	0	0	0	0	
1	0	0	0	1	1	1	1	1	1	Final

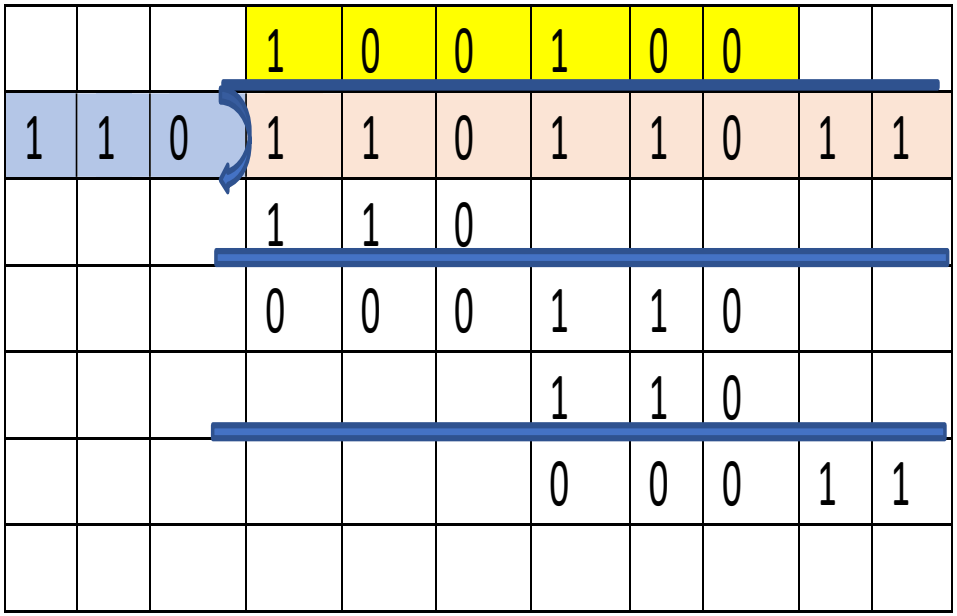
Fractional digits in the final product=Fractional digits in multiplicand +Fractional digits in multiplier =2+2=4 → $(101.11)_2$ x $(110.01)_2$ = $(100011.1111)_2$

Binary Division

Rules for Binary Division are:

No.	Rule
1	$0 \div 1 = 0$
2	$1 \div 1 = 1$

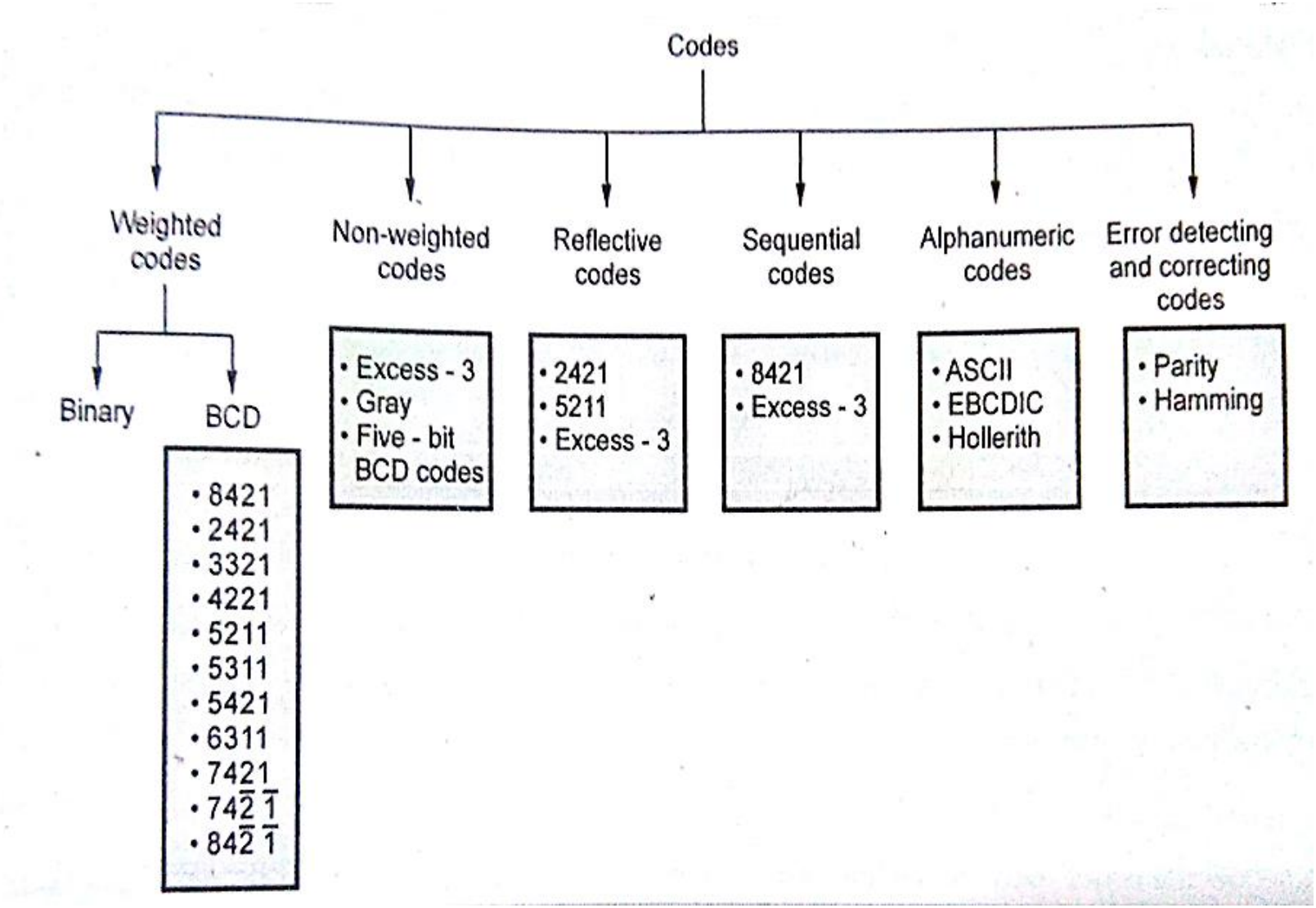
Divide $(11011011)_2$ by $(110)_2$



Binary Codes

- When numbers, alphabets or words are represented by a specific group of symbols i.e., they are encoded
- The group of symbols used to encode them is called codes. The digital data is represented, stored and transmitted as groups of binary digits (bits)
- Group of bits--- binary code---- numeric and alphanumeric code

Classification of binary codes



- **Weighted codes:**
  - In weighted codes, each digit position of the number represents a specific weight
  - Examples: 93→(1001)(0011) → 8421 code

    93→(1100)(0011) →5421 code
- **Non-weighted codes :**
  - Non-weighted codes are not assigned with any weight to each digit position, i.e., each digit position within the number is not assigned fixed value
  - Excess-3 and gray codes are the non-weighted codes
- **Reflective codes:**
  - A code is said to be reflective when the code for 9 is the complement for 0, the code for 8 is complement for 1, 7 for 2,6 for 3 and 5 for 4.
  - Like 2421,codes 5211 and excess-3 are also reflective.
  - The 8421 code is not reflective

- **Sequential codes**
  - In sequential codes each succeeding code is one binary number greater than its preceding code.
  - This greatly aids mathematical manipulation of data
  - The 8421 and excess-3 are sequential, whereas the 2421 and 5211 codes are not sequential

- **Alphanumeric codes**
  - The codes which consists of both numbers and alphabetic characters are called alphanumeric codes.
  - Most of these codes, however, also represent symbols and various instructions necessary for conveying intelligible information
  - The most commonly used alphanumeric codes are: ASCII, EBCDIC and Hollerith code

- **Error detecting and correcting codes**
  - When the digital information in the binary form is transmitted from one circuit or system to another circuit or system an error may occur.
  - This means the signal corresponding to 0 may change to 1 or vice-versa due to presence of noise
  - To maintain data integrity between transmitter and receiver, extra bit or more than one bit are added in the data.
  - These extra bits allow the detection and sometimes the correction of error in the data.
  - The data along with the extra bit /bits form the code
  - Codes which allow only error detection are called error detecting codes and codes which allow error detection and correction are called error detecting and correcting codes

- **BCD(Binary Coded Decimal) codes**
  - BCD is a numeric code in which each digit of a decimal number is represented by a separate group of 4-bits.

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

5	8	Decimal
0101	1000	BCD Code

Advantages:

- Easy to convert between it and decimal

Disadvantages:

- Less efficient
- Arithmetic operations are more complex

- Excess-3 code
  - The excess-3 code can be derived from the natural BCD code by adding 3 to each coded number.
  - It is a non-weighted code
  - It is a sequential code
  - In excess-3 code we get 9's complement of a number by just complementing each bit. Due to this excess-3 code is called self-complementing code or reflective code.

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

- **Gary code**
  - Gray code is a non-weighted code and is a special case of unit-distance code
  - In unit distance code, bit patterns for two consecutive numbers differ in only one bit position.
  - These codes are also called as cyclic codes.
  - The gray code is also called reflected code.

Application of Gray code

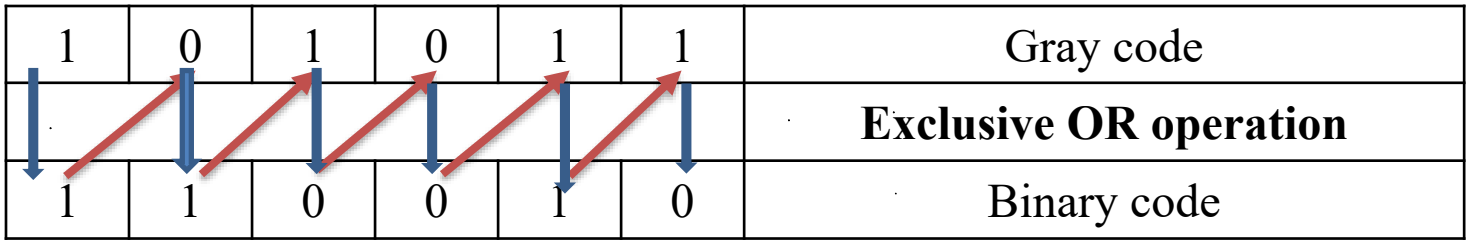
- Let us consider an application where 3-bit binary code is provided to indicate position of the rotating disk with the help of brushes.
- If one brush is slightly ahead of the other, an 180° error occur in the disk position.
- When the gray code is used to represent disk position then error due to improper brush alignment can be reduced. This is because the gray code assures that only one bit will change each time the decimal number is incremented.
- In 3-bit code probability of error is reduced upto 66% and in 4-bit code it is reduced upto 75%. This is an advantage of gray code.

Gray to Binary Conversion:

Exclusive OR operation

A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

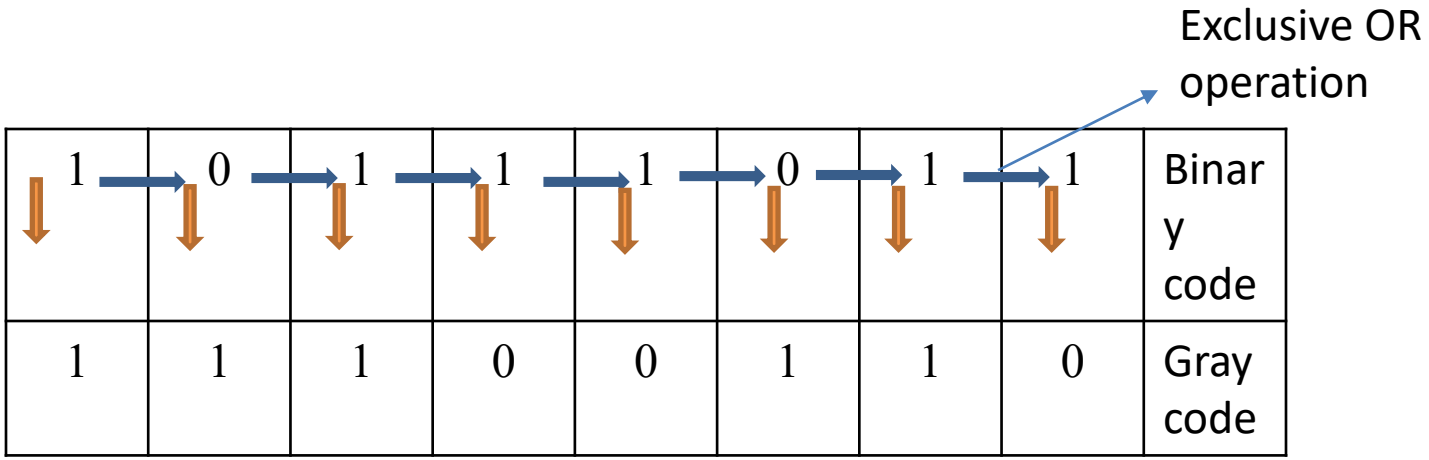
Convert gray code 101011 into its binary equivalent.



$$(101011)_{gray} = (110010)_2$$

Binary to Gray code:

Convert 10111011 in binary into its equivalent gray code.



BCD Addition

- **Case 1: Sum equals 9 or less with carry 0**

Addition of 3 and 6 in BCD

	1	1			Carry
	0	1	1	0	BCD for 6
+	0	0	1	1	BCD for 3
	1	0	0	1	BCD for 9

- **Case 2: Sum greater than 9 with carry 0**

	0	1	1	0	BCD for 6
+	1	0	0	0	BCD for 8
	1	1	1	0	Invalid BCD number (14)
	1	1	1	0	Invalid BCD number (14)
	0	1	1	0	Add 6 for correction
1	0	1	0	0	Answer

0	0	0	1	0	1	0	0	BCD for 14

- **Case 3: Sum greater than 9 with carry 0**

					Carry
	1	0	0	0	BCD for 8
+	1	0	0	1	BCD for 9
1	0	0	0	1	Incorrect BCD number

0	0	0	1	0	0	0	1	Incorrect BCD number
0	0	0	0	0	1	1	0	Add 6 for correction
0	0	0	1	0	1	1	1	Answer

0	0	0	1	0	1	1	1	BCD for 17

BCD Subtraction using 9's complement

Perform $(46)_{10} - (22)_{10}$ in BCD using 9's complement.

Step 1: Find 9's complement of 22

9's complement of 22=(99-22)=77

Step 2:Add 46 and 9's complement of 22

	1				1	1			Carry
	0	1	0	0	0	1	1	0	BCD of 46
+	0	1	1	1	0	1	1	1	BCD of 77
	1	0	1	1	1	1	0	1	Invalid BCD numbers
	1	0	1	1	1	1	0	1	Invalid BCD numbers
+	0	1	1	0	0	1	1	0	Add 6 in each digit
1	0	0	1	0	0	0	1	1	
								1	Add end around carry
	0	0	1	0	0	1	0	0	Result (BCD of 24)
Since there is carry the result is positive and true									

BCD Subtraction using 9's complement

Perform $(24)_{10} - (56)_{10}$ in BCD using 9's complement.

Step 1: Find 9's complement of 56

9's complement of 56=(99-56)=43

Step 2:Add 24 and 9's complement of 56

									Carry
	0	0	1	0	0	1	0	0	BCD of 24
+	0	1	0	0	0	0	1	1	BCD of 43
	0	1	1	0	0	1	1	1	BCD of 67
Since there is 0 the result is negative									

Step 3: Take 9's complement of answer

99-67=32➔24-56=32

BCD Subtraction using 10's complement

Perform $(46)_{10} - (22)_{10}$ in BCD using 9's complement.

Step 1: Find 10's complement of 22

10's complement of 22=9's complement of 22+1 =(99-22)+1=78

Step 2:Add 46 and 10's complement of 22

	1								Carry
	0	1	0	0	0	1	1	0	BCD of 46
+	0	1	1	1	1	0	0	0	BCD of 78
	1	0	1	1	1	1	1	0	Invalid BCD numbers
	1	0	1	1	1	1	1	0	Invalid BCD numbers
+	0	1	1	0	0	1	1	0	Add 6 in each digit
1	0	0	1	0	0	1	0	0	
Carry is ignored									
	0	0	1	0	0	1	0	0	Result (BCD of 24)
Since there is carry the result is positive and true									

BCD Subtraction using 10's complement

Perform $(24)_{10} - (56)_{10}$ in BCD using 10's complement.

Step 1: Find 10's complement of 56

9's complement of 56=(99-56)+1=44

Step 2:Add 24 and 10's complement of 56

									Carry
	0	0	1	0	0	1	0	0	BCD of 24
+	0	1	0	0	0	1	0	0	BCD of 44
	0	1	1	0	1	0	0	0	BCD of 68
Since there is 0 the result is negative									

Step 3: Take 10's complement of answer

(99-68)+1=32➔24-56=32

Excess-3 Addition:

a)Carry is generated:8+6

		1	0	1	1	Excess-3 for 8
	+	1	0	0	1	Excess-3 for 6
	1	0	1	0	0	Carry is 1

0	0	0	1	0	1	0	0	
0	0	1	1	0	0	1	1	Add 3
0	1	0	0	0	1	1	1	Result (Excess-3 for 14)
1				4				Result in decimal

Excess-3 Addition:

a)Carry is not generated:1+2

		0	1	0	0	Excess-3 for 1
	+	0	1	0	1	Excess-3 for 2
	0	1	0	0	1	Carry is 0

0	1	0	0	1	
	0	0	1	1	Sub 3
	0	1	1	0	Result (Excess-3 for 3)
3					Result in decimal

Excess-3 Subtraction:

a)Carry is generated:8-5

1	0	0	0	Excess-3 for 5
0	1	1	1	Complement

		1	0	1	1	Excess-3 for 8
	+	0	1	1	1	Complement of 5 in Excess-3
	1	0	0	1	0	Carry is 1

1	0	0	1	0	
	0	0	1	1	Add 3
1	0	1	0	1	Result (Excess-3 for 3)
				1	Add end-around carry
	0	1	1	0	Excess-3 for 3

Excess-3 Subtraction:

a)Carry is not generated:5-8

1	0	1	1	Excess-3 for 8
0	1	0	0	Complement

		1	0	0	0	Excess-3 for 5
	+	0	1	0	0	Complement of 8 in Excess-3
	0	1	1	0	0	Carry is 0

0	1	1	0	0	
	0	0	1	1	Sub 3
0	1	0	0	1	Result (Excess-3 for 3)

Boolean Algebra

- Boolean algebra is a mathematical system that defines a series of logical operations (AND,OR,NOT) performed on sets of variables (a,b,c,….).
- When stated in this form, the expression is called a Boolean equation or switching equation.

Terminologies :

Variable: The symbol which represent an arbitrary elements of an Boolean algebra is known as variable.

Any single variable or a function of several variables can have either a 1 or 0 value.

Constant: In expression Y=A+1, the first term A is a variable and the second term has a fixed value 1. So 1 is a constant here. The constant may be 1 or 0.

Complement: A complement of a variable is presented by a "bar" over the letter and sometimes denoted by (`).

Example: $\overline{A}$ is the complement of the variable A, if A=0 $\rightarrow \overline{A}$ =1 and A=1$\rightarrow \overline{A}$ =0

Literal: Each occurrence of a variable in Boolean function either in a complemented or an uncomplemented form is called a literal.

Boolean Function: Boolean expressions are constructed by connecting the Boolean constants and variables with the Boolean operations. These Boolean expressions are also known as Boolean formulae.

We use Boolean expressions to describe the Boolean functions.

Example: f(A,B,C)= (A+ $\overline{B}$)C

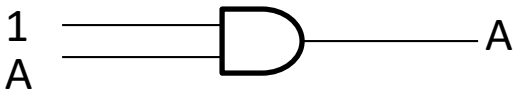
Properties of Boolean Algebra

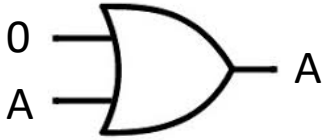
- **Closure Property**

Closure(a): Closure with respect to operator +: when two binary elements are operated by operator +, the result is a unique binary element.

Closure(b):Closure with respect to operator .(dot): when two binary elements are operated by operator .(dot), the result is a unique binary element.

- **Identity property:** A.1=1.A=A





- **Commutative property**

Commutative with respect to +: A+B=B+A

Commutative with respect to . : A.B=B.A

- **Distributive property**

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

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

- **Associative property**

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

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

- **Complement property**

$$A.\overline{A}=0 \quad A+\overline{A}=1$$

- **Idempotency property**

$$A.A=A \qquad A+A=A$$

- **Absorption property**

$$A+AB=A(1+B)=A$$

$$A(A+B)=A+AB=A$$

- **Involution property**

$$\overline{\overline{A}}=A$$

De Morgan's Theorem

Theorom:1

the complement of the product of all the terms is equal to the sum of the complement of each term.

Theorom:2

- the complement of the sum of all the terms is equal to the product of the complement of each term.

- **De-Morgan's theorem**

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

A	B	$\overline{A+B}$	$\overline{A}.\overline{B}$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

- **Principle of duality:**
- The principle of duality theorem says that, starting with a Boolean relation, we can derive another Boolean relation by,
- Changing the OR sign to an AND sign
- Changing each AND sign to an OR sign and
- Complementing any 0 or 1 appearing in the expression.
- Dual of relation → A+$\overline{A}$=1 is A.$\overline{A}$=0

- **Consensus Law:**
- In simplification of Boolean expression, an expression of the form AB+$\overline{A}$C+BC the term BC is redundant and can be eliminated to form the equivalent expression AB+$\overline{A}$C. The theorem used for this simplification is known as consensus theorem and it is stated as

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

Proof:

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

$$= AB+\overline{A}C+A\,BC+\overline{A}\,BC$$

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

$$=AB+\overline{A}C$$

- **Prove the following Boolean identities**

$$(x_1+x_2)(\overline{x_1}\,\overline{x_3}+x_3)\,\overline{(\overline{x_2}+x_1x_3)}=\overline{x_1}x_2$$

$$(x_1+x_2)(\overline{x_1}\,\overline{x_3}+x_3)\,\overline{(\overline{x_2}+x_1x_3)}$$

$$=(x_1+x_2)(\overline{x_1}\,\overline{x_3}+x_3)\,\overline{(\overline{x_2}\,.\,x_1x_3)}$$

$$=(x_1+x_2)(\overline{x_1}\,\overline{x_3}+x_3)\,(x_2\,.\,(\overline{x_1}+\overline{x_3}))$$

$$=(x_1+x_2)(\overline{x_1}\,\overline{x_3}+x_3)\,(x_2\,\overline{x_1}+x_2\overline{x_3})$$

$$=(x_1+x_2)(\overline{x_1}+x_3)\,(x_2\,\overline{x_1}+x_2\overline{x_3}) \quad \text{(since A+}\overline{A}B=A+B\text{)}$$

$$=(x_1\overline{x_1}+x_2\overline{x_1}+x_1x_3+x_2x_3)\,(x_2\,\overline{x_1}+x_2\overline{x_3})$$

$$=(0+x_2\overline{x_1}+x_1x_3+x_2x_3)\,(x_2\,\overline{x_1}+x_2\overline{x_3}) \quad \text{(since A}\overline{A}=0\text{)}$$

$$=x_2\overline{x_1}x_2\,\overline{x_1}+x_1x_3x_2\,\overline{x_1}+x_2x_3x_2\,\overline{x_1}+x_2\overline{x_1}x_2\overline{x_3}$$

$$+x_1x_3x_2\overline{x_3}+x_2x_3x_2\overline{x_3}$$

$$=x_2\overline{x_1}x_2\,\overline{x_1}+0+x_2x_3x_2\,\overline{x_1}+x_2\overline{x_1}x_2\overline{x_3}+0+0$$

$$=x_2\,\overline{x_1}+x_2x_3\overline{x_1}+\overline{x_1}x_2\overline{x_3}$$

$$=x_2\,\overline{x_1}(1+x_3+\overline{x_3}) \quad \text{(Since 1+A=1)}=x_2\,\overline{x_1}$$

- Prove the following using DeMorgan's theorem

$$[(x+y)' + (x+y)']' = x + y$$

$$= (\overline{\overline{(x+y)}} + \overline{\overline{(x+y)}}$$

$$= \overline{\overline{(x+y)}}.\overline{\overline{(x+y)}}$$

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

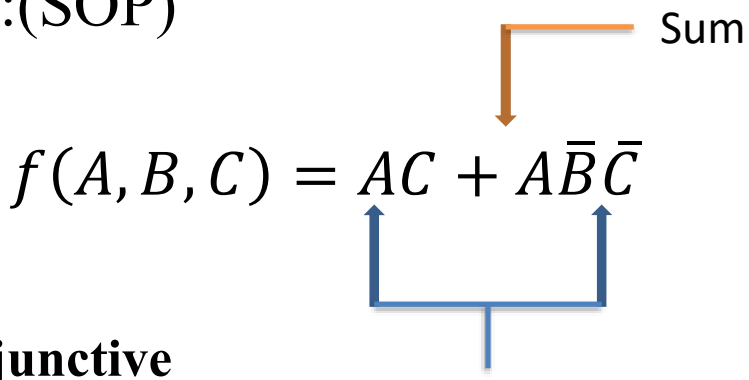
$=(x+y)$ (since A.A=A)

Boolean expression

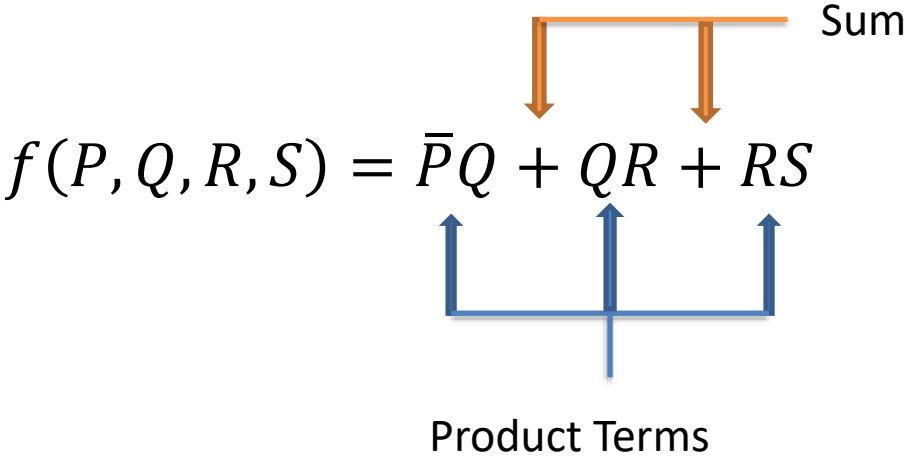
- Boolean expressions are constructed by connecting the Boolean constants and variables with the Boolean operations.

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

Sum of Product form:(SOP)

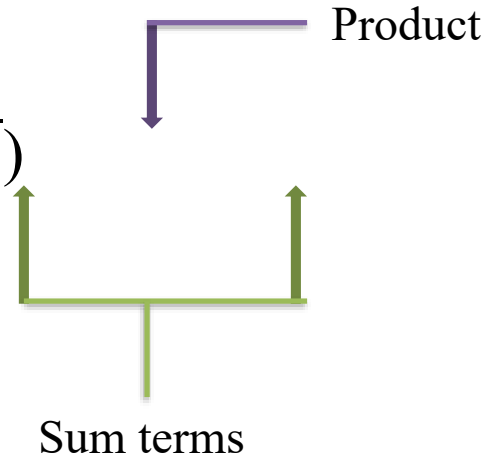


Also known as disjunctive normal form or disjunctive normal formula



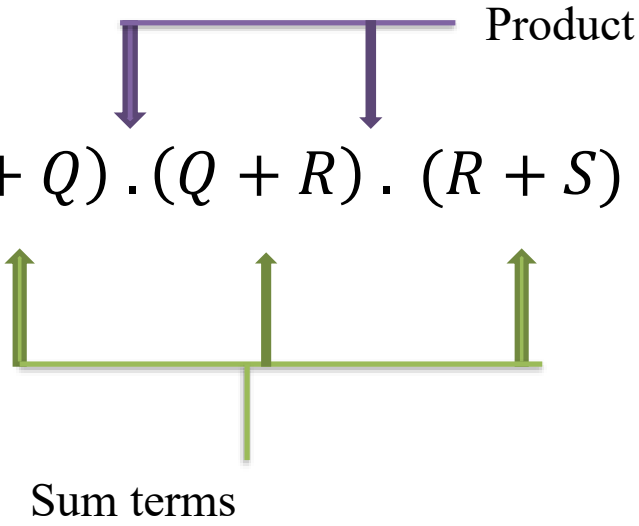
- Product of Sum form (POS)

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



Conjunctive normal formula or conjunctive normal form

$$f(P,Q,R,S) = (\bar{P} + Q)\,.(Q + R)\,.\ (R + S)$$



Standard (Canonical) SOP & Standard (Canonical) POS Form

- If each term in the SOP form contains all the literals then the SOP form is known as **standard or canonical SOP form.**
- Each individual term in the standard SOP form is called **minterm**.

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

- If each term in POS form contains all the literals then the POS form is known as **standard or canonical POS form.**
- Each individual term in the standard SOP form is called **maxterm**.

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

Converting Expressions in Standard SOP form

Step 1: Find the missing literal in each product term if any.

Step 2:AND each product term having missing literal(s) with term(s) form by ORing the literal and its complement.

Step 3:Expand the terms by applying distributive law and reorder the literals in the product terms.

Step 4:Reduce the expression by omitting the repeated product terms if any.

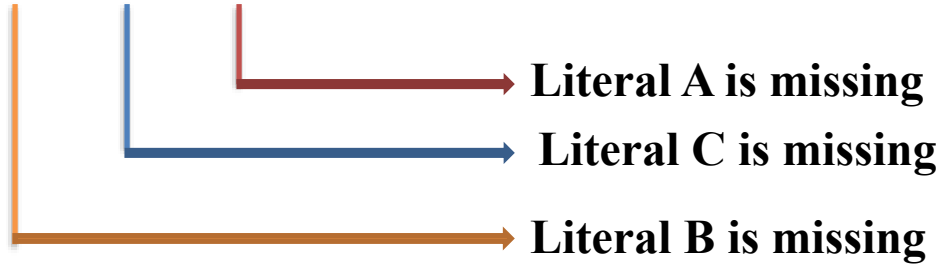
Example: Convert the given expression in standard SOP form.

f(A,B,C)=AC+AB+BC

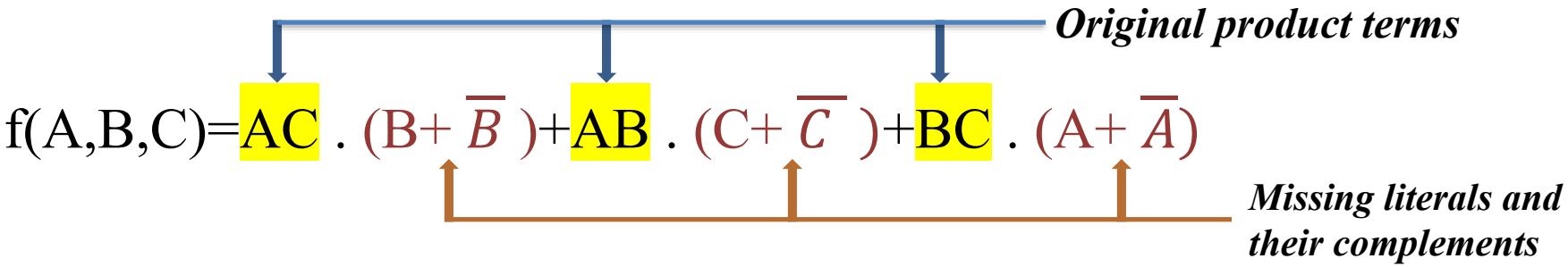
Solution:

Step 1: Find the missing literals in each product term

f(A,B,C)=AC+AB+BC



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



Step 3:Expand the terms and reorder the literals

Expand: f(A,B,C)=ACB + AC$\overline{B}$ +ABC+AB$\overline{C}$ +BCA + BC$\overline{A}$

Reorder: f(A,B,C)=ABC + A$\overline{B}$C+ABC+AB$\overline{C}$ +ABC+$\overline{A}$BC

Step 4: Omit repeated product terms

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

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

Converting Expressions in Standard POS form

Step 1: Find the missing literal in each sum term if any.

Step 2:OR each sum term having missing literal(s) with term(s) form by ANDing the literal and its complement.

Step 3:Expand the terms by applying distributive law and reorder the literals in the sum terms.

Step 4:Reduce the expression by omitting the repeated sum terms if any.

Example: Convert the given expression in standard POS form. Y=A . (A+B+C)

Solution:

Step 1: Find the missing literals in each sum term

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

Literals B and C are missing

Step 2: OR sum term with (missing literal . its complement)

$Y=[A+(B.\overline{B})+(C.\overline{C})].(A+B+C)$

Step 3: Expand the terms and reorder literals

$Y=(A+B).(A+\overline{B})+(C.\overline{C})].(A+B+C)$

$Y=(A+B+C).(A+\overline{B}+C).(A+B+\overline{C}).(A+\overline{B}+\overline{C}).(A+B+C)$

Step 4: Omit repeated sum terms

$Y=(A+B+C).(A+\overline{B}+C).(A+B+\overline{C}).(A+\overline{B}+\overline{C}).(A+B+C)$

$Y=(A+B+C).(A+\overline{B}+C).(A+B+\overline{C}).(A+\overline{B}+\overline{C})$

M Notations: Minterms and Maxterms

- Each individual term in standard SOP form is called **minterm** and each individual term in standard POS form is called **maxterm**.
- In general, for a n-variable logical function there are 2^n minterms and an equal number of maxterms.
- Each minterm is represented by m_i and each maxterm is represented by M_i , where the subscript i is the decimal number equivalent of the natural binary number.

  - Σ, m_i → denotes sum of product form(SOP)
  - Π, M_i → denotes product of sum form(POS)

Minterms and Maxterms for three variables

Decimal	Variables			Minterms (SOP)	Maxterms (POS)
	A (4)	B (2)	C (1)	m_i	M_i
0	0	0	0	$\bar{A}\bar{B}\,\bar{C}=m_0$	$A+B+C=M_0$
1	0	0	1	$\bar{A}\bar{B}\,C=m_1$	$A+B+\bar{C}=M_1$
2	0	1	0	$\bar{A}\,B\bar{C}=m_2$	$A+\bar{B}+C=M_2$
3	0	1	1	$\bar{A}\,BC=m_3$	$A+\bar{B}+\bar{C}=M_3$
4	1	0	0	$A\bar{B}\,\bar{C}=m_4$	$\bar{A}+B+C=M_4$
5	1	0	1	$A\bar{B}\,C=m_5$	$\bar{A}+B+\bar{C}=M_5$
6	1	1	0	$AB\bar{C}=m_6$	$\bar{A}+\bar{B}+C=M_6$
7	1	1	1	$ABC=m_7$	$\bar{A}+\bar{B}+\bar{C}=M_7$

Minterms and Maxterms for four variables

Decimal	Variables				Minterms	Maxterms
	A	B	C	D	m_i	M_i
0	0	0	0	0		
1						
2						
3						
4						
5						
6						
7						
8						
9						
10	1	0	1	0		
11						
12						
13						
14						
15	1	1	1	1		

Examples:

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

$=m_7 + m_5 + m_6 + m_3$

$=\Sigma m(3,5,6,7)$

$Y=(A+B+C).\ (A+\overline{B}+C)\ .\ (A+B+\overline{C}).\ (A+\overline{B}+\overline{C})$

$=M_0\ .\ M_2\ .\ M_1\ .\ M_3$

$=\pi M(0,1,2,3)$

Complements of Standard Forms:

$f(A,B,C)= m_7 + m_5 + m_6 + m_3 = M_0\ .\ M_1\ .\ M_2\ .\ M_4$

$f(A,B,C)= \Sigma m(3,5,6,7) = \pi M(0,1,2,4)$

In case of four variables,

$f(A,B,C,D)= \Sigma m(0,2,3,5,6,7,11,13,14) = \pi M(1,4,8,9,10,12,15)$

Express F=A+B'C as sum of minterms.

Solution:

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

$=\Sigma m(7,5,6,4,5,1)$

$=\Sigma m(1,4,5,6,7)$

Express the following F=XY+X'Z in product of maxterm.

K-map Minimization

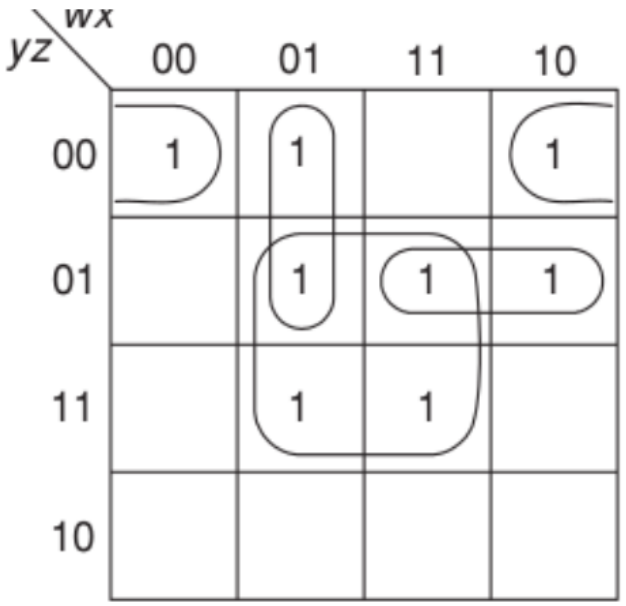
- During the process of simplification of Boolean expression we have to predict each successive step
- We can never be absolutely certain that an expression simplified by Boolean algebra alone is the simplest possible expression
- On the other hand, the map method gives us a systematic approach for simplifying a Boolean expression
- The map method, first proposed by Veitch and modified by Karnaugh, hence it is known as the **Veitch diagram or the Karnaugh map**.

Advantages of K-Maps

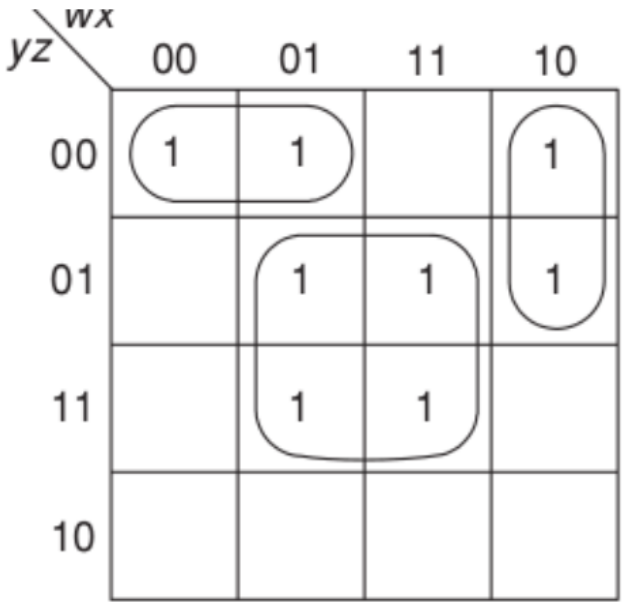
- The K-map simplification technique is simpler and less error-prone compared to the method of solving the logical expressions using Boolean laws.
- It prevents the need to remember each and every Boolean algebraic theorem.
- It involves fewer steps than the algebraic minimization technique to arrive at a simplified expression.
- K-map simplification technique always results in minimum expression if carried out properly.

MINIMIZATION

Example: Two irredundant expressions for $f(w,x,y,z) = \sum(0,4,5,7,8,9,13,15)$



(a) $f = x'y'z' + w'xy' + wy'z + xz$ is an irredundant expression.

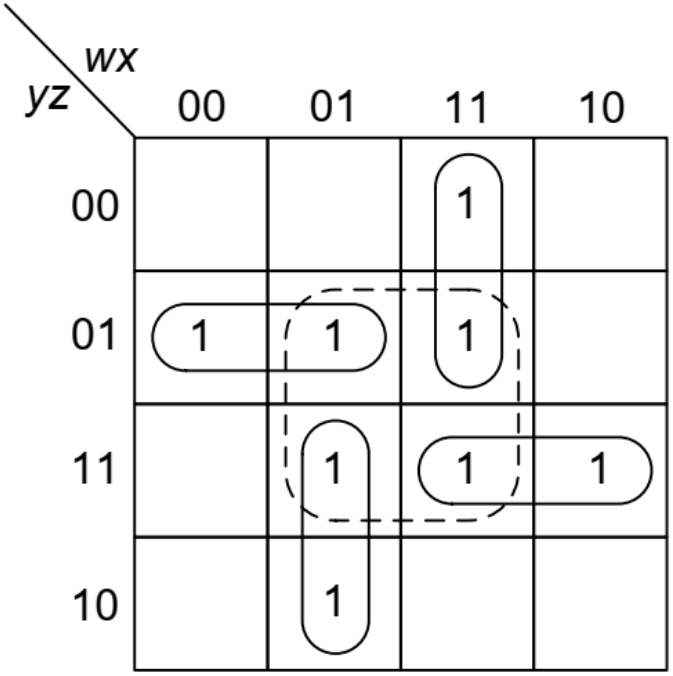


(b) $f = w'y'z' + wx'y' + xz$ is the unique minimal expression.

MINIMIZATION

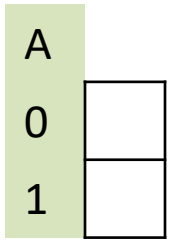
Example: $f(w,x,y,z) = \sum(1,5,6,7,11,12,13,15)$

Only one irredundant form: $f = wxy' + wyz + w'xy + w'y'z$
Dotted cube xz is redundant

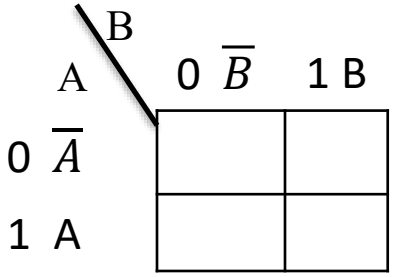


One-Variable, Two-Variable, Three-Variable and Four-Variable Maps

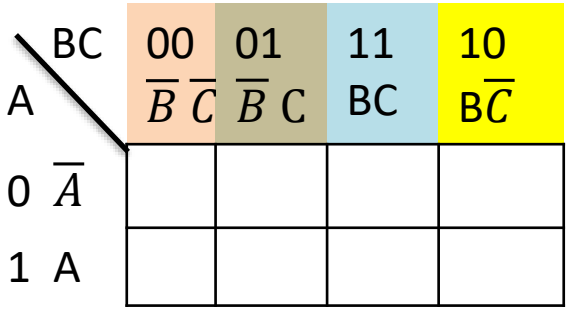
- The basis of this method is a graphical chart known as Karnaugh map (K-map)
- It contains boxes called cells.
- Each of the cell represents one of the 2^n possible products that can be formed from n variables.
- Thus, a 2-variable map contains 2^2 =4 cells, a 3-variable map contains 2^3 =8 cells and so forth.



1-Variable map
(2 cells)



2-Variable map
(4 cells)



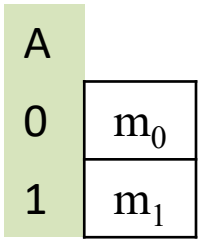
3-Variable map
(8 cells)

4-variable map (16 cells)

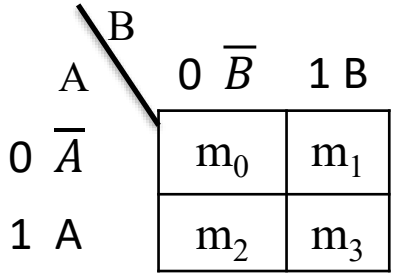
CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$				
01 $\overline{A}\,B$				
11 AB				
10 $A\,\overline{B}$				

One-Variable, Two-Variable, Three-Variable and Four-Variable Maps

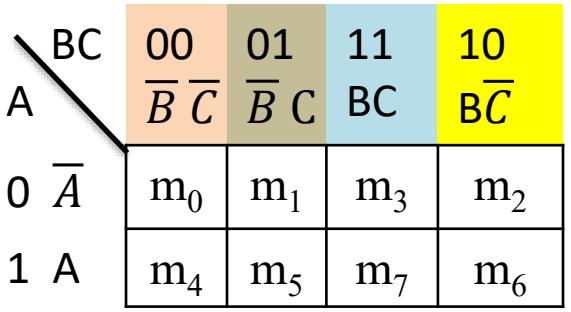
Representation:



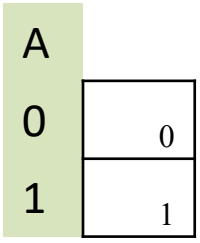
1-Variable map
(2 cells)

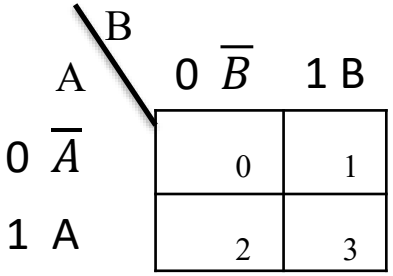


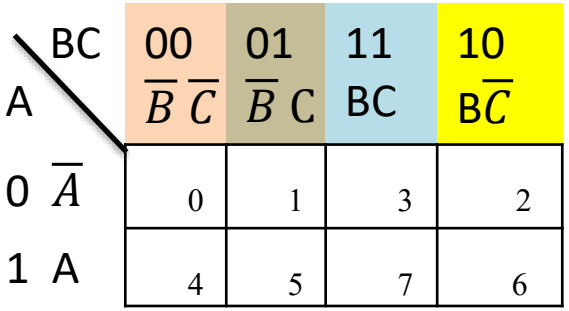
2-Variable map
(4 cells)



3-Variable map
(8 cells)







4-variable map (16 cells)

CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$	m_0	m_1	m_3	m_2
01 $\overline{A}\,B$	m_4	m_5	m_7	m_6
11 AB	m_{12}	m_{13}	m_{15}	m_{14}
10 $A\,\overline{B}$	m_8	m_9	m_{11}	m_{10}

CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$	**0**	**1**	**3**	**2**
01 $\overline{A}\,B$	**4**	**5**	**7**	**6**
11 AB	**12**	**13**	**15**	**14**
10 $A\,\overline{B}$	**8**	**9**	**11**	**10**

One-Variable, Two-Variable, Three-Variable and Four-Variable Maps

Representation(POS):

A	
0	M_0
1	M_1

1-Variable map
(2 cells)

A \ B	0 B	$1\overline{B}$
0 A	M_0	M_1
1 $\overline{A}$	M_2	M_3

2-Variable map
(4 cells)

A \ B+C	00 $B+C$	01 B+$\overline{C}$	11 $\overline{B}+\overline{C}$	10 $\overline{B}$+C
0 A	M_0	M_1	M_3	M_2
1 $\overline{A}$	M_4	M_5	M_7	M_6

3-Variable map
(8 cells)

A	
0	0
1	1

A \ B	0 B	1 $\overline{B}$
0 A	0	1
1 $\overline{A}$	2	3

A \ B+C	00 $B+C$	01 B+$\overline{C}$	11 $\overline{B}+\overline{C}$	10 $\overline{B}$+C
0 A	0	1	3	2
1 $\overline{A}$	4	5	7	6

4-variable map (16 cells)

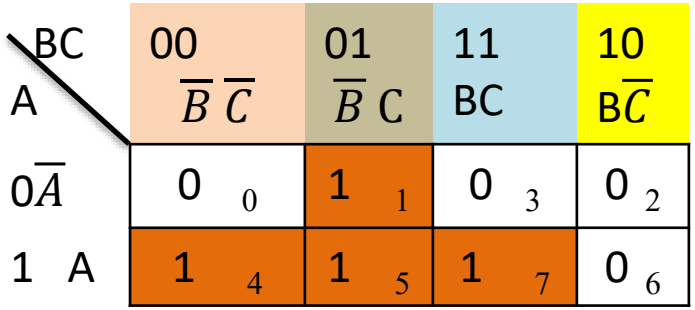
C+D / A+B	C+D 00	$C+\overline{D}$ 01	$\overline{C}+\overline{D}$ 11	$\overline{C}$+D 10
00 $A+B$	M_0	M_1	M_3	M_2
01 A+$\overline{B}$	M_4	M_5	M_7	M_6
11 $\overline{A}+\overline{B}$	M_{12}	M_{13}	M_{15}	M_{14}
10 $\overline{A}$+B	M_8	M_9	M_{11}	M_{10}

C+D / A+B	C+D 00	$C+\overline{D}$ 01	$\overline{C}+\overline{D}$ 11	$\overline{C}$+D 10
00 $A+B$	**0**	**1**	**3**	**2**
01 A+$\overline{B}$	**4**	**5**	**7**	**6**
11 $\overline{A}+\overline{B}$	**12**	**13**	**15**	**14**
10 $\overline{A}$+B	**8**	**9**	**11**	**10**

Plotting a K-map

Cell: the smallest unit of a Karnaugh map, corresponding to one line of a truth table. The input variables are the cell's co-ordinates and the output variable is the cell's contents.

No.	Inputs			Output
	A	B	C	Y
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	0
4	1	0	0	1
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1



No.	Inputs				Output
	A	B	C	D	Y
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	1
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	0
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	1
15	1	1	1	1	1

CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$	*1* 0	*1* 1	0 3	*1* 2
01 $\overline{A}\,B$	0 4	*1* 5	*1* 7	0 6
11 AB	*1* 12	13	*1* 15	*1* 14
10 $A\,\overline{B}$	0 8	*1* 9	0 11	0 10

Representing Standard SOP on K-Map

- Plot the Boolean expression $Y = AB\bar{C} + ABC + \bar{A}\bar{B}C$ on the Karnaugh map.

A \ BC	00 $\bar{B}\,\bar{C}$	01 $\bar{B}$ C	11 BC	10 B$\bar{C}$
0 $\bar{A}$	0 0	1 1	0 3	0 2
1 A	0 4	0 5	1 7	1 6

- Plot the Boolean expression $Y = AB\bar{C}D + ABCD + \bar{A}\bar{B}CD + \bar{A}\bar{B}\bar{C}\bar{D} +$ ABC$\bar{D}$ + $\bar{A}BC\bar{D}$ on the Karnaugh map.

AB \ CD	$\bar{C}\,\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
00 $\bar{A}\,\bar{B}$	0	1	3	2
01 $\bar{A}\,B$	4	5	7	6
11 AB	12	13	15	14
10 $A\,\bar{B}$	8	9	11	10

- Represent the following in Karnaugh map f(a,b,c)=Σm(1,4,6,7)

bc / a	00 $\overline{b}\,\overline{c}$	01 $\overline{b}\,c$	11 bc	10 $b\overline{c}$
0 $\overline{a}$	0 0	1 1	0 3	0 2
1 a	1 4	0 5	1 7	1 6

- Represent the following in Karnaugh map f(w,x,y,z)=Σm(1,2,5,6,7,11,14)

yzw / x	$\overline{y}\,\overline{z}$ 00	$\overline{y}z$ 01	yz 11	$y\overline{z}$ 10
00 $\overline{w}\,\overline{x}$	0	1	3	2
01 $\overline{w}\,x$	4	5	7	6
11 wx	12	13	15	14
10 $w\overline{x}$	8	9	11	10

Representing Standard POS on K-Map:

- Plot the Boolean expression Y=(A+B+C). (A+$\overline{B}$+C) . (A+B+ $\overline{C}$). (A+$\overline{B}$+ $\overline{C}$)

A \ B+C	00 $B+C$	01 B+$\overline{C}$	11 $\overline{B}+\overline{C}$	10 $\overline{B}$+C
0 A	0 0	0 1	0 3	0 2
1 $\overline{A}$	1 4	1 5	1 7	1 6

- Plot the expression F(A,B,C,D)=(A+B+C+D). (A+$\overline{B}$+C+D) . (A+B+ $\overline{C}$ + D). (A+$\overline{B}$+ $\overline{C}$ + D) . (A+B+$\overline{C}$ + $\overline{D}$)

A+B \ C+D	C+D 00	$C+\overline{D}$ 01	$\overline{C}+\overline{D}$ 11	$\overline{C}$+D 10
00 $A+B$	0	1	3	2
01 A+$\overline{B}$	4	5	7	6
11 $\overline{A}+\overline{B}$	12	13	15	14
10 $\overline{A}$+B	8	9	11	10

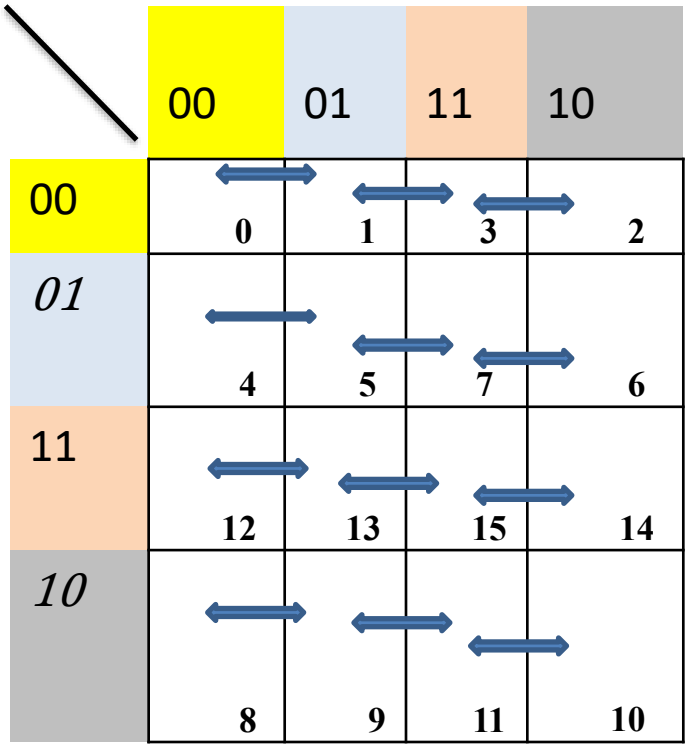
- Represent the function in k-map f=πM(1,4,6,9,11)

A+B \ C+D	C+D 00	$C+\overline{D}$ 01	$\overline{C}+\overline{D}$ 11	$\overline{C}$+D 10
00 $A+B$	1 0	0 1	1 3	1 2
01 A+$\overline{B}$	0 4	1 5	1 7	0 6
11 $\overline{A}+\overline{B}$	1 12	1 13	1 15	1 14
10 $\overline{A}$+B	1 8	0 9	0 11	1 10

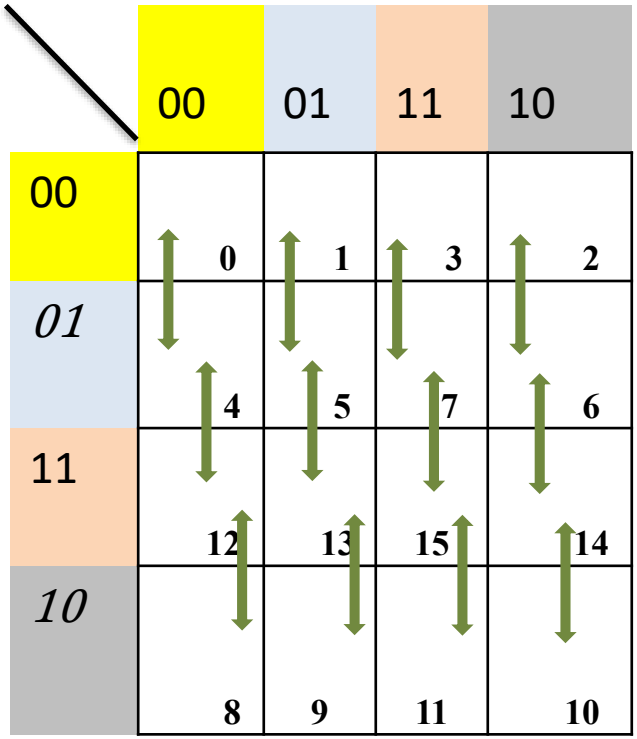
Grouping cells for simplification

- Once the Boolean function is plotted on the Karnaugh map we have to use grouping technique to simplify the Boolean function.
- The grouping is nothing combining terms in adjacent cells.
- Two cells are said to be adjacent if they conform the single change rule.,i.e., there is only one variable difference between co-ordinates of two cells.

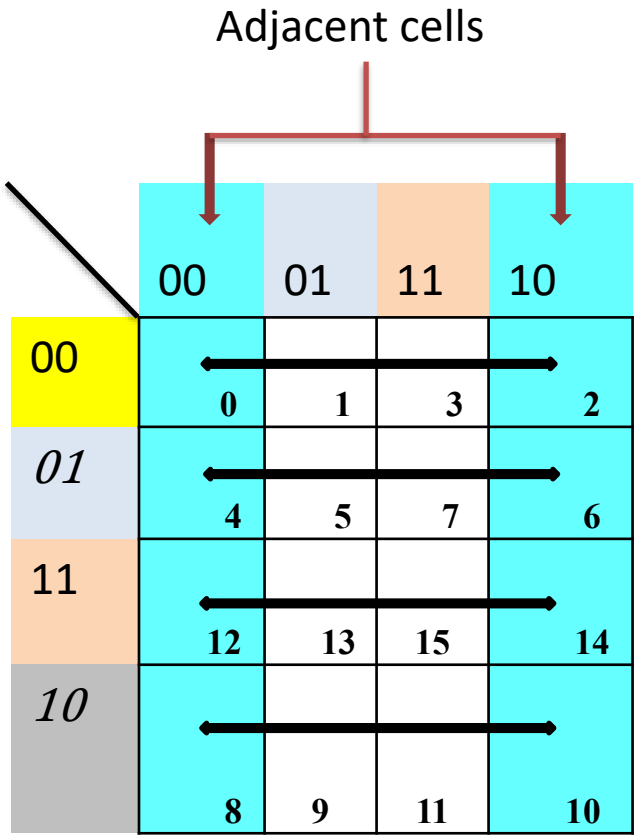
Grouping Two Adjacent Ones(Pair)



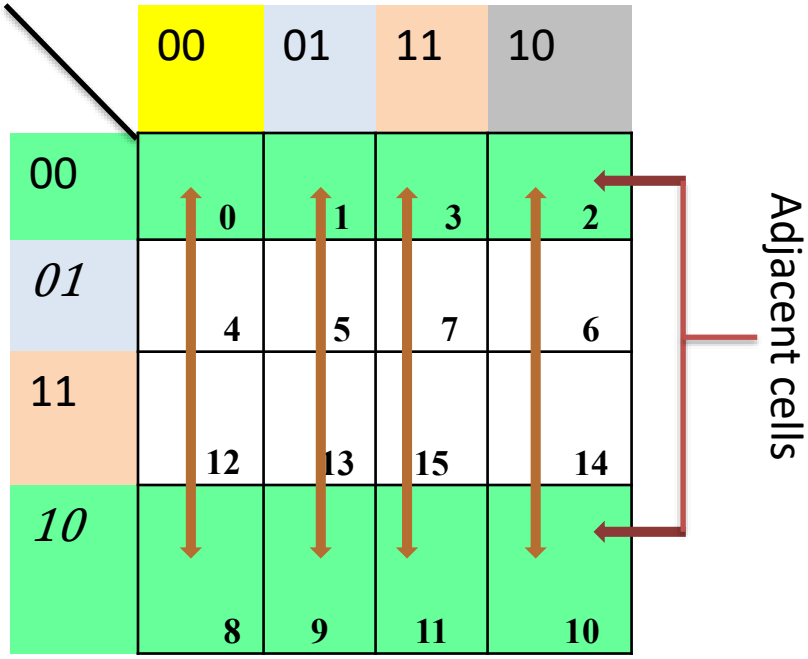
Neighbouring cells in the row are adjacent



Neighbouring cells in the column are adjacent



Leftmost and corresponding rightmost cells are adjacent



Top and corresponding bottom cells are adjacent

$$Y = \bar{A}BC + ABC$$

A \ BC	00 $\bar{B}\,\bar{C}$	01 $\bar{B}$ C	11 BC	10 B$\bar{C}$
0 $\bar{A}$	0	1	1 3	2
1 A	4	5	1 7	6

A \ BC	00 $\bar{B}\,\bar{C}$	01 $\bar{B}$ C	11 BC	10 B$\bar{C}$
0 $\bar{A}$	0 0	1 1	1 3	0 2
1 A	0 4	0 5	0 7	0 6

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

A \ BC	00 $\bar{B}\,\bar{C}$	01 $\bar{B}$ C	11 BC	10 B$\bar{C}$
0 $\bar{A}$	0 0	0 1	1 3	0 2
1 A	0 4	0 5	1 7	0 6

$$Y = BC$$

A \ BC	00 $\bar{B}\,\bar{C}$	01 $\bar{B}$ C	11 BC	10 B$\bar{C}$
0 $\bar{A}$	0 0	1 1	1 3	0 2
1 A	0 4	0 5	1 7	0 6

$\bar{A}C$

$$BC$$

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

CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$	0	1 1	3	2
01 $\overline{A}\,B$	4	5	7	6
11 AB	12	13	15	14
10 $A\,\overline{B}$	8	1	11	10

$$Y = \bar{B}\bar{C}D$$

BC / A	00 $\overline{B}\,\overline{C}$	01 $\overline{B}$ C	11 BC	10 B$\overline{C}$
0 $\overline{A}$	0 0	1 1	1 3	0 2
1 A	0 4	0 5	1 7	1 6

Grouping four adjacent ones(Quad)

A \ BC	00 $\overline{B}\,\overline{C}$	01 $\overline{B}\,C$	11 BC	10 $B\overline{C}$
0 $\overline{A}$	0 0	0 1	0 3	0 2
1 A	1 4	1 5	1 7	1 6

→ A

CD \ AB	00 $\overline{C}\,\overline{D}$	01 $\overline{C}D$	11 CD	10 $C\overline{D}$
00 $\overline{A}\,\overline{B}$	0 0	1 1	0 3	0 2
01 $\overline{A}\,B$	0 4	1 5	0 7	0 6
11 AB	0 12	1 13	0 15	0 14
10 $A\,\overline{B}$	0 8	1 9	0 11	0 10

→ $\overline{C}D$

CD \ AB	00 $\overline{C}\,\overline{D}$	01 $\overline{C}D$	11 CD	10 $C\overline{D}$
00 $\overline{A}\,\overline{B}$	0 0	0 1	0 3	0 2
01 $\overline{A}\,B$	0 4	1 5	1 7	0 6
11 AB	0 12	1 13	1 15	0 14
10 $A\,\overline{B}$	0 8	0 9	0 11	0 10

→ BD

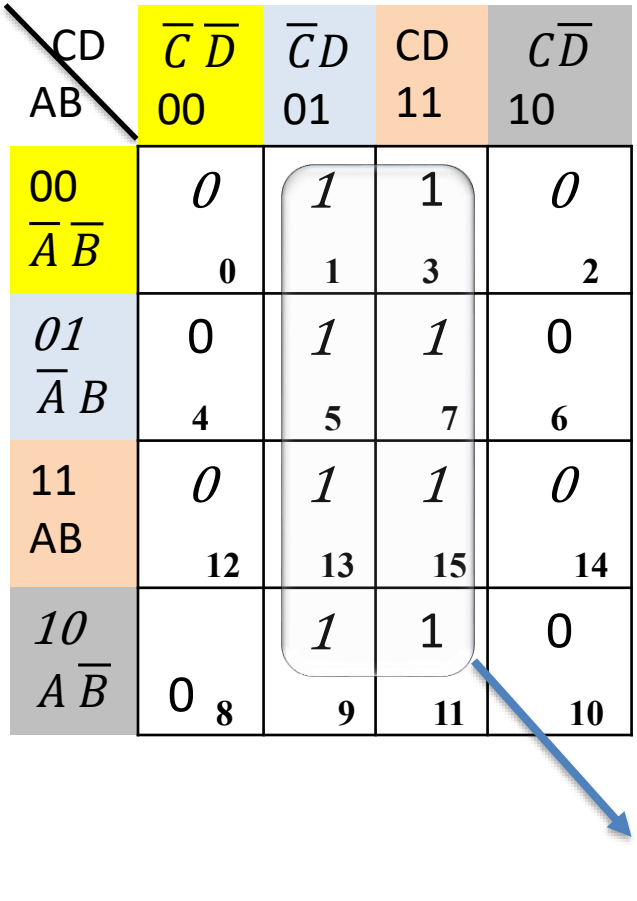
CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$	1 0	0 1	0 3	1 2
01 $\overline{A}\,B$	0 4	0 5	0 7	0 6
11 AB	0 12	0 13	0 15	0 14
10 $A\,\overline{B}$	1 8	0 9	0 11	1 10

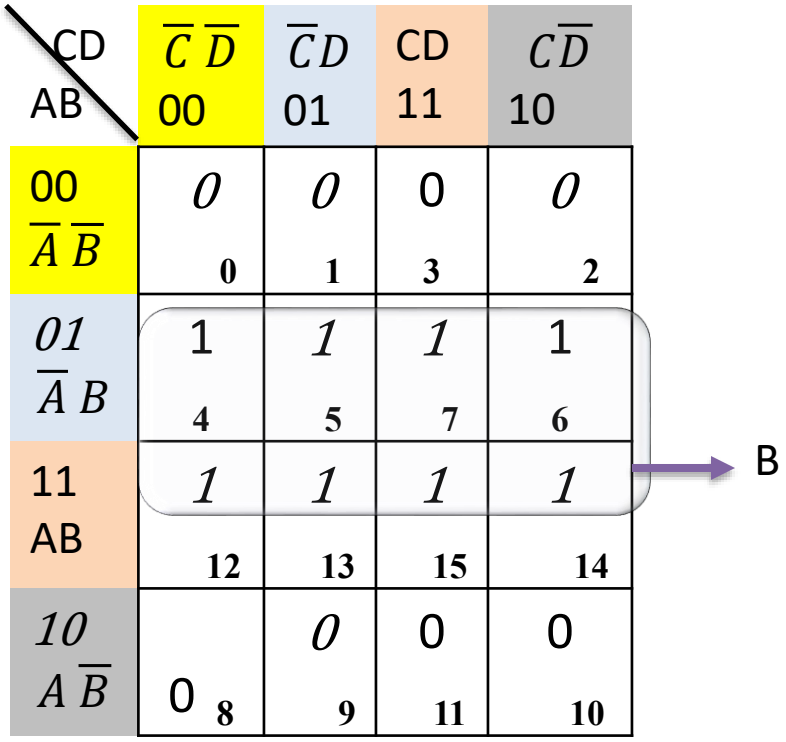
$\overline{B}\,\overline{D}$

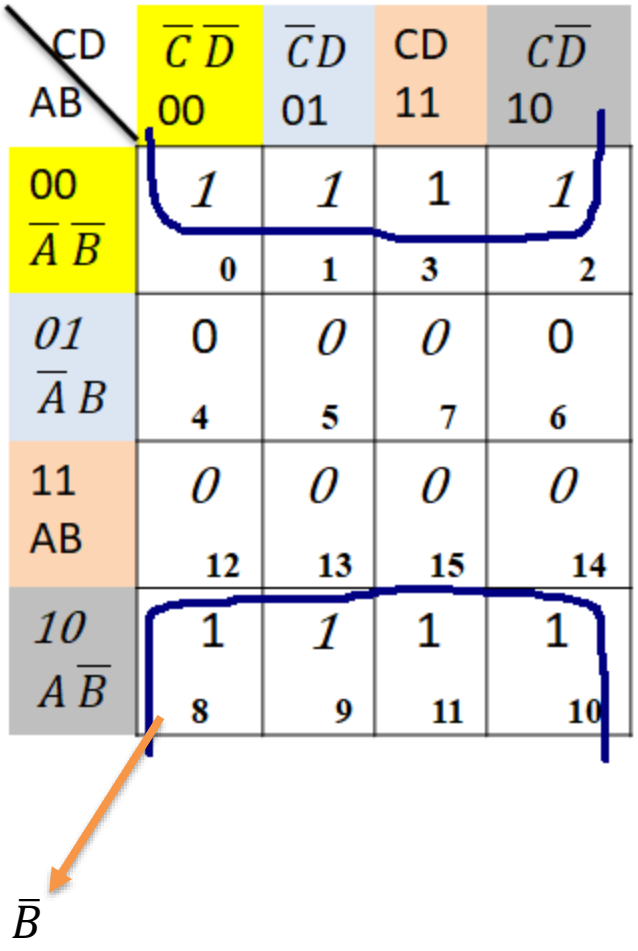
CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$	0 0	0 1	0 3	0 2
01 $\overline{A}\,B$	0 4	0 5	0 7	0 6
11 AB	1 12	0 13	0 15	1 14
10 $A\,\overline{B}$	1 8	0 9	0 11	1 10

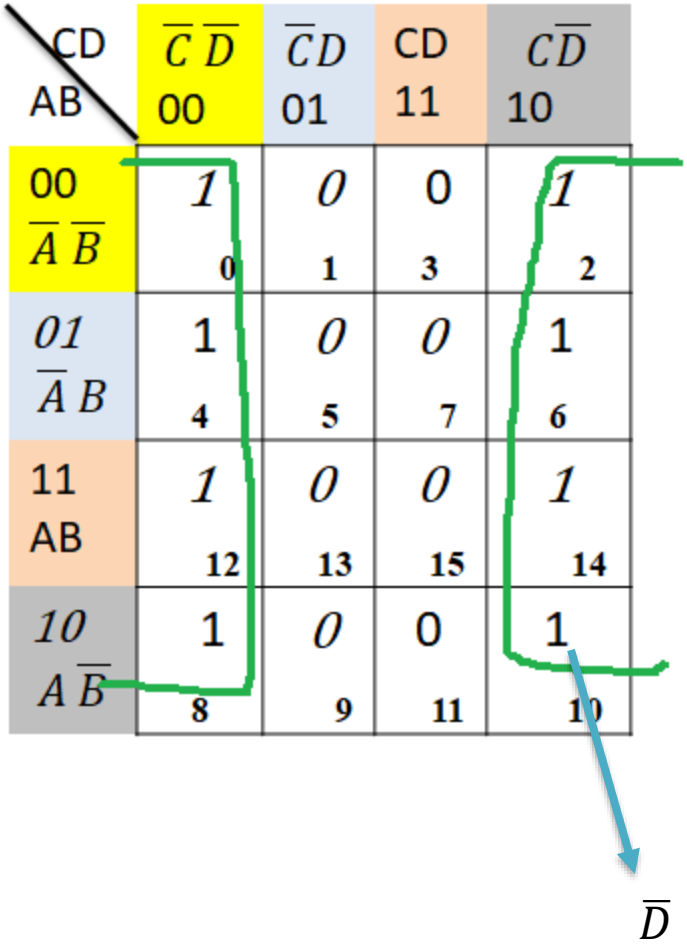
$A\overline{D}$

Grouping Eight Adjacent Ones (Octet)






CD
AB
$\overline{C}\,\overline{D}$ 00
$\overline{C}D$ 01
CD 11
$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$
01 $\overline{A}\,B$
11 AB
10 $A\,\overline{B}$
1 1 1 1
0 1 3 2
0 0 0 0
4 5 7 6
0 0 0 0
12 13 15 14
1 1 1 1
8 9 11 10
$\overline{B}$


CD
AB
$\overline{C}\,\overline{D}$ 00
$\overline{C}D$ 01
CD 11
$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$
01 $\overline{A}\,B$
11 AB
10 $A\,\overline{B}$
1 0 0 1
0 1 3 2
1 0 0 1
4 5 7 6
1 0 0 1
12 13 15 14
1 0 0 1
8 9 11 10
$\overline{D}$

Illegal Grouping

CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$	0 0	0 1	1 3	0 2
01 $\overline{A}\,B$	0 4	1 5	0 7	0 6
11 AB	0 12	0 13	0 15	0 14
10 $A\,\overline{B}$	0 8	0 9	0 11	0 10

Diagonal grouping is illegal

CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$	0 0	1 1	1 3	0 2
01 $\overline{A}\,B$	0 4	1 5	0 7	1 6
11 AB	0 12	1 13	0 15	1 14
10 $A\,\overline{B}$	0 8	1 9	0 11	1 10

Grouping odd number of cells is illegal

- Reduce the following using k-map

Y= $\bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D} + ABC\bar{D} + A\bar{B}C\bar{D} + A\bar{B}CD + A\bar{B}\bar{C}\bar{D} + AB\bar{C}\bar{D} + \bar{A}\bar{B}CD$

CD / AB	$\bar{C}\,\bar{D}$ 00	$\bar{C}D$ 01	CD 11	$C\bar{D}$ 10
00 $\bar{A}\,\bar{B}$	1 0	0 1	1 3	1 2
01 $\bar{A}\,B$	0 4	0 5	0 7	0 6
11 AB	1 12	0 13	0 15	1 14
10 $A\,\bar{B}$	1 8	0 9	1 11	1 10

a)F(a,b,c,d)=Σ(0,1,2,4,5,7,11,15)

b)F(A,B,C)=Σ(0,3,4,7)

c) F(A,B,C,D)=Σ(0,3,5,7,8,9,10,12,15)

d)Minimize Y=$\bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + AB\bar{C}\bar{D} + \bar{A}\bar{B}C\bar{D}$

Essential Prime Implicants

- After grouping the cells, the sum terms appear in the k-map are called essential prime implicants groups.
- Some cells may appear in only one prime implicants group; while other cells may appear in more than one prime implicants group.
- The cells which appear in only one prime implicant group are called essential cells and corresponding implicants are called essential prime implicants.

Incompletely Specified Functions(Don’t Care Terms)

- In some logic circuits, certain input conditions never occur, therefore the corresponding output never appears. In such cases the output level is not defined, it can be either HIGH or LOW. These output levels are indicated by ‘X’ or ‘d’ in the truth tables and are called don’t care outputs or don’t care conditions or incompletely specified functions.

- Find the reduced SOP form of the following function.

F(A,B,C,D)=Σm(1,3,7,11,15)+Σd(0,2,4)

CD / AB	$\overline{C}\,\overline{D}$ 00	$\overline{C}D$ 01	CD 11	$C\overline{D}$ 10
00 $\overline{A}\,\overline{B}$	X 0	1 1	1 3	X 2
01 $\overline{A}\,B$	X 4	0 5	1 7	0 6
11 AB	0 12	0 13	1 15	0 14
10 $A\,\overline{B}$	0 8	0 9	1 11	0 10

F(a,b,c,d)= Σ(0,2,4,5,6,8,10,15)+ Σφ(7,13,14)

F(W,X,Y,Z)= Σm(0,7,8,9,10,12)+ Σd(2,5,13)

F(a,b,c,d)= Σ(0,2,4,5,6,8)+ Σφ(10,11,12,13,14,15)

Simplification of POS Expression

- Minimize the following expression

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

C+D / A+B	C+D 00	$C+\overline{D}$ 01	$\overline{C}+\overline{D}$ 11	$\overline{C}$+D 10
00 $A+B$	0	1	3	2
01 A+$\overline{B}$	4	5	7	6
11 $\overline{A}+\overline{B}$	12	13	15	14
10 $\overline{A}$+B	8	9	11	10

- Reduce the following function using K-map technique f(A,B,C,D)=πM(0,2,3,8,9,12,13,15)

C+D / A+B	C+D 00	$C+\overline{D}$ 01	$\overline{C}+\overline{D}$ 11	$\overline{C}$+D 10
00 $A+B$	0	1	3	2
01 A+$\overline{B}$	4	5	7	6
11 $\overline{A}+\overline{B}$	12	13	15	14
10 $\overline{A}$+B	8	9	11	10

- **Reduce the following function f(A,B,C,D)=π(0,3,4,7,8,10,12,14)+d(2,6)**
- **Simplify f(x,y,z)= πM(3,5,7)**
- **Simplify F(A,B,C,D)=Σ(0,1,2,5,8,9,10) in sum of products and product of sums using K-map.**
- **Simplify the Boolean expression in sum of products and product of sums using K-map,** $A\overline{C} + \overline{B}D + \overline{A}CD + ABCD$

Five variable K-Map

F(A,B,C,D,E)=Σm(0,5,6,8,9,10,11,16,20,24,25,26,27,29,31)

$\overline{A}$ (0)

DE / BC	$\overline{D}\,\overline{E}$ 00	$\overline{D}E$ 01	DE 11	D$\overline{E}$ 10
00 $\overline{B}\,\overline{C}$	0	1	3	2
01 $\overline{B}C$	4	5	7	6
11 BC	12	13	15	14
10 B$\overline{C}$	8	9	11	10

A(1)

DE / BC	$\overline{D}\,\overline{E}$ 00	$\overline{D}E$ 01	DE 11	D$\overline{E}$ 10
00 $\overline{B}\,\overline{C}$	16	17	19	18
01 $\overline{B}C$	20	21	23	22
11 BC	28	29	31	30
10 B$\overline{C}$	24	25	27	26

Rules for simplifying logic function using K-map are:

- Group should not include any cell containing a zero.
- The number of cells in a group must be a power of 2,such as 1,2,4,8 or 16.
- Group may be horizontal, vertical but not diagonal.
- Cell containing 1 must be included in at least one group
- Groups may overlap.
- Each group should be as large as possible to get maximum simplification.
- Groups may be wrapped around the map. The leftmost cell in a row may be grouped with the rightmost cell and the top cell in a column may be grouped with the bottom cell.
- A cell may be grouped more than once. The only condition is that every group must have at least one cell that does not belong to any other group. Otherwise redundant terms will result.
- We need not group all don't care cells, only those that actually contribute to a maximum simplification.

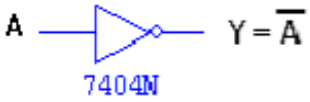
Limitations of Karnaugh map

- The map method of simplification is convenient as long as the number of variables does not exceed five or six.
- As the number of variable increases it is difficult to make judgments about which combinations form the minimum expression.
- Another important point is that the K-map simplification is manual technique and simplification process is heavily depends on the human abilities.

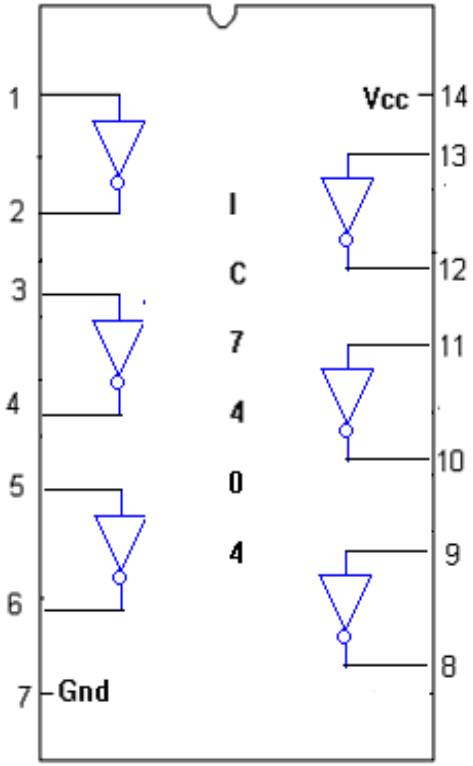
Logic gates

NOT GATE:

SYMBOL:



PIN DIAGRAM:

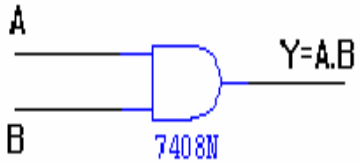


TRUTH TABLE :

A	$\overline{A}$
0	1
1	0

AND GATE:

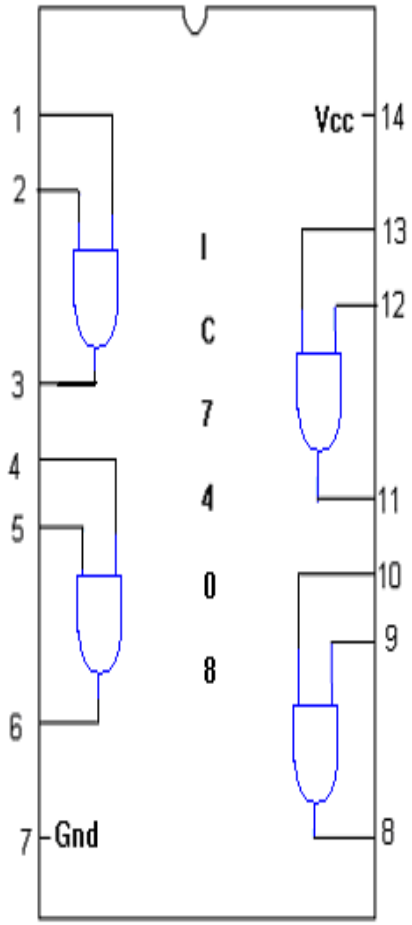
SYMBOL:



TRUTH TABLE

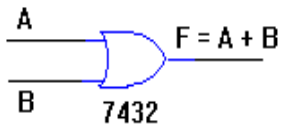
A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

PIN DIAGRAM:

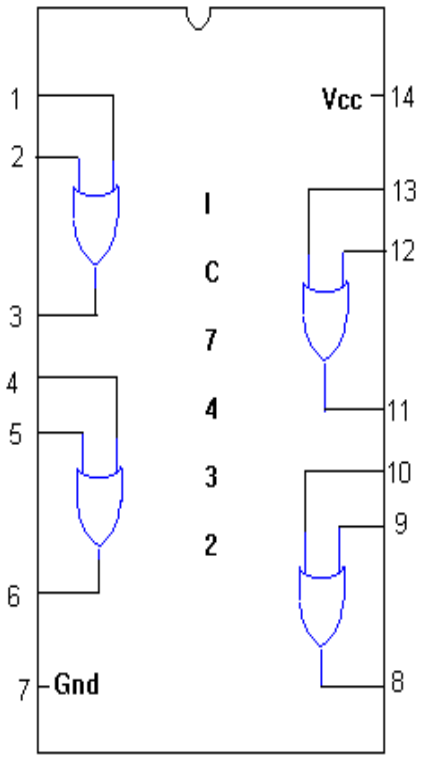


OR GATE:

SYMBOL :



PIN DIAGRAM :

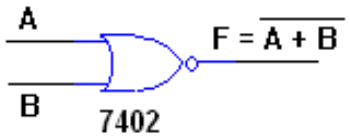


TRUTH TABLE

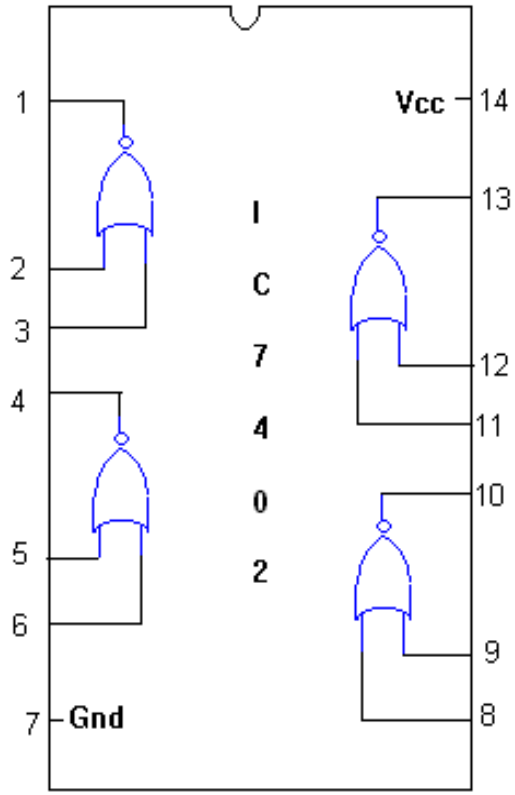
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

NOR GATE:

SYMBOL :



PIN DIAGRAM :

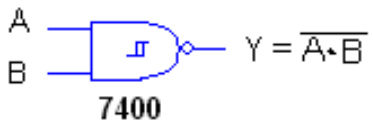


TRUTH TABLE

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

2-INPUT NAND GATE:

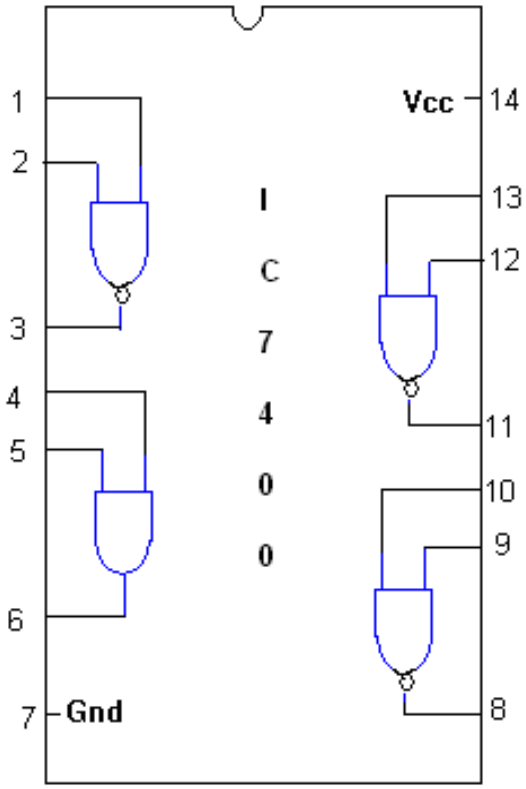
SYMBOL:



TRUTH TABLE

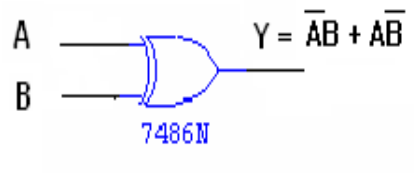
A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

PIN DIAGRAM:



X-OR GATE :

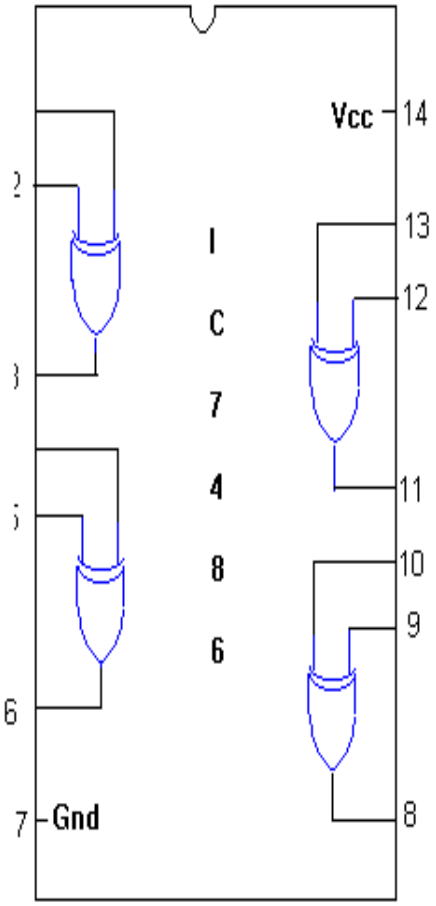
SYMBOL :

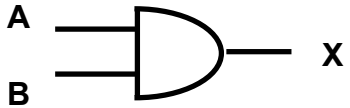
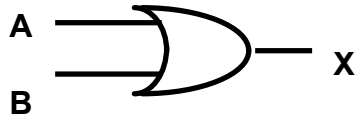
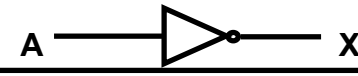
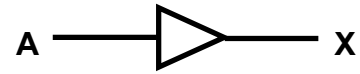
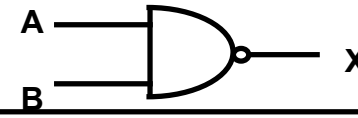
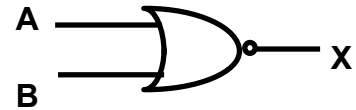
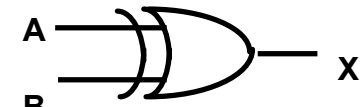
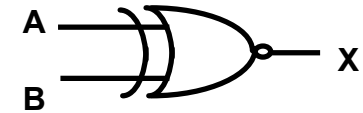


TRUTH TABLE :

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

PIN DIAGRAM :



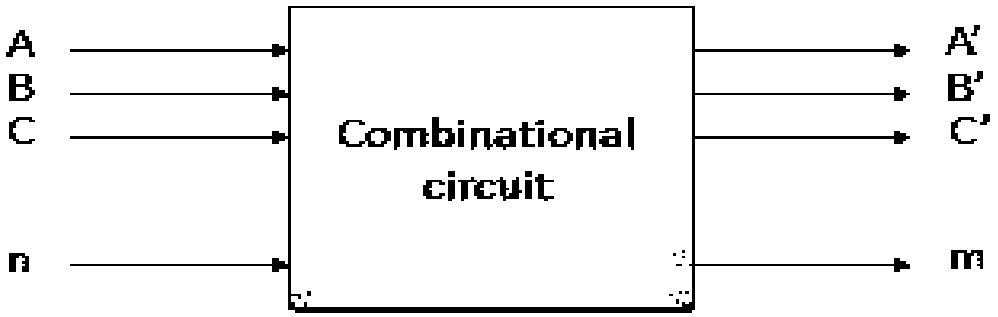
Name	Symbol	Function	Truth Table
AND		X = A • B or X = AB	A B | X 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		X = A + B	A B | X 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
I		X = A’	A | X 0 | 1 1 | 0
Buffer		X = A	A | X 0 | 0 1 | 1
NAND		X = (AB)’	A B | X 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		X = (A + B)’	A B | X 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR Exclusive OR		X = A ⊕ B or X = A’B + AB’	A B | X 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0
XNOR Exclusive NOR or Equivalence		X = (A ⊕ B)’ or X = A’B’+ AB	A B | X 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1

UNIT II
COMBINATIONAL LOGIC

- Combinational Circuits –Analysis and Design Procedures
- Binary Adder-Subtractor
- Decimal Adder
- Binary Multiplier
- Magnitude Comparator
- Decoders–Encoders
- Multiplexers
- Introduction to HDL –HDL Models of Combinational circuits.

Introduction:

- When logic gates are connected together to produce a specified output for certain specified combinations of input variables, with no storage involved, the resulting circuit is called combinational logic circuit.
- In combinational logic circuit, the output variables are at all times dependent on the combination of input variables.
- The combinational circuit do not use any memory. The previous state of input does not have any effect on the present state of the circuit.
- A combinational circuit can have an n number of inputs and m number of outputs.

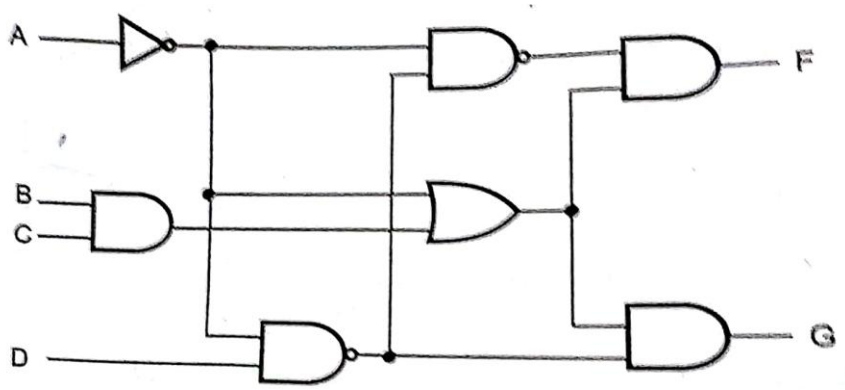


Analysis Procedure

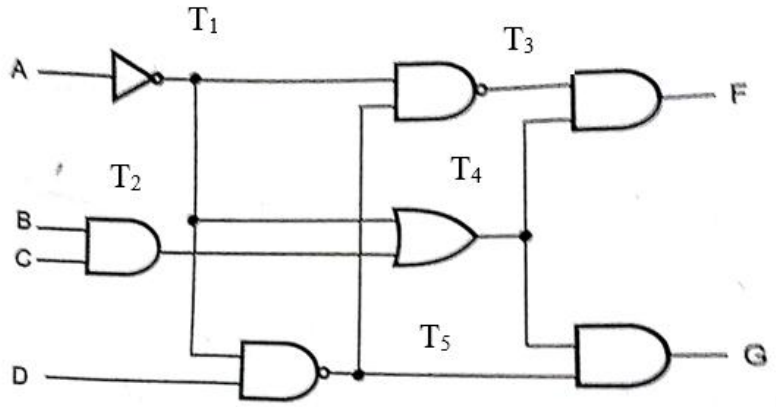
Steps to analyse combinational circuit:

1. First make sure that the given circuit is combinational circuit and not the sequential circuit. The combinational circuit has logic gates with no feedback path or memory elements.
2. Label all gate outputs that are a function of input variables with arbitrary symbols and determine the Boolean functions for each gate output.
3. Label the gates that are a function of input variables and previously labelled gates and determine the Boolean functions for them.
4. Repeat the step 3 until the Boolean function for outputs of the circuit are obtained.
5. Finally, substituting previously defined Boolean functions, obtain the output Boolean functions in terms of input variables.

1. Analyze the following logic diagram.



Solution:



$$T_1 = \bar{A}$$

$$T_2 = \mathrm{BC}$$

$$T_3 = \overline{T_1.T_5} = \bar{A}.\left(\overline{\bar{A}.D}\right) = \bar{A}.(A + \bar{D}) = \bar{A}\bar{D}$$

$$T_4 = T_1 + BC = \bar{A} + BC$$

$$T_5 = \left(\overline{\bar{A}.D}\right) = (A + \bar{D})$$

$$F = T_3.T_4 = (\bar{A}\bar{D}).(\bar{A} + BC) = \bar{A}\bar{D} + \bar{A}BC\bar{D}$$

$$= \bar{A}\bar{D}(1 + BC) = \bar{A}\bar{D}$$

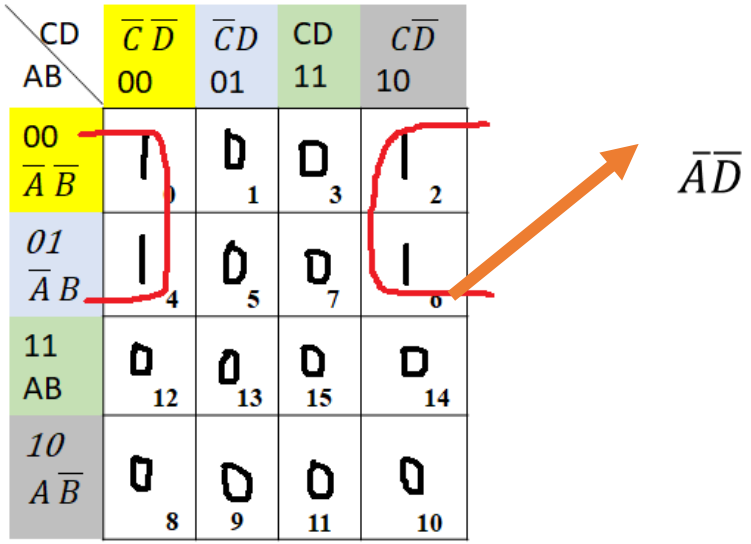
$$G = T_4.T_5 = (\bar{A} + BC).(A + \bar{D})$$

$$= \bar{A}\mathrm{A} + \bar{A}\bar{D} + ABC + BC\bar{D}$$

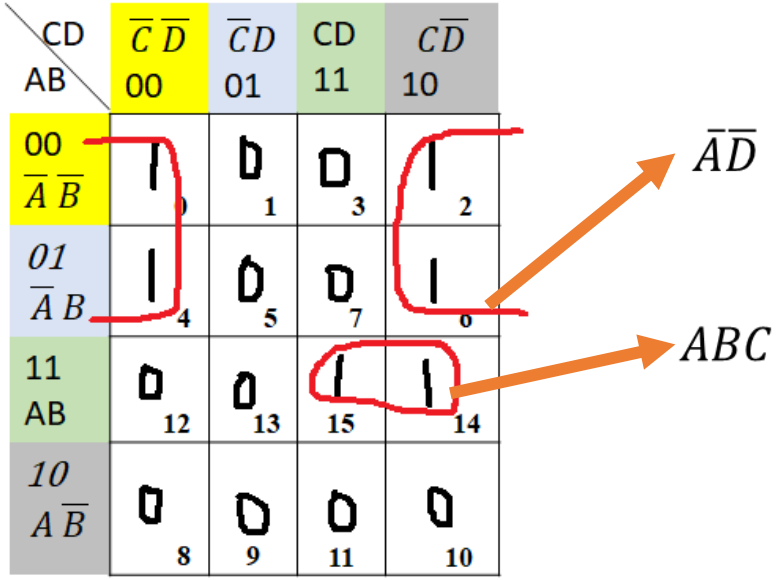
Truth Table

A	B	C	D	T_1	T_2	T_3	T_4	T_5	F	G
0	0	0	0	1	0	1	1	1	1	1
0	0	0	1	1	0	0	1	0	0	0
0	0	1	0	1	0	1	1	1	1	1
0	0	1	1	1	0	0	1	0	0	0
0	1	0	0	1	0	1	1	1	1	1
0	1	0	1	1	0	0	1	0	0	0
0	1	1	0	1	1	1	1	1	1	1
0	1	1	1	1	1	0	1	0	0	0
1	0	0	0	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0	1	0	0
1	0	1	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	1	0	0
1	1	0	1	0	0	0	0	1	0	0
1	1	1	0	0	1	0	0	1	0	1
1	1	1	1	0	1	0	0	1	0	1

K-Map for output F:



K-Map for output G:

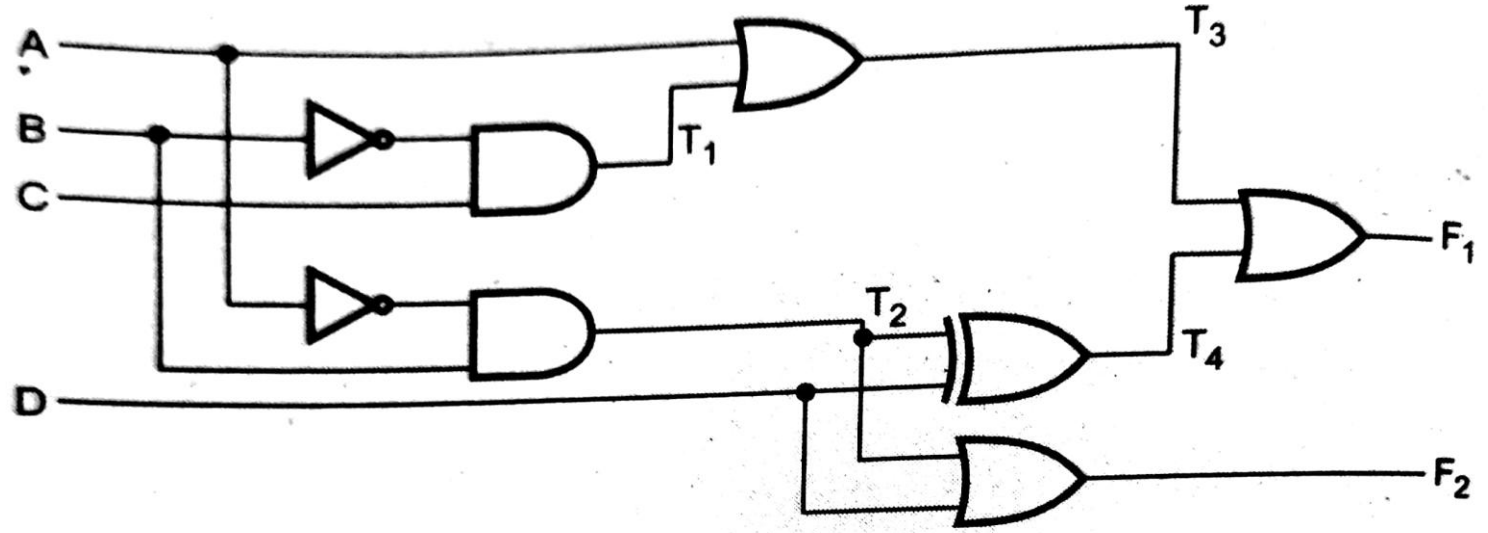


2. Consider the combinational circuit shown.

i)Derive the Boolean expressions for T_1 through T_4.Evaluate the outputs $F_1$and F_2as a function of the four inputs.

ii)List the truth table with 16 binary combinations of the four input variables. Then list the binary values for T_1 through $T_4$and the outputs $F_1$and F_2 in the table.

iii)Plot the output Boolean function obtained in part (ii) on maps and show the simplified Boolean expressions are equivalent to the ones obtained in part (i).



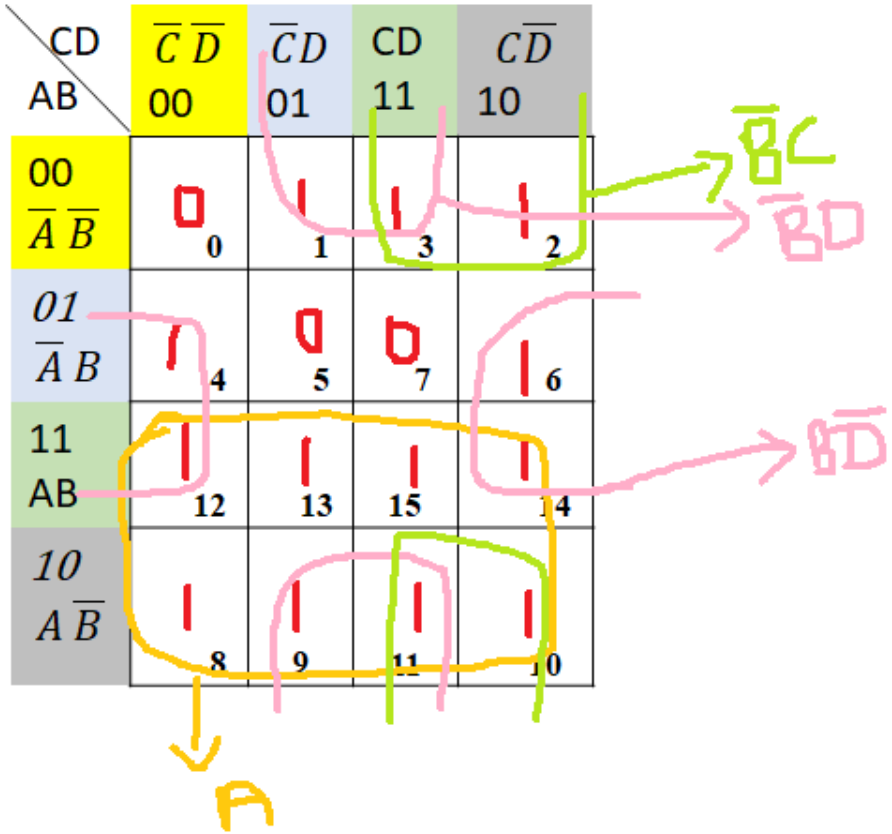
Solution:

- $T_1 = \bar{B}C$
- $T_2 = \bar{A}B$
- $T_3 = \mathrm{A} + \bar{B}\mathrm{C}$
- $T_4 = \mathrm{AD} + \bar{B}\mathrm{D} + \bar{A}\mathrm{B}\bar{D}$
- $F_1 = \mathrm{A} + \mathrm{B}\bar{D} + \bar{B}\mathrm{C} + \bar{B}\mathrm{D}$
- $F_2 = \bar{A}\mathrm{B} + \mathrm{D}$

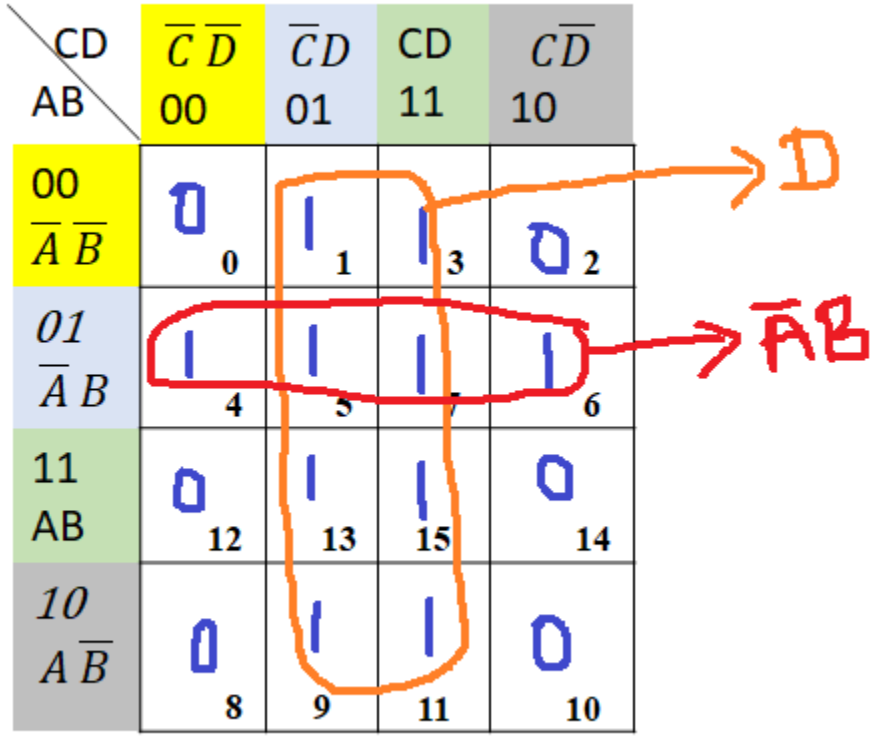
Truth Table

A	B	C	D	T_1	T_2	T_3	T_4	F_1	F_2
0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1	1	1
0	0	1	0	1	0	1	0	1	0
0	0	1	1	1	0	1	1	1	1
0	1	0	0	0	1	0	1	1	1
0	1	0	1	0	1	0	0	0	1
0	1	1	0	0	1	0	1	1	1
0	1	1	1	0	1	0	0	0	1
1	0	0	0	0	0	1	0	1	0
1	0	0	1	0	0	1	1	1	1
1	0	1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	1	1	1
1	1	0	0	0	0	1	0	1	0
1	1	0	1	0	0	1	1	1	1
1	1	1	0	0	0	1	0	1	0
1	1	1	1	0	0	1	1	1	1

K-Map for output F_1:



K-Map for output F_2:



Design Procedure

Steps to design the combinational circuit

1. The problem definition
2. The determination of number of available input variables and required output variables.
3. Assigning letter symbols to input and output variables
4. The derivation of truth table indicating the relationships between input and output variables
5. Obtain simplified Boolean expression for each output
6. Obtain the logic diagram

(i) Explain the design procedure of a combinational circuit.

(ii)The inputs to a circuit are the 4 bits of the binary number $D_3D_2D_1D_0$.The circuit produces a 1 if and only if all of the following conditions hold

1)MSB is '1'or any of the other bits are a '0'

2) D_2 is a 1 or any of the other bits are a '0'.

3)Any of the 4 bits are 0

Obtain a minimal expression for the output

D_1D_0 / D_3D_2	$\overline{D_1}\,\overline{D_0}$ 00	$\overline{D_1}D_0$ 01	D_1D_0 11	$D_1\overline{D_0}$ 10
00 $\overline{D_3}\,\overline{D_2}$	0	1	3	2
01 $\overline{D_3}D_2$	4	5	7	6
11 D_3D_2	12	13	15	14
10 $D_3\overline{D_2}$	8	9	11	10

Inputs				Output
D_3	D_2	D_1	D_0	Y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Truth Table

A majority gate is a digital circuit whose output is equal to 1 if the majority of inputs are 1's. The output is 0 otherwise. Using a truth table, find the Boolean function implemented by a 3-input majority gate. Simplify the function and implement with gates.

Solution:

Step 1:Derive the truth table

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

Step 2: Obtain the simplified Boolean expression

A \ BC	00 $\overline{B}\,\overline{C}$	01 $\overline{B}\,C$	11 BC	10 $B\overline{C}$
0 $\overline{A}$	0	1	3	2
1 A	4	5	7	6

Step 3: Draw the logic diagram

Binary Adder Subtractor

Adders:

- The most basic operation, is the addition of two binary digits.
- The simple addition consists of four possible elementary operations, namely,

$$0+0=0$$

$$0+1=1$$

$$1+0=1$$

$$1+1= 10_2$$

- The first three operations produce a sum whose length is one digit, but when the last operation is performed sum is two digits. The higher significant bit of this result is called a carry, and lower significant bit is called sum.
- The logic circuit which performs this operation is called a half-adder.
- The circuit which performs addition of three bits is called a full-adder.

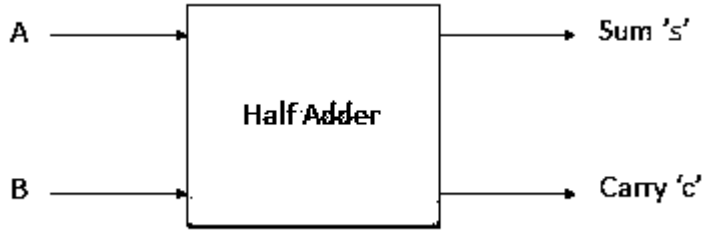
11. Design a half adder and full adder.

Two binary inputs: augend and addend bits

Two binary outputs: sum and carry

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

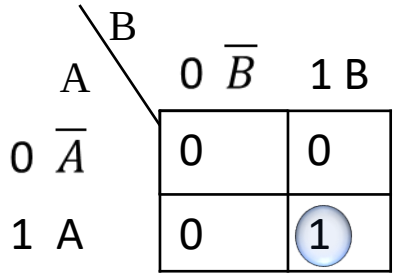
Truth Table for half adder



Block schematic of half adder

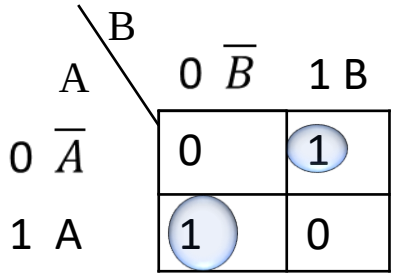
K-map simplification:

Carry



Carry=AB

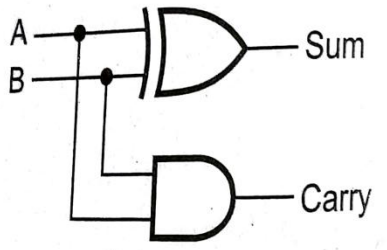
Sum



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

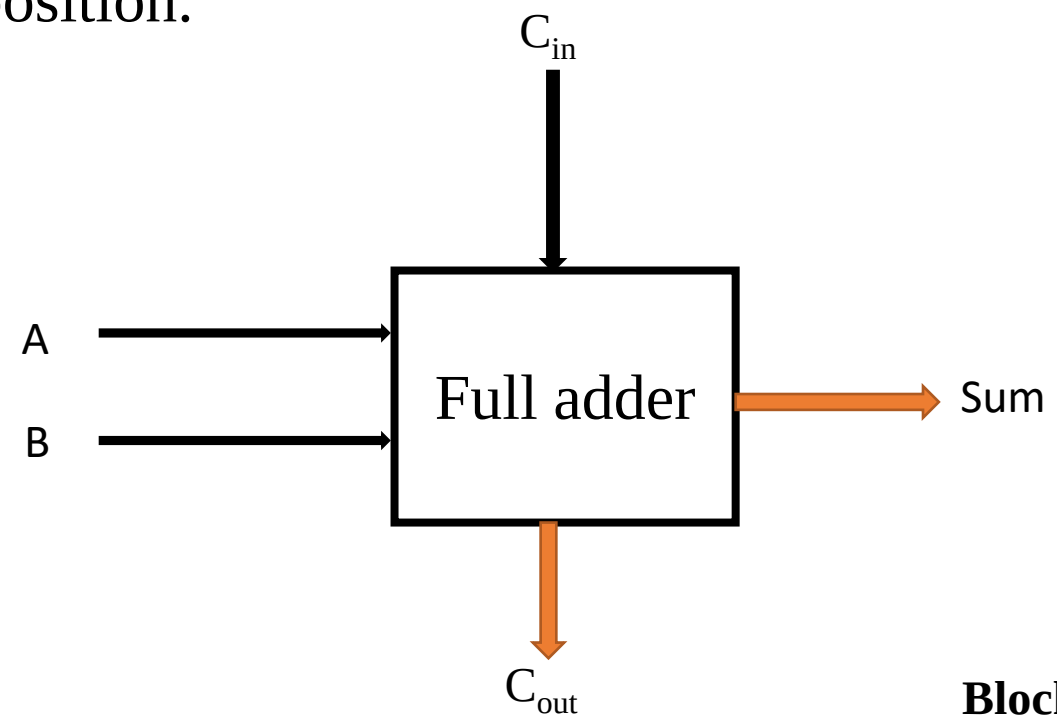
=A ⊕ B

Logic diagram:



Full Adder

- A full adder is a combinational circuit that forms the arithmetic sum of three input bits.
- It consist of three inputs and two outputs.
- Two of the input variables, denoted by A and B, represent the two significant bits to be added. The third input $C_{in,}$ represents the carry from the previous lower significant position.



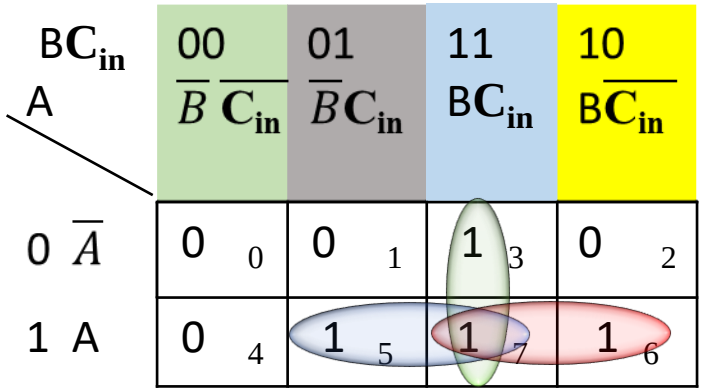
Block Schematic of full adder

Truth table of full adder

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

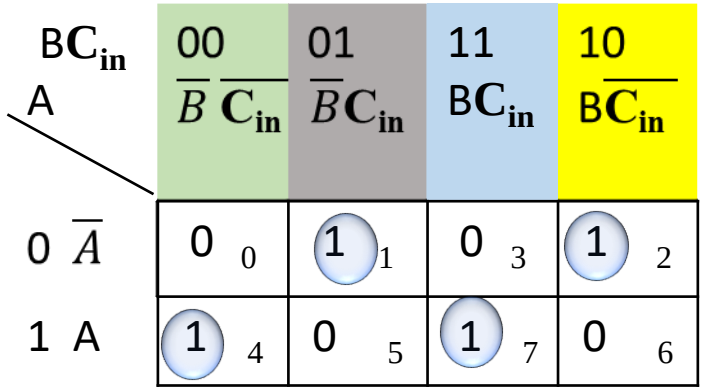
K-map simplification for carry and sum

K-map for Carry (C_{out})



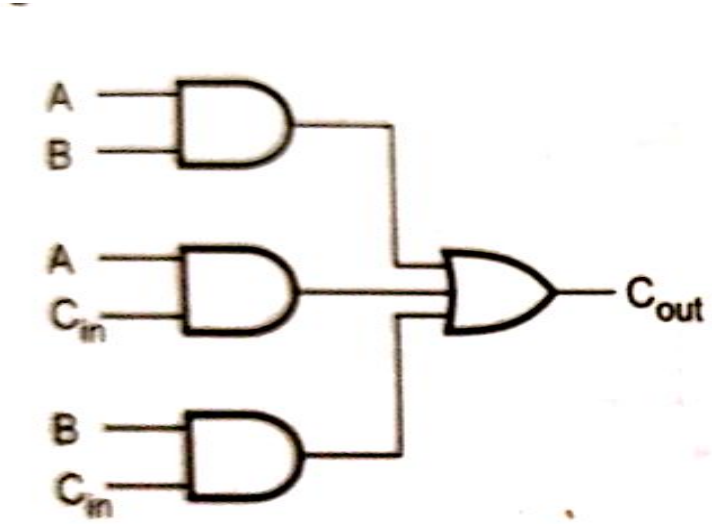
$$C_{out} = AB + AC_{in} + BC_{in}$$

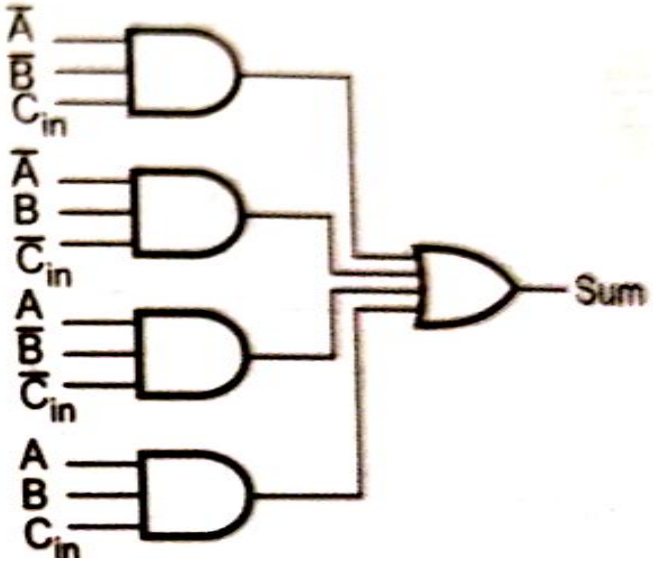
K-map for Sum



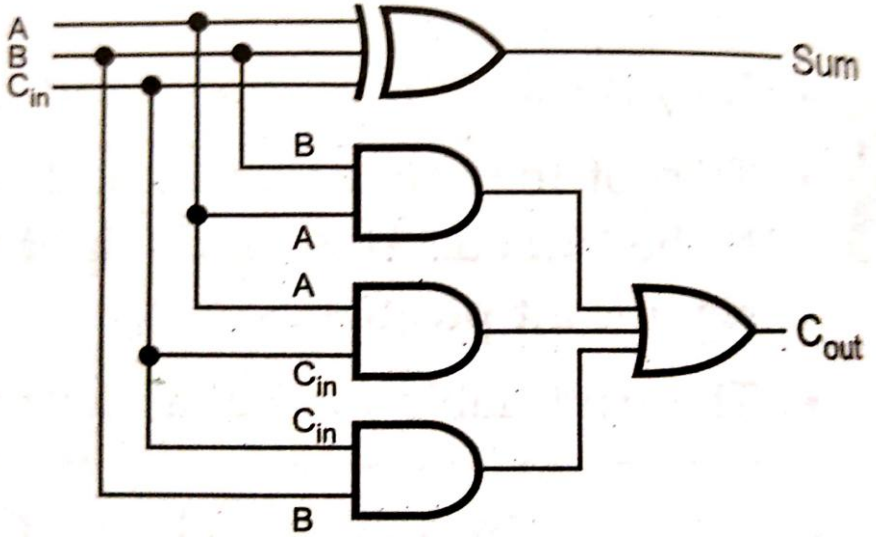
$$Sum = \overline{A}\overline{B}C_{in} + \overline{A}B\overline{C_{in}} + A\overline{B}\overline{C_{in}} + ABC_{in}$$

Implementation using logic gates:





$$Sum = \bar{A}\bar{B}C_{in} + \bar{A}B\overline{C_{in}} + A\bar{B}\overline{C_{in}} + ABC_{in}$$
$$= C_{in}(\bar{A}\bar{B} + AB) + \overline{C_{in}}(\bar{A}B + A\bar{B})$$
$$= C_{in}(A \odot B) + \overline{C_{in}}(A \oplus B)$$
$$= C_{in}(\overline{A \oplus B}) + \overline{C_{in}}(A \oplus B)$$
$$Sum = C_{in} \oplus (A \oplus B)$$



Full adder using two half adders:

$$C_{out} = AB + AC_{in} + BC_{in}$$

$$= AB + AC_{in}(B + \bar{B}) + BC_{in}(A + \bar{A})$$

$$= AB + (ABC_{in} + A\bar{B}C_{in}) + (ABC_{in} + \bar{A}BC_{in})$$

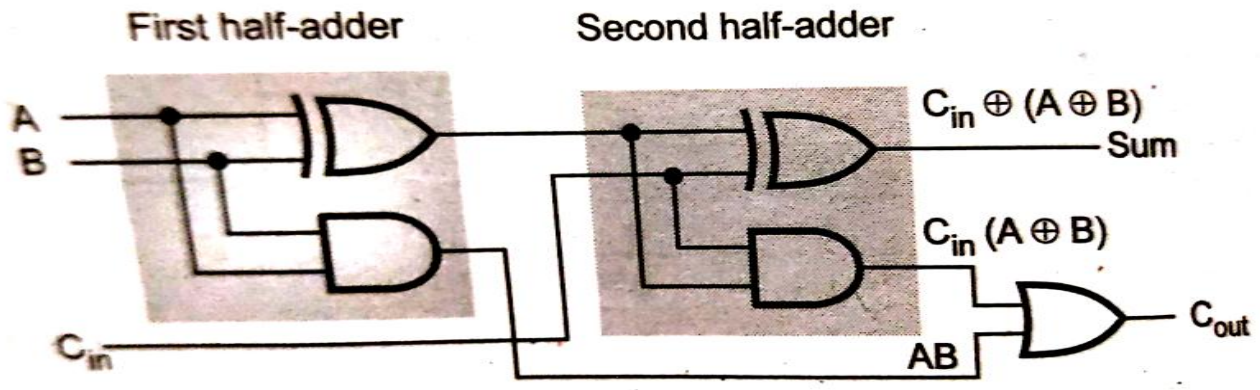
$$= AB(1 + C_{in} + C_{in}) + A\bar{B}C_{in} + \bar{A}BC_{in}$$

$$= AB + A\bar{B}C_{in} + \bar{A}BC_{in}$$

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

$$C_{out} = AB + C_{in}(A \oplus B)$$

$$Sum = C_{in} \oplus (A \oplus B)$$



Subtractors:

- The subtraction consists of four possible elementary operations, namely,

0-0=0

0-1=1 with 1 borrow

1-0=1

1-1= 0

- In all operations, each subtrahend bit is subtracted from the minuend bit.
- In case of second operation the minuend bit is smaller than the subtrahend bit, hence 1 is borrowed.

Half Subtractor:

A half-subtractor is a combinational circuit that subtracts two bits and produces their difference.

It also has an output to specify if a 1 has been borrowed.

Inputs		Outputs	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Truth Table for half subtractor

Limitations:

In multidigit subtraction, we have to add two bits along with the borrow of previous digit subtraction. Effectively such subtraction requires subtraction of three bits. This is not possible with half subtractor.

K-map simplification:

Borrow

A \ B	0 $\overline{B}$	1 B
0 $\overline{A}$	0	1
1 A	0	0

Borrow=$\overline{A}$B

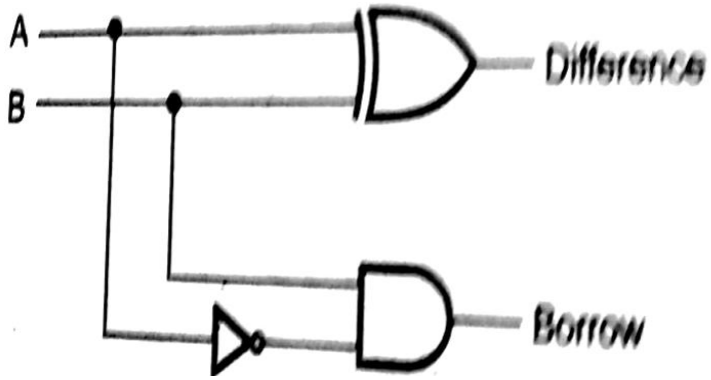
Difference

A \ B	0 $\overline{B}$	1 B
0 $\overline{A}$	0	1
1 A	1	0

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

=A $\oplus$ B

Logic diagram:



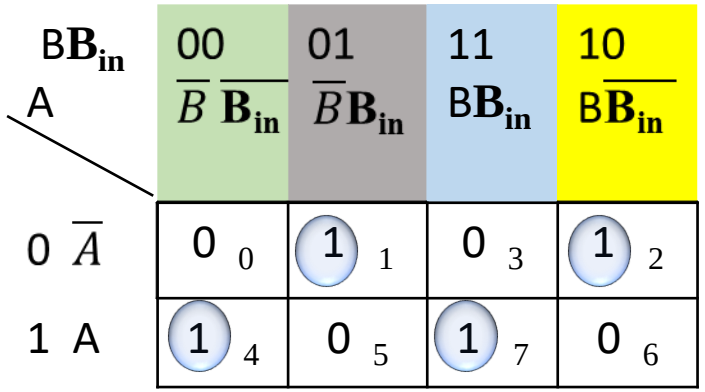
Full Subtractor

- A half-subtractor is a combinational circuit that performs a subtraction between two bits taking into account of the lower significant stage.
- This circuit has three inputs: A,B **and B_{in}** →minuend,subtrahend and previous borrow and two outputs→D and **B_{out}**

Inputs			Outputs	
A	**B**	**B_{in}**	**D**	**B_{out}**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

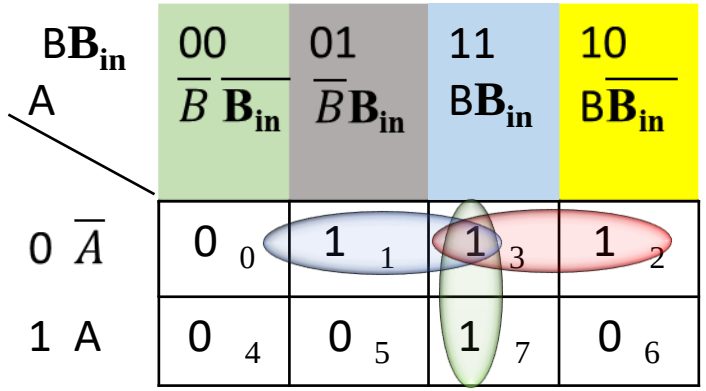
Truth table of full subtractor

K-map simplification for Difference,D:



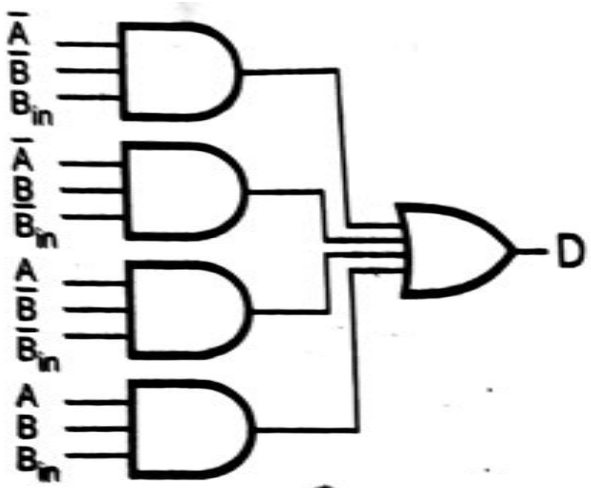
$$D = \overline{A}\overline{B}B_{in} + \overline{A}B\overline{B_{in}} + A\overline{B}\overline{B_{in}} + ABB_{in}$$

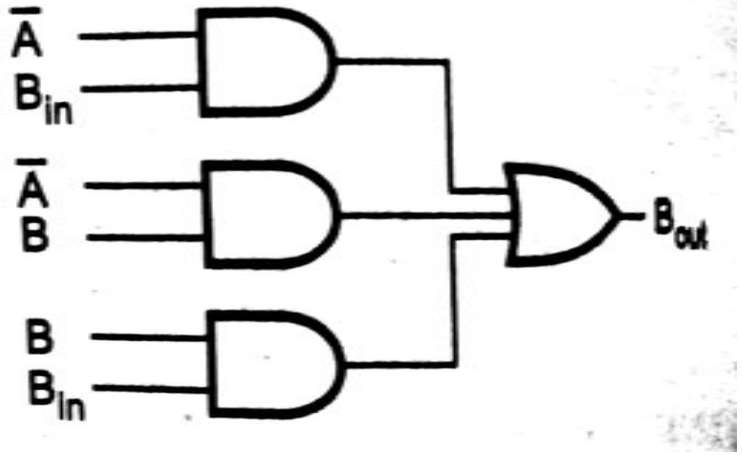
K-map simplification for B_{out} :



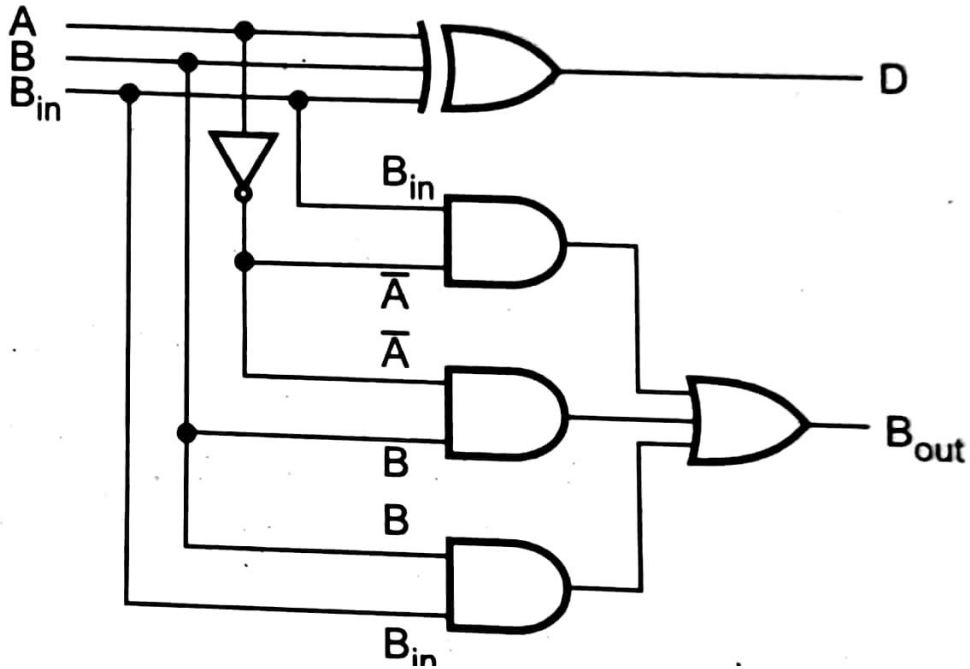
$$B_{out} = \overline{A}B + \overline{A}B_{in} + BB_{in}$$

Implementation using logic gates:





$$Difference, D = \bar{A}\bar{B}B_{in} + \bar{A}B\overline{B_{in}} + A\bar{B}\overline{B_{in}} + ABB_{in}$$
$$= B_{in}(\bar{A}\bar{B} + AB) + \overline{B_{in}}(\bar{A}B + A\bar{B})$$
$$= B_{in}(A \odot B) + \overline{B_{in}}(A \oplus B)$$
$$= B_{in}(\overline{A \oplus B}) + \overline{B_{in}}(A \oplus B)$$
$$D = B_{in} \oplus (A \oplus B)$$



Full Subtractor using two half subtractor:

$$B_{out} = \overline{A}B + \overline{A}B_{in} + BB_{in}$$

$$= \overline{A}B + \overline{A}B_{in}(B + \overline{B}) + BB_{in}(A + \overline{A})$$

$$= \overline{A}B + \overline{A}BB_{in} + \overline{A}\overline{B}B_{in} + ABB_{in} + \overline{A}BB_{in}$$

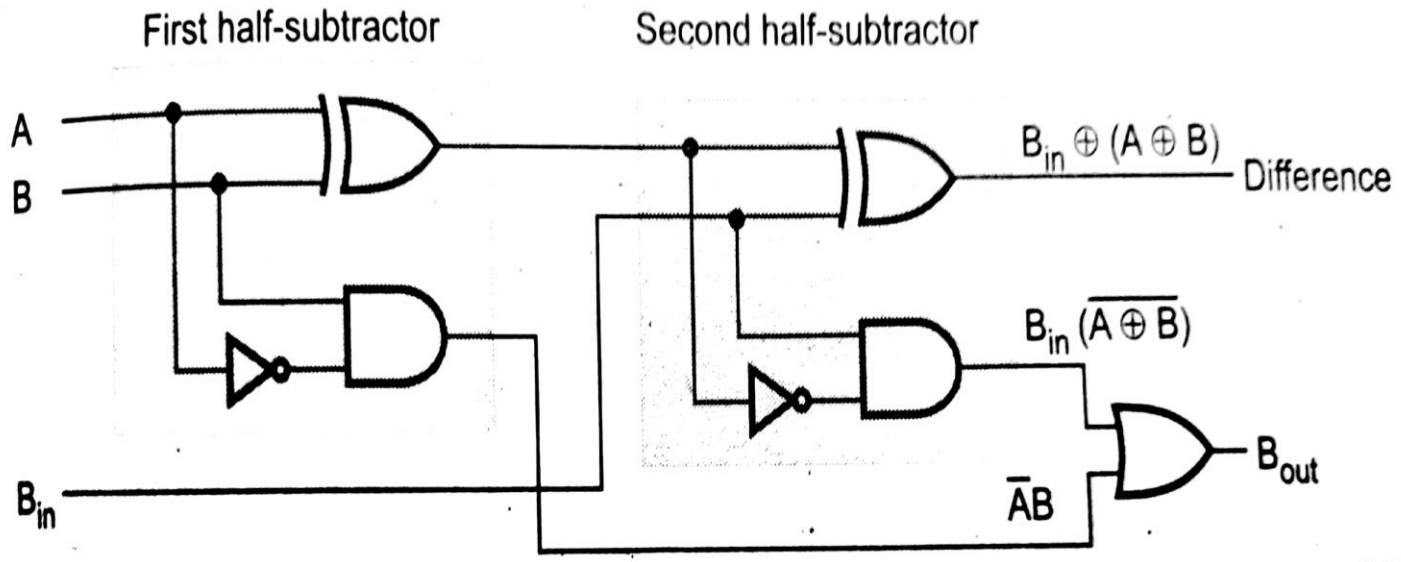
$$= \overline{A}B(1 + B_{in} + B_{in}) + \overline{A}\overline{B}B_{in} + ABB_{in}$$

$$= \overline{A}B + \overline{A}\overline{B}B_{in} + ABB_{in}$$

$$= \overline{A}B + B_{in}(\overline{A}\overline{B} + AB)$$

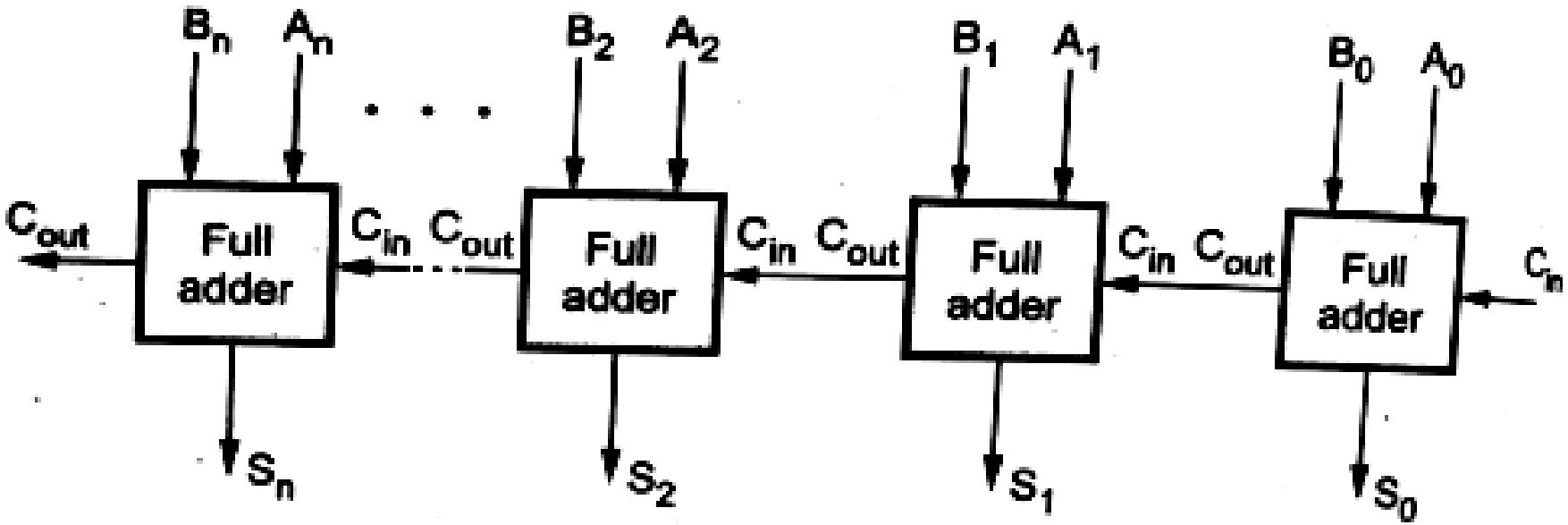
$$B_{out} = \overline{A}B + B_{in}(\overline{A \oplus B})$$

$$D = B_{in} \oplus (A \oplus B)$$

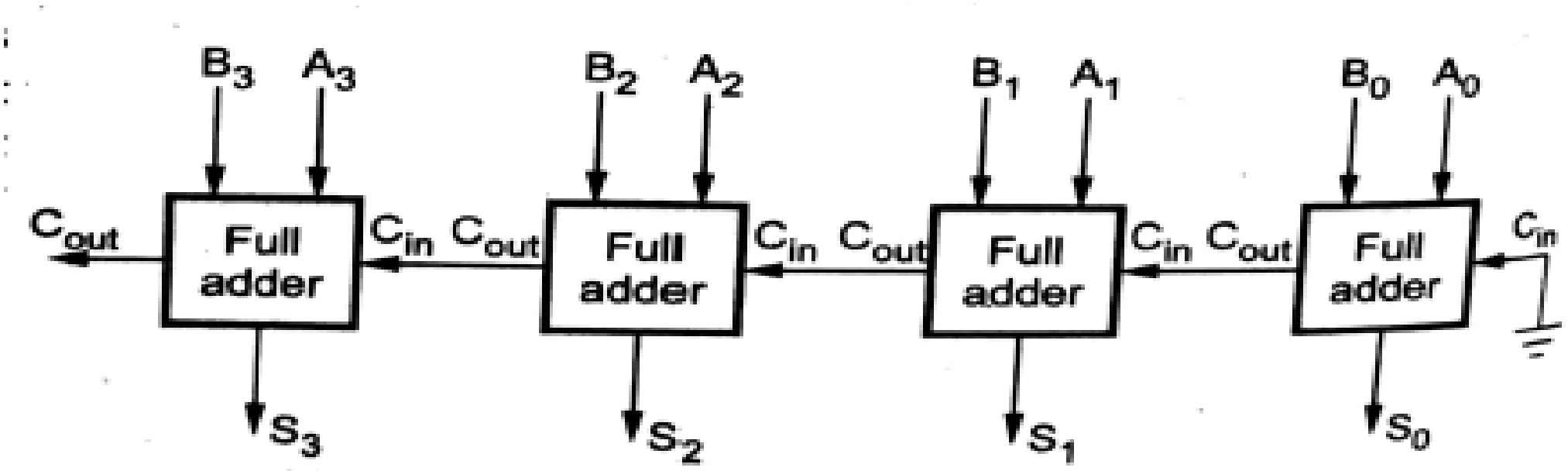


Parallel Adder:

- In order to add binary numbers with more than one bit, additional full adders must be employed.
- A n-bit parallel adder can be constructed using number of full adder circuits connected in parallel.
- The n-bit parallel adder is built using number of full adder circuits connected in cascade, i.e., the carry output of each adder is connected to the carry input of the next higher-order adder.
- It should be noted that either a half-adder can be used for the least significant position or the carry input of a full-adder is made 0 because there is no carry into the least significant bit position.



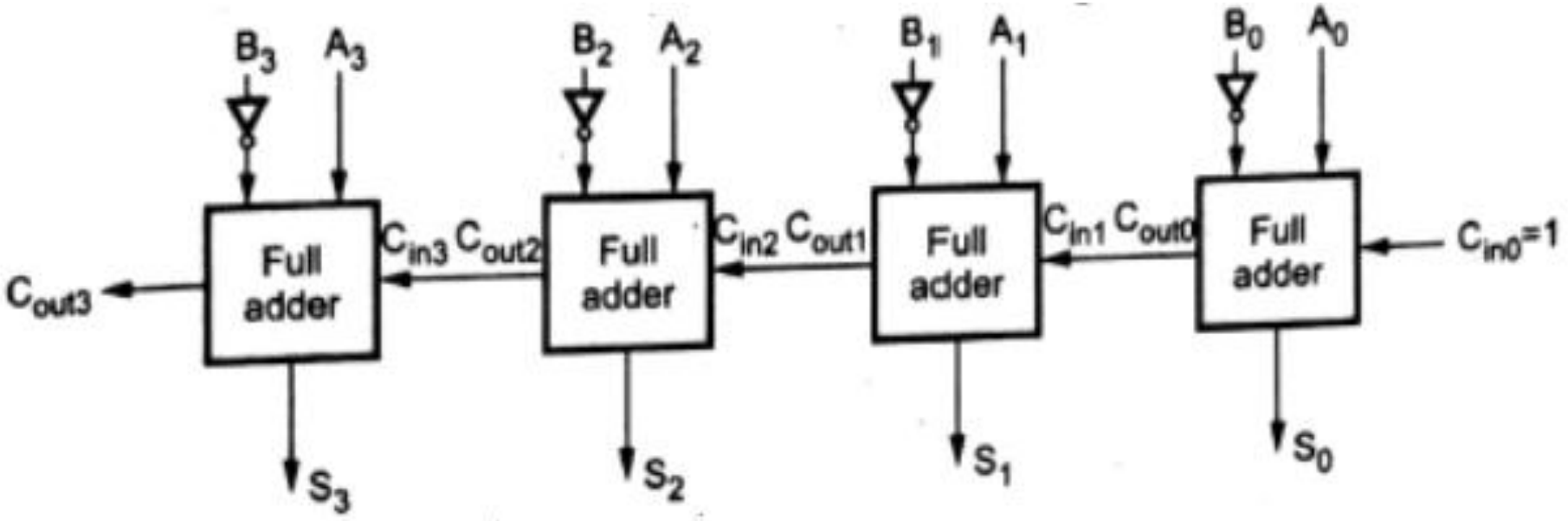
Design a 4-bit parallel adder using full-adders.



Here, for least significant position, carry input of full adder is made 0.

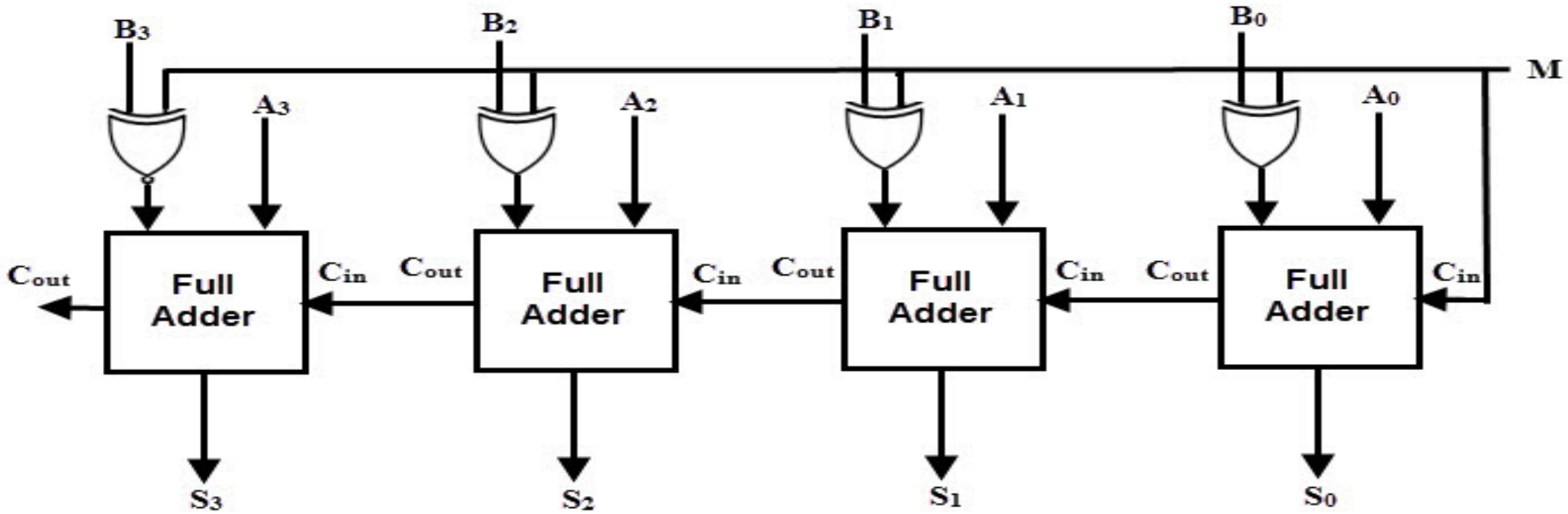
Parallel Subtractor

- The subtraction of binary numbers can be done most conveniently by means of complements.
- The subtraction A-B can be done by taking the 2's complement of B and adding it to A.
- The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits.
- The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry.



Parallel Adder/Subtractor:

- The operations of both addition and subtraction can be performed by a one common binary adder.
- Such binary circuit can be designed by adding an Ex-OR gate with each full adder.
- The mode input control line M is connected with carry input of the least significant bit of the full adder.
- This control line decides the type of operation, whether addition or subtraction.



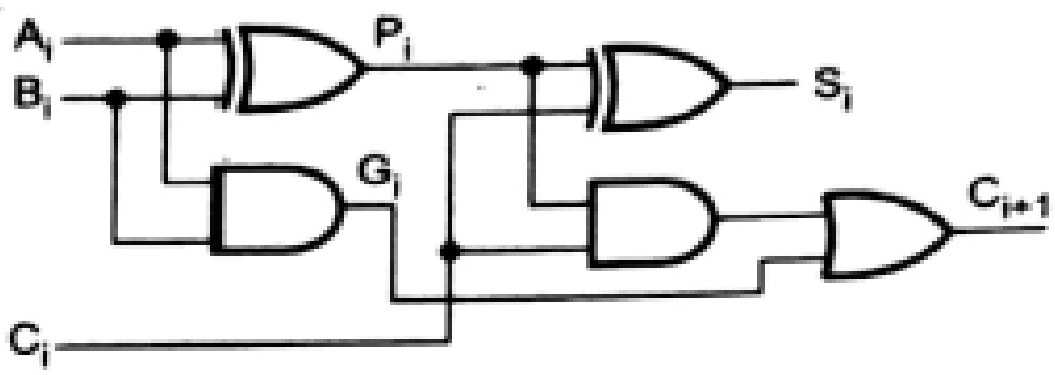
- When M= 1, the circuit is a subtractor and when M=0, the circuit becomes adder.
- The Ex-OR gate consists of two inputs to which one is connected to the B and other to input M.
- When M = 0, B Ex-OR of 0 produce B.
- Then full adders add the B with A with carry input zero and hence an addition operation is performed.
- When M = 1, B Ex-OR of 1 produce B complement and also carry input is 1.
- Hence the complemented B inputs are added to A and 1 is added through the input carry, nothing but a 2's complement operation.
- Therefore, the subtraction operation is performed.
- **The parallel adder is ripple carry adder in which the carry output of each full-adder stage is connected to the carry input of the next higher order stage. Therefore, the sum and carry outputs of any stage cannot be produced until the input carry occurs; this leads to a time delay in the addition process. This delay is known as carry-propagation delay.**
- The method of speeding up the process of parallel adder by eliminating inter stage carry delay is called look ahead-carry addition. This method utilizes logic gates to look at the lower-order bits of the augend and addend to see if a higher-order carry is to be generated.

Look-Ahead Carry adder

- It uses two functions : carry generate and carry propagate

  $P_i = A_i \oplus B_i$

  $G_i = A_i B_i$



- The output sum and carry can be expressed as,

  $S_i = P_i \oplus C_i$

  $C_{i+1} = G_i + P_i C_i$

- G_i is called a carry generate and it produces on carry when both A_i and B_i are one, regardless of the input carry.
- P_i is called a carry propagate because it is the term associated with the propagation of the carry from C_i to C_{i+1}

- Now the Boolean function for the carry output of each stage can be written as follows. $C_{i+1}=G_i + P_i C_i$

$C_2=G_1 + P_1 C_1$

$C_3=G_2 + P_2 C_2$

$=G_2 + P_2 (G_1 + P_1 C_1)$

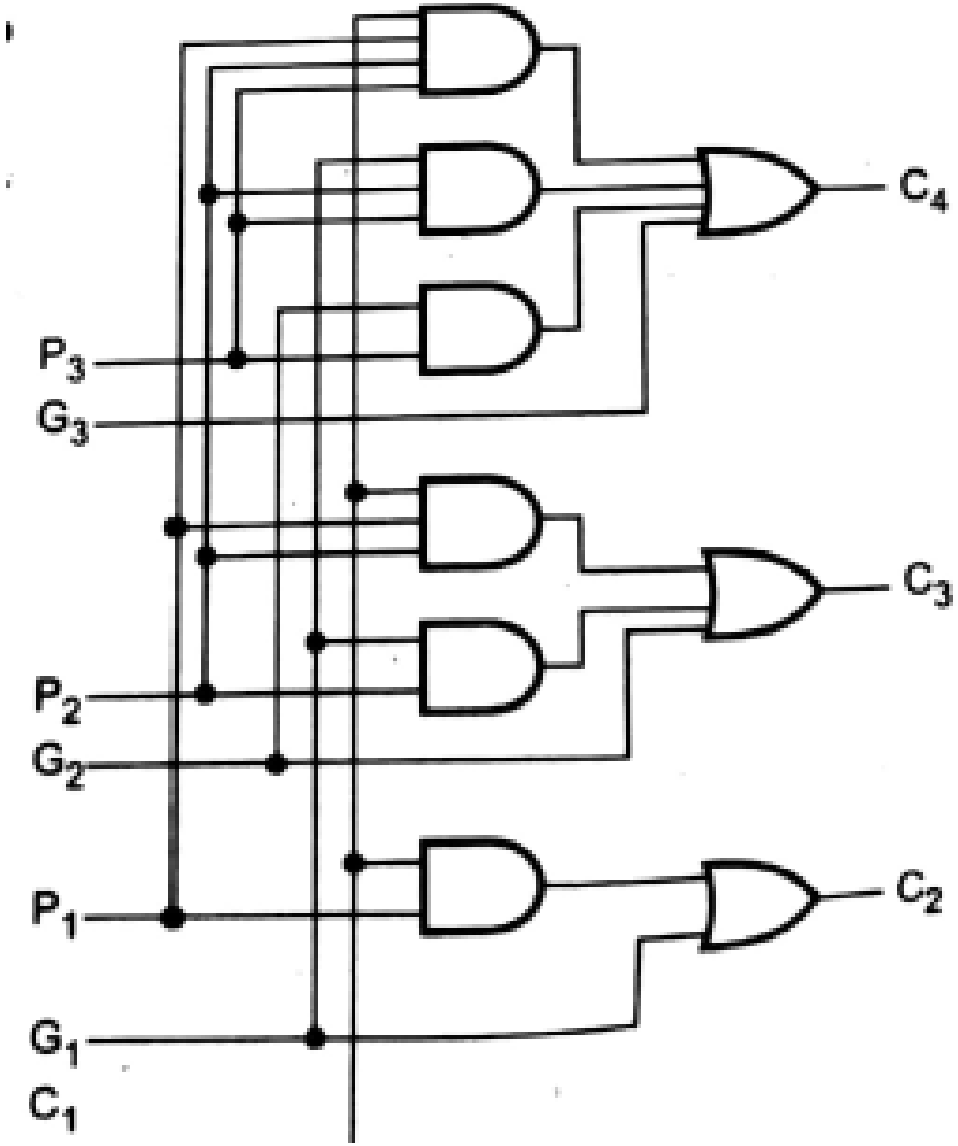
$=G_2 + P_2 G_1 + P_2P_1 C_1$

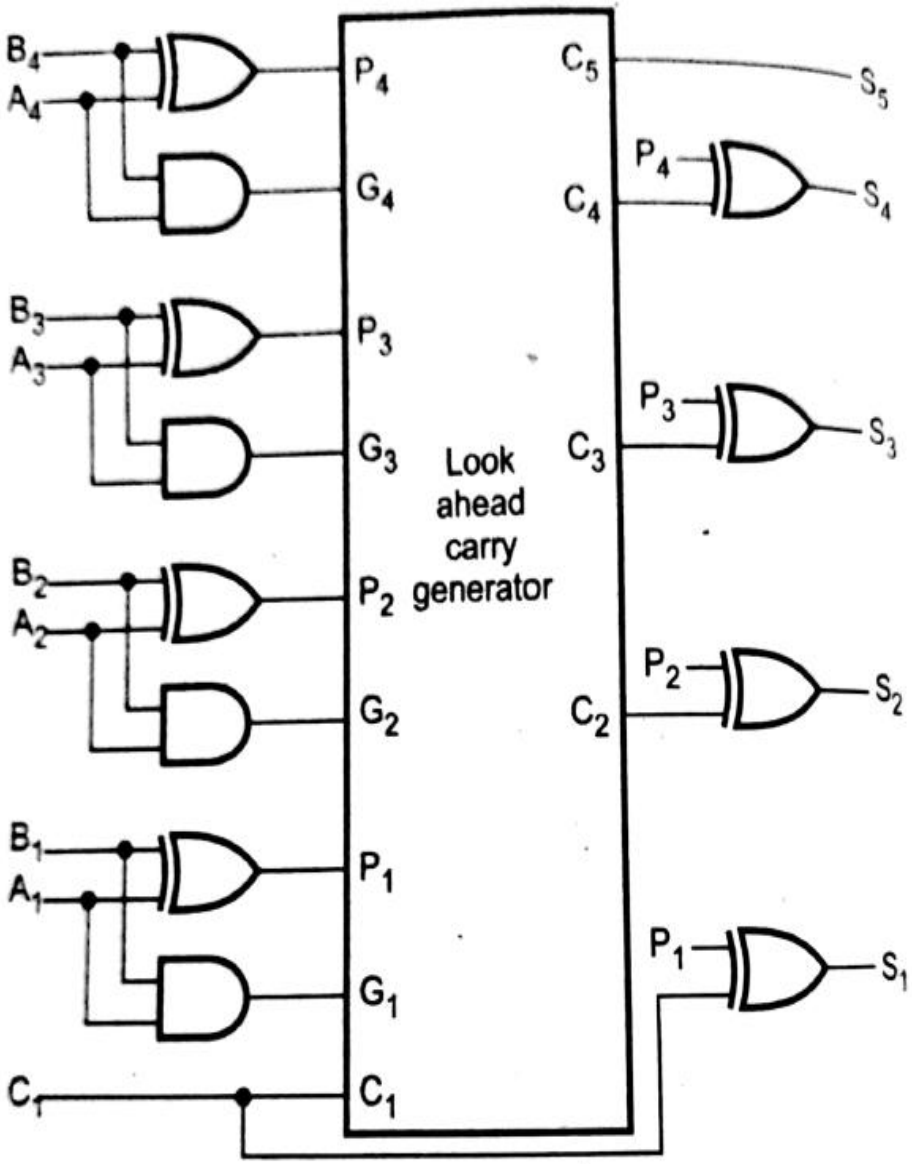
$C_4=G_3 + P_3 C_3$

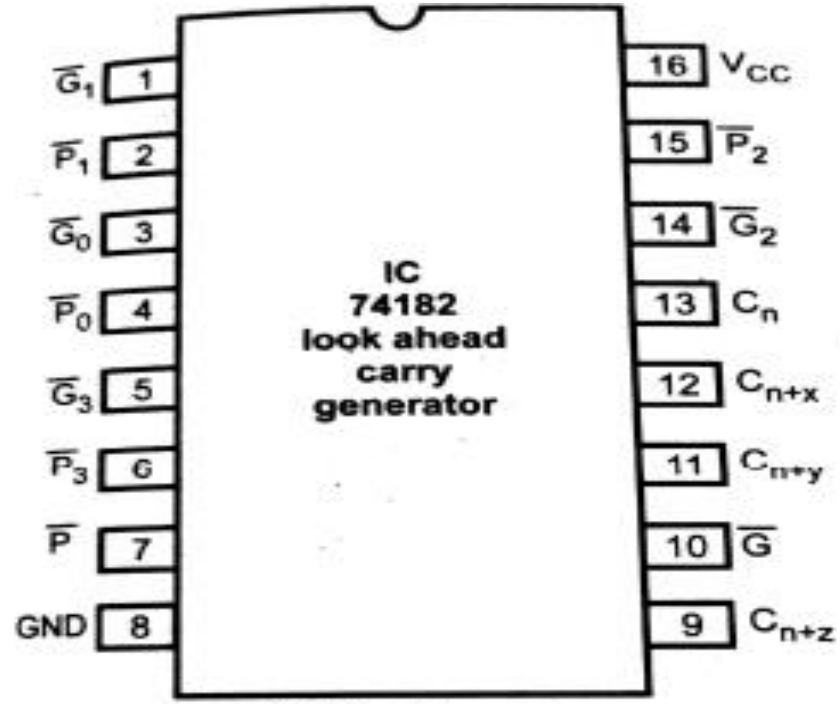
$=G_3 + P_3 (G_2 + P_2 G_1 + P_2P_1 C_1)$

$=G_3 + P_3G_2 + P_3P_2 G_1 + P_3P_2P_1 C_1$

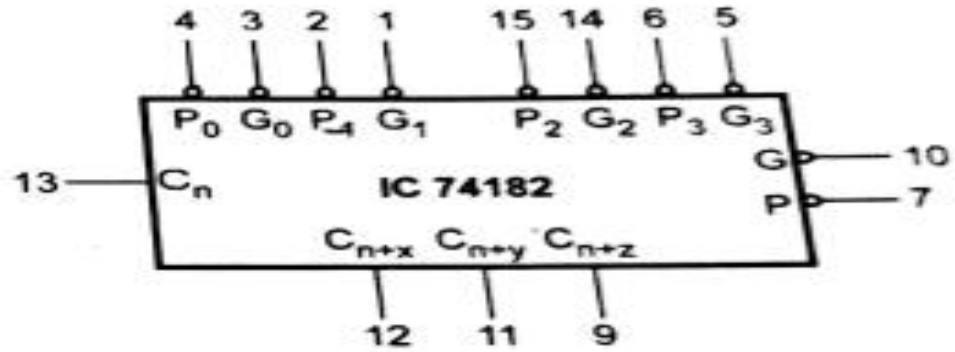
From the above Boolean function it can be seen that C_4 does not have to wait for C_3 and C_2 to propagate; in fact C_4 is propagated at the same time as C_2 and C_3







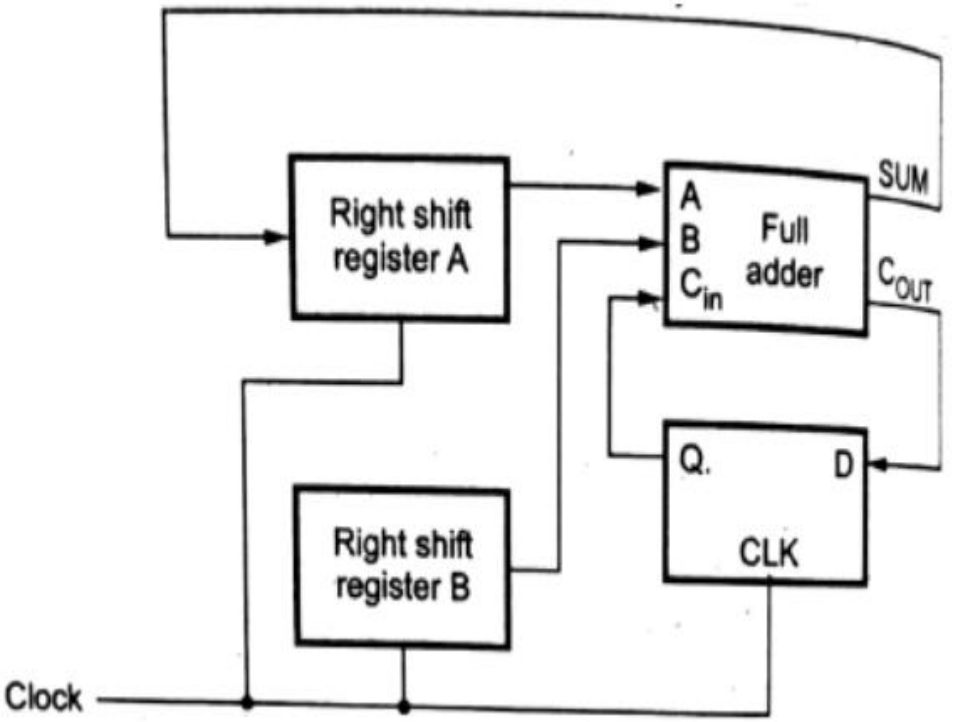
(a) Pin diagram



(b) Logic symbol

Serial Adder

- We can add numbers stored in the right shift registers A and B, serially.
- The full-adder is used to perform bit by bit addition and D-flipflop is used to store the carry output generated after addition.
- This carry is used as carry input for the next addition.
- Initially, the D-flipflop is cleared and addition starts with the least significant bits of both register.
- After each clock pulse data within the right shift registers are shifted right 1-bit and we get bits from next digit and carry of previous addition as new inputs for the full adder.
- The result SUM is stored bit by bit in the register A.
- We can implement serial subtractor by replacing full subtractor instead of full adder and thereby we get difference and borrow instead of sum and carry.



Comparison between serial and parallel adder

Serial adder	Parallel adder
• Uses shift register	• Uses registers with parallel load capacity
• Requires only one full adder circuit	• Number of full adder circuit equal to the number of bits in the binary number
• It is a sequential circuit	• Purely a combinational circuit
• Time required depends on number of bits	• Time required does not depend on number of bits
• It is slower	• It is faster

BCD Adder

- A BCD adder is a circuit that adds two BCD digits and produces a sum digit also in BCD.
- To implement BCD adder we require:
  - 4-bit binary adder for initial addition
  - Logic circuit to detect sum greater than 9
  - One more 4-bit adder to add 0110_2 in the sum if the sum is greater than 9 or carry is 1
  - The logic circuit to detect sum greater than 9 can be simplified by Boolean expression of given truth table.
- Y=1 indicates sum is greater than 9.
- We can put one more term, C_{out} in the above expression to check whether carry is one.
- If any one condition is satisfied we add 6(0110) in the sum.

Inputs				Outputs
S_3	S_2	S_1	S_0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

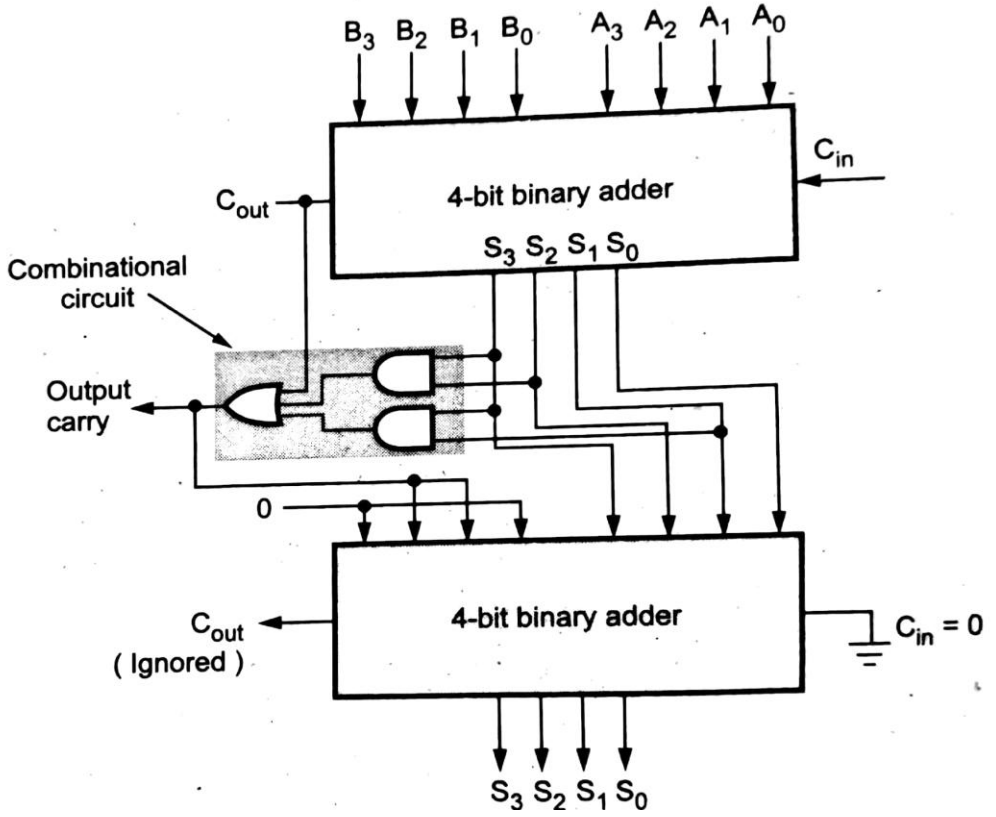
Truth Table

K-map Simplification

S_1S_0 / S_3S_2	$\overline{S_1}\,\overline{S_0}$ 00	$\overline{S_1}S_0$ 01	S_1S_0 11	$S_1\overline{S_0}$ 10
00 $\overline{S_3}\,\overline{S_2}$	0 0	0 1	0 3	0 2
01 $\overline{S_3}S_2$	0 4	0 5	0 7	0 6
11 S_3S_2	1 12	1 13	1 15	1 14
10 $S_3\overline{S_2}$	0 8	0 9	1 11	1 10

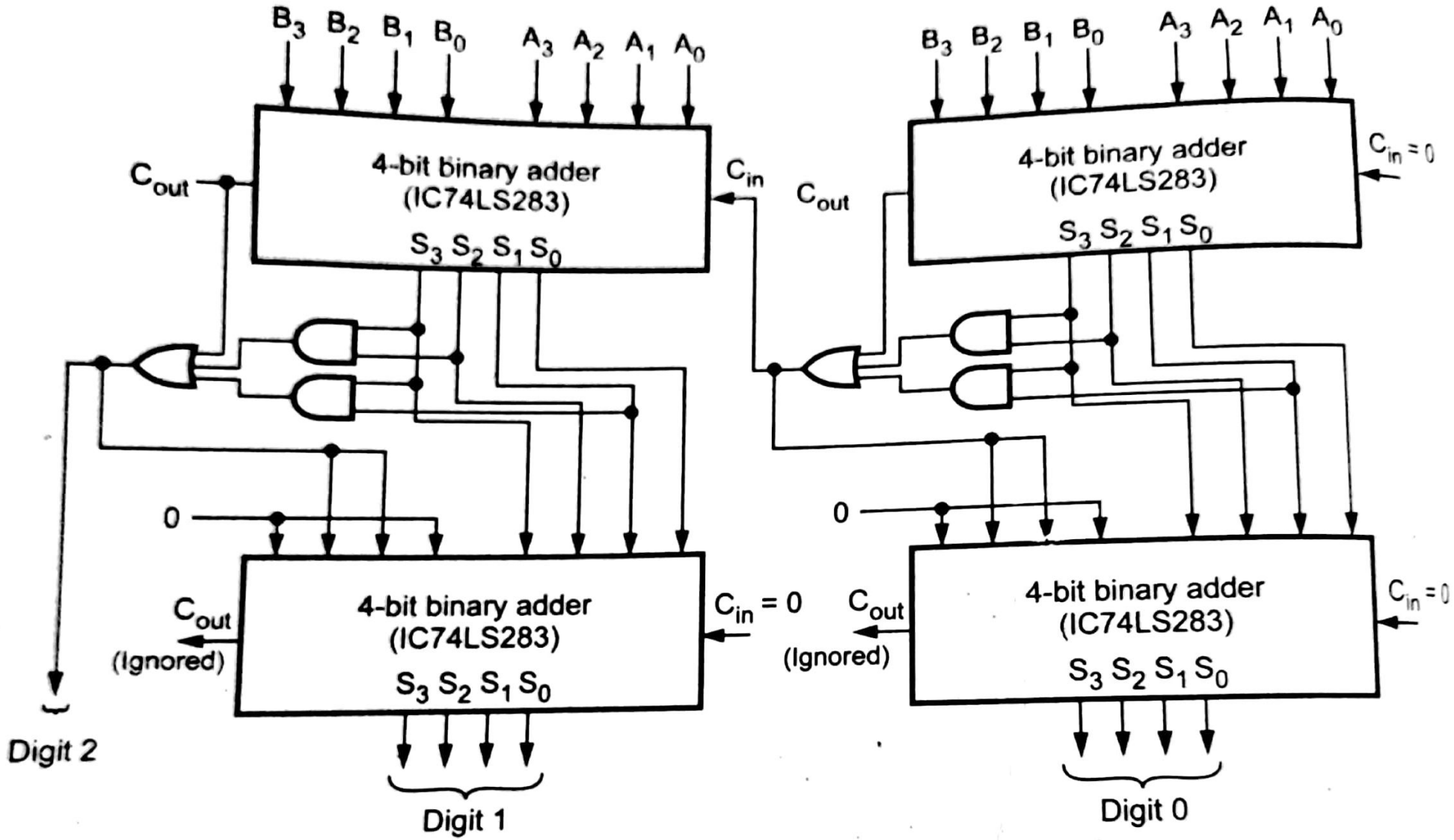
$$Y = S_3S_2 + S_3S_1$$

Block diagram of BCD adder



- The two BCD numbers, together with input carry, are first added in the top 4-bit binary adder to produce a binary sum.
- When the output carry is equal to zero (i.e., when sum ≤ 9 and C_{out} =0) nothing (zero) is added to the binary sum.
- When the output carry is equal to one (i.e., when sum > 9 and C_{out} =1) binary 0110 is added to the binary sum through the bottom 4-bit binary adder.
- The output carry generated from the bottom binary adder can be ignored, since it supplies information already available at the output-carry terminal.

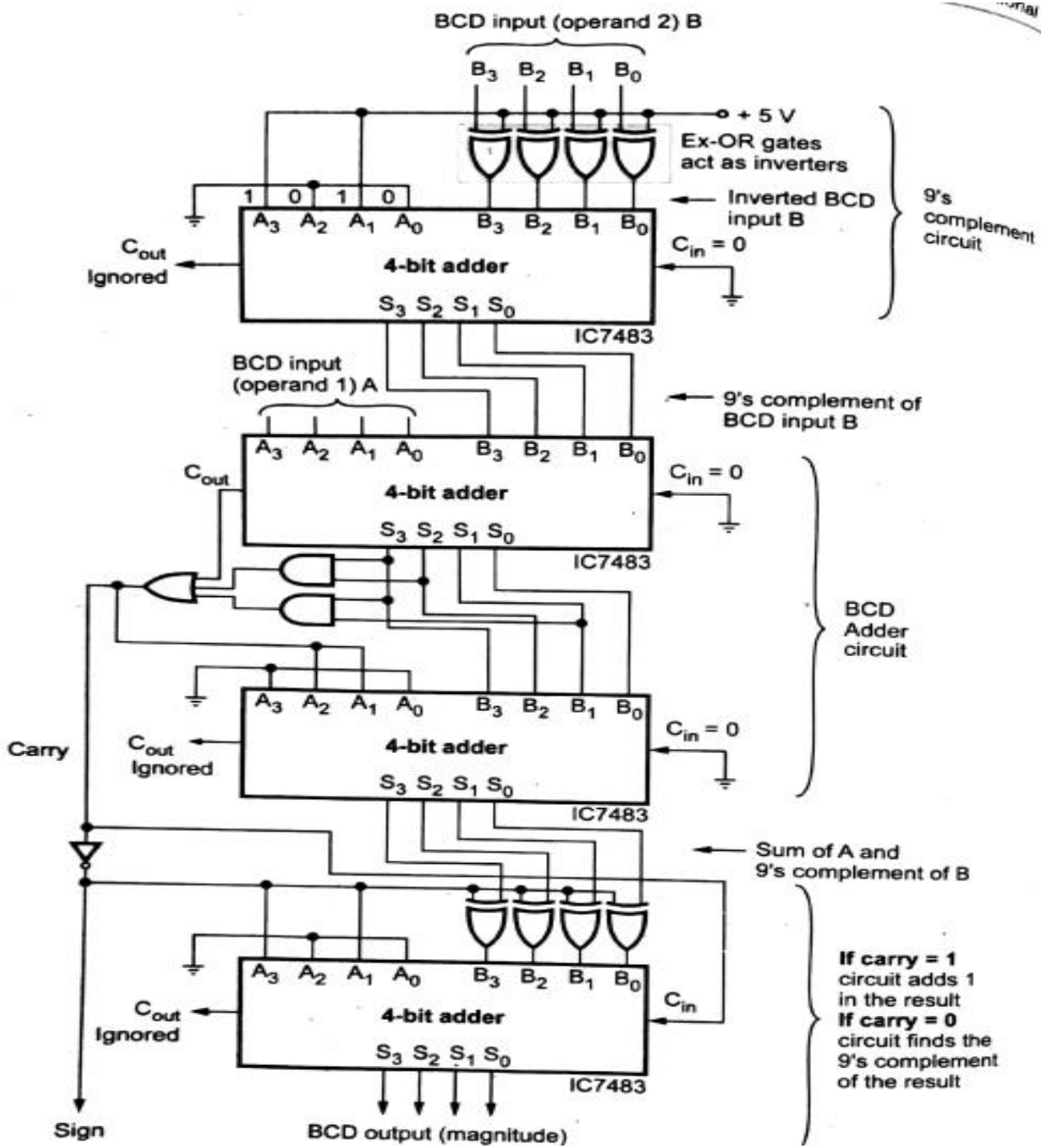
Design an 8-bit BCD adder using 4-bit binary adder.



BCD Subtractor

Subtractor using 9’s Complement method:

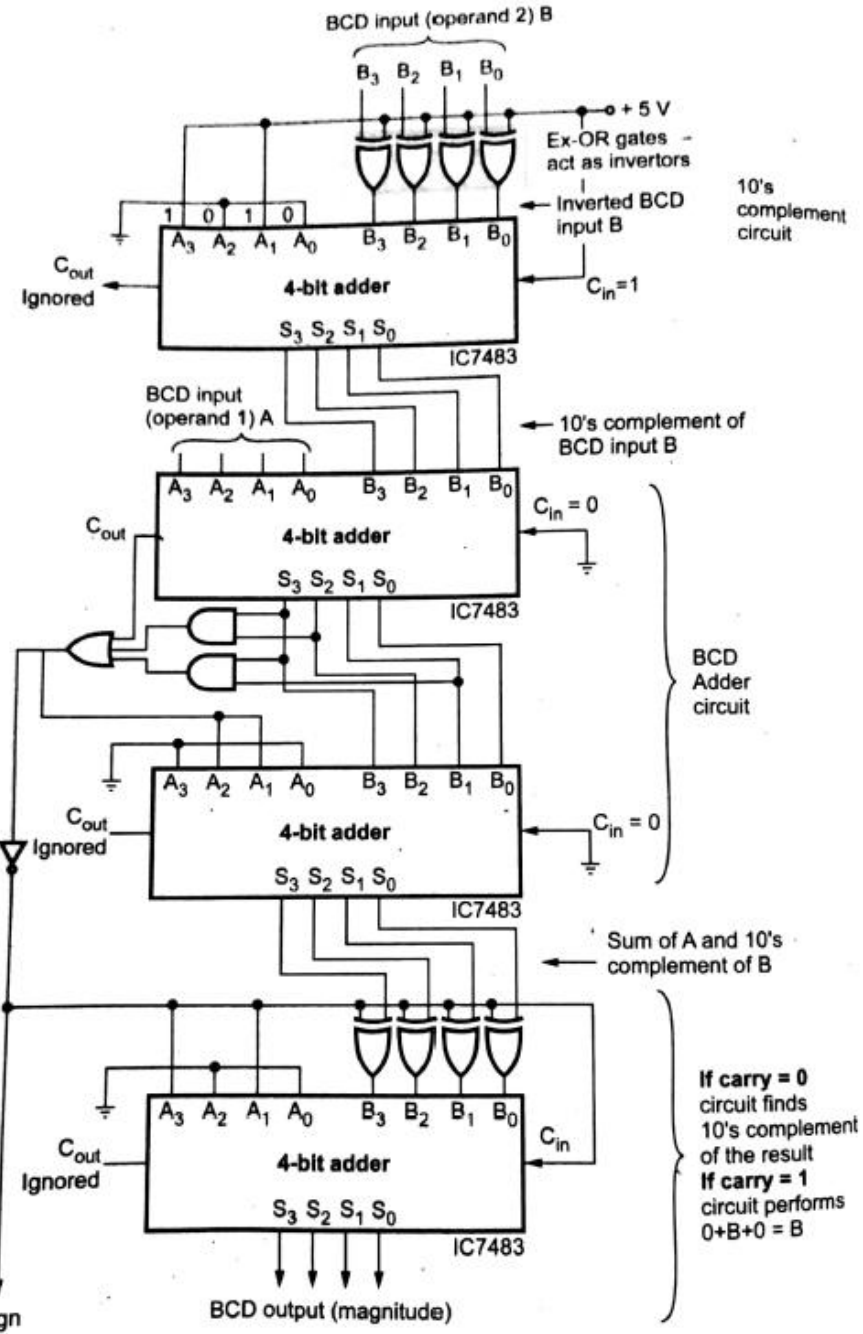
- The steps for 9’s complement BCD subtraction is as follows:
- Find the 9’s complement of the negative number → it is done by inverting each bit of BCD number and adding 10(1010) to it.
- Add two numbers using BCD addition
- If carry is generated add carry to the result otherwise find the 9’s complement of the result.


BCD input (operand 2) B
B_3 B_2 B_1 B_0
+ 5 V
Ex-OR gates act as inverters
1 0 1 0
Inverted BCD input B
9's complement circuit
A_3 A_2 A_1 A_0 B_3 B_2 B_1 B_0
C_{out} Ignored
C_{in} = 0
4-bit adder
S_3 S_2 S_1 S_0
IC7483
BCD input (operand 1) A
9's complement of BCD input B
BCD Adder circuit
Carry
Sum of A and 9's complement of B
If carry = 1 circuit adds 1 in the result
If carry = 0 circuit finds the 9's complement of the result
Sign
BCD output (magnitude)

Subtractor using 10’s Complement method:

The steps for 10’s complement BCD subtraction is as follows:

- Find the 10’s complement of the negative number → (9’s complement+1)
- Add two numbers using BCD addition
- If carry is not generated find the 10’s complement of the result.


BCD input (operand 2) B
B_3 B_2 B_1 B_0
+ 5 V
Ex-OR gates act as invertors
Inverted BCD input B
10's complement circuit
1 0 1 0
A_3 A_2 A_1 A_0 B_3 B_2 B_1 B_0
C_{out} Ignored
4-bit adder
S_3 S_2 S_1 S_0
IC7483
C_{in}=1
BCD input (operand 1) A
10's complement of BCD input B
C_{in} = 0
BCD Adder circuit
Sum of A and 10's complement of B
If carry = 0 circuit finds 10's complement of the result
If carry = 1 circuit performs 0+B+0 = B
Sign
BCD output (magnitude)

Binary Multiplier:

- The multiplication process for binary numbers is similar to the decimal numbers.
- Actually binary multiplication is simple than decimal multiplication since it involves only 1s and 0s.
- Rules for binary multiplication:
    - 0*0=0
    - 0*1=0
    - 1*0=0
    - 1*1=1
- The combinational logic circuit implemented to perform such multiplication is called combinational multiplier or array multiplier.

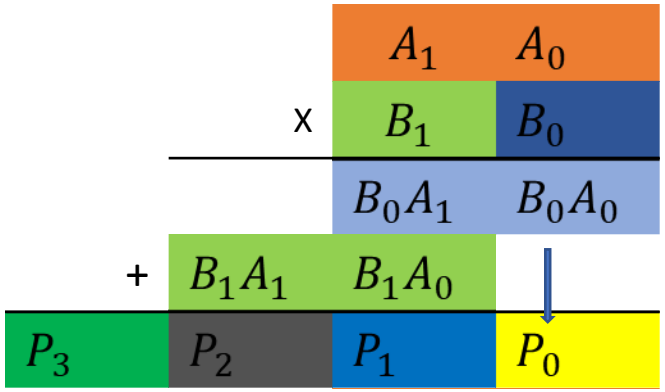
2 x 2 Multiplier:

Two unsigned 2-bit numbers: Multiplicand,A= A_1A_0 and multiplier B= B_1 B_0

The multiplication process involves multiplication(product) of 2-bit number and addition of 2-bit number.

The multiplication of 2-bits can be implemented using 2-input AND gate whereas addition of 2-bits can be implemented using half-adder.

Multiplication Process



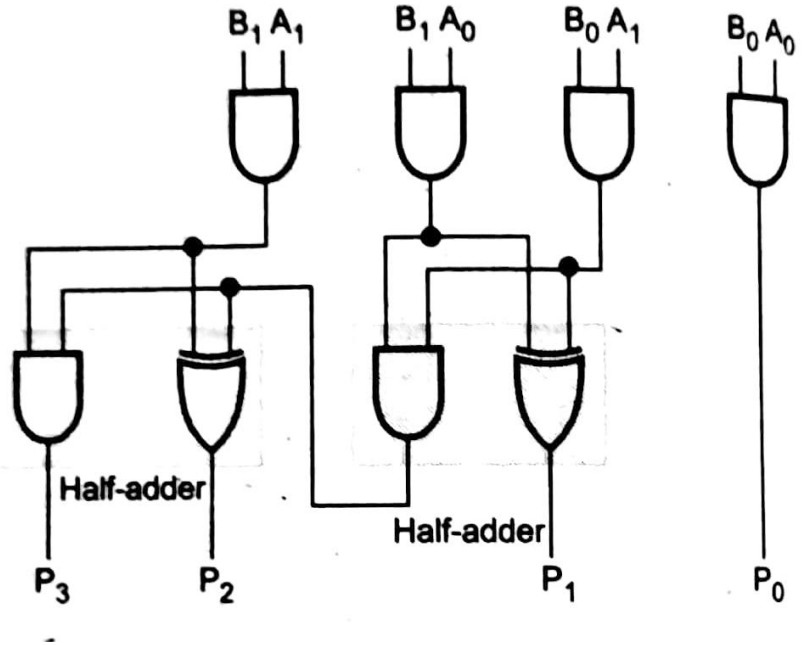
$P_0 = B_0A_0$

$P_1 = B_0A_1 + B_1A_0$

$P_2 = B_1A_1$ + carry output of P_1

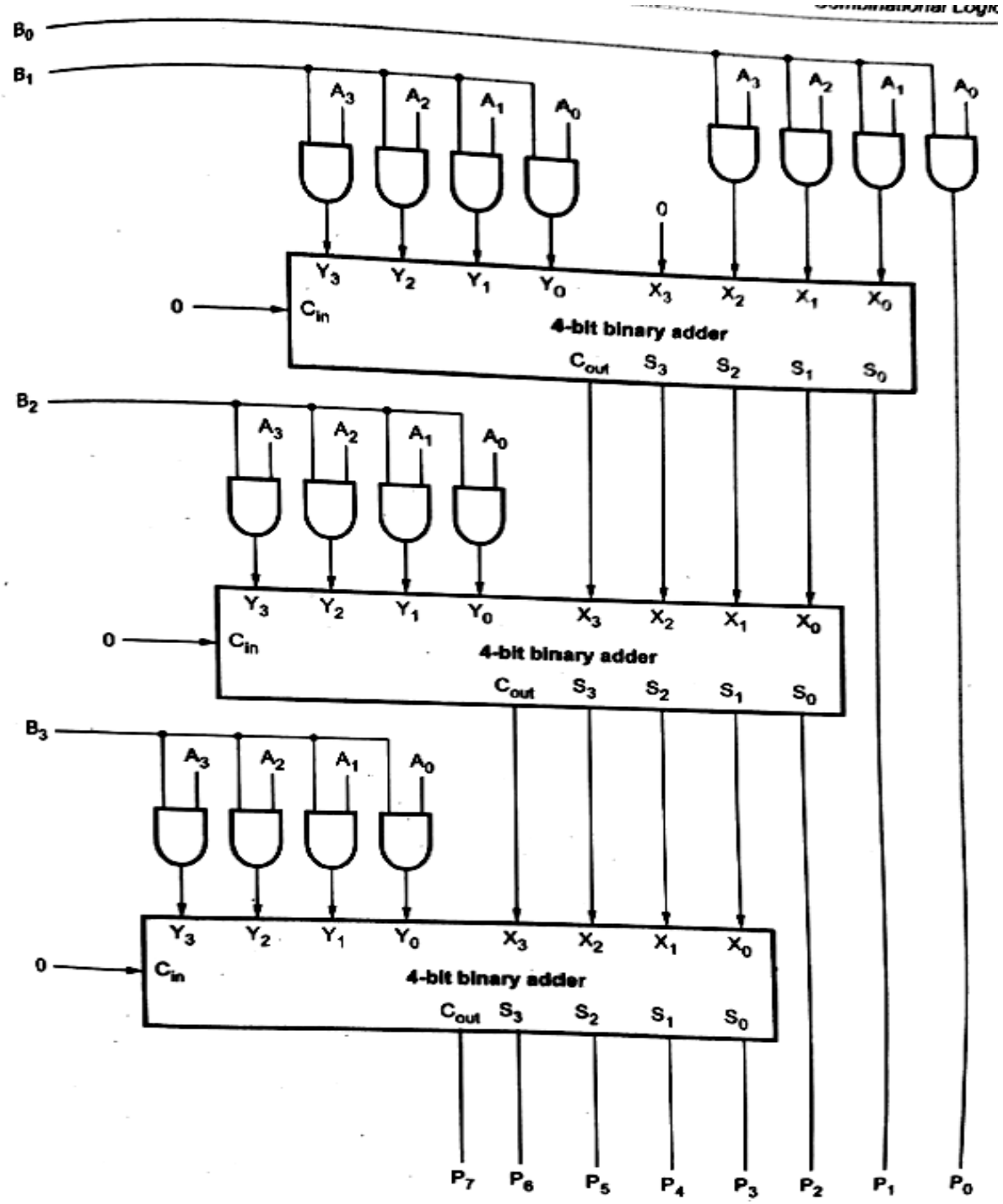
P_3 = carry output of P_2

2 x 2 bit combinational array multiplier

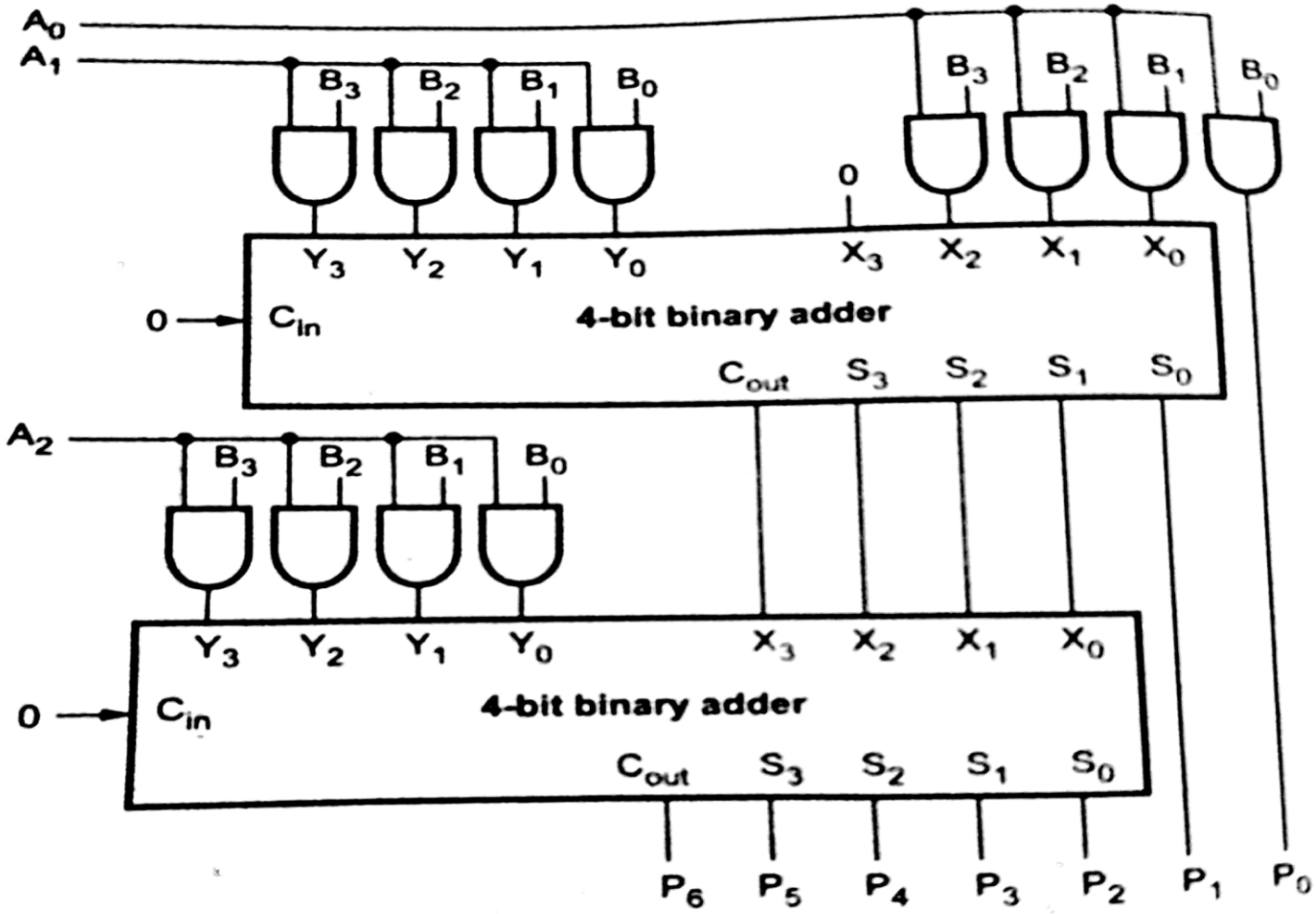


4 x 4 multiplier:

			A_3	A_2	A_1	A_0	
		x	B_3	B_2	B_1	B_0	
			B_0A_3	B_0A_2	B_0A_1	B_0A_0	
		B_1A_3	B_1A_2	B_1A_1	B_1A_0		
	B_2A_3	B_2A_2	B_2A_1	B_2A_0			
+	B_3A_3	B_3A_2	B_3A_1	B_3A_0			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

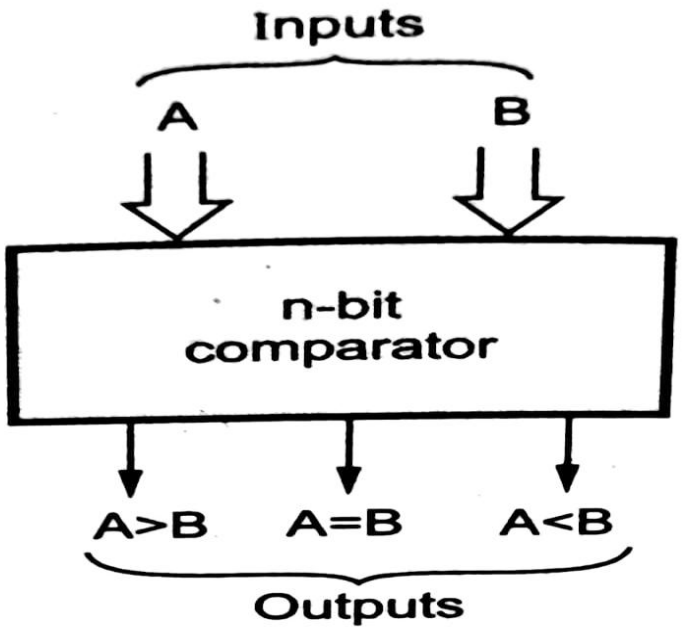


Design a multiple circuit to multiply the following binary number $A=A_0A_1A_2$ and $B=B_0B_1B_2B_3$



Magnitude comparator

- A comparator is a special combinational circuit designed primarily to compare the relative magnitude of two binary numbers.
- The n-bit comparator receives two n-bit numbers A and B as inputs and outputs are A>B, A=B and A<B.
- Depending upon the relative magnitudes of the two number, one of the outputs will be high.



Design a 1-bit comparator using basic gates.

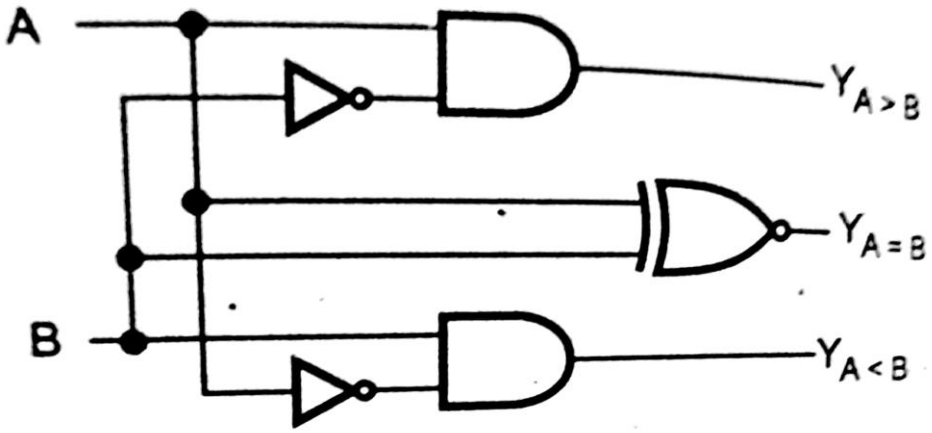
Solution:

Inputs		Outputs		
A	B	$Y_{A=B}$	$Y_{A>B}$	$Y_{A<B}$
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

$Y_{A=B}$

A \ B	0 $\overline{B}$	1 B
0 $\overline{A}$	1	0
1 A	0	1

$$Y_{A=B} = \overline{A}\,\overline{B} + AB = \overline{A \oplus B} = A \odot B$$



$Y_{A<B}$

A \ B	0 $\overline{B}$	1 B
0 $\overline{A}$	0	1
1 A	0	0

$$Y_{A<B} = \overline{A}B$$

$Y_{A>B}$

A \ B	0 $\overline{B}$	1 B
0 $\overline{A}$	0	0
1 A	1	0

$$Y_{A>B} = A\overline{B}$$

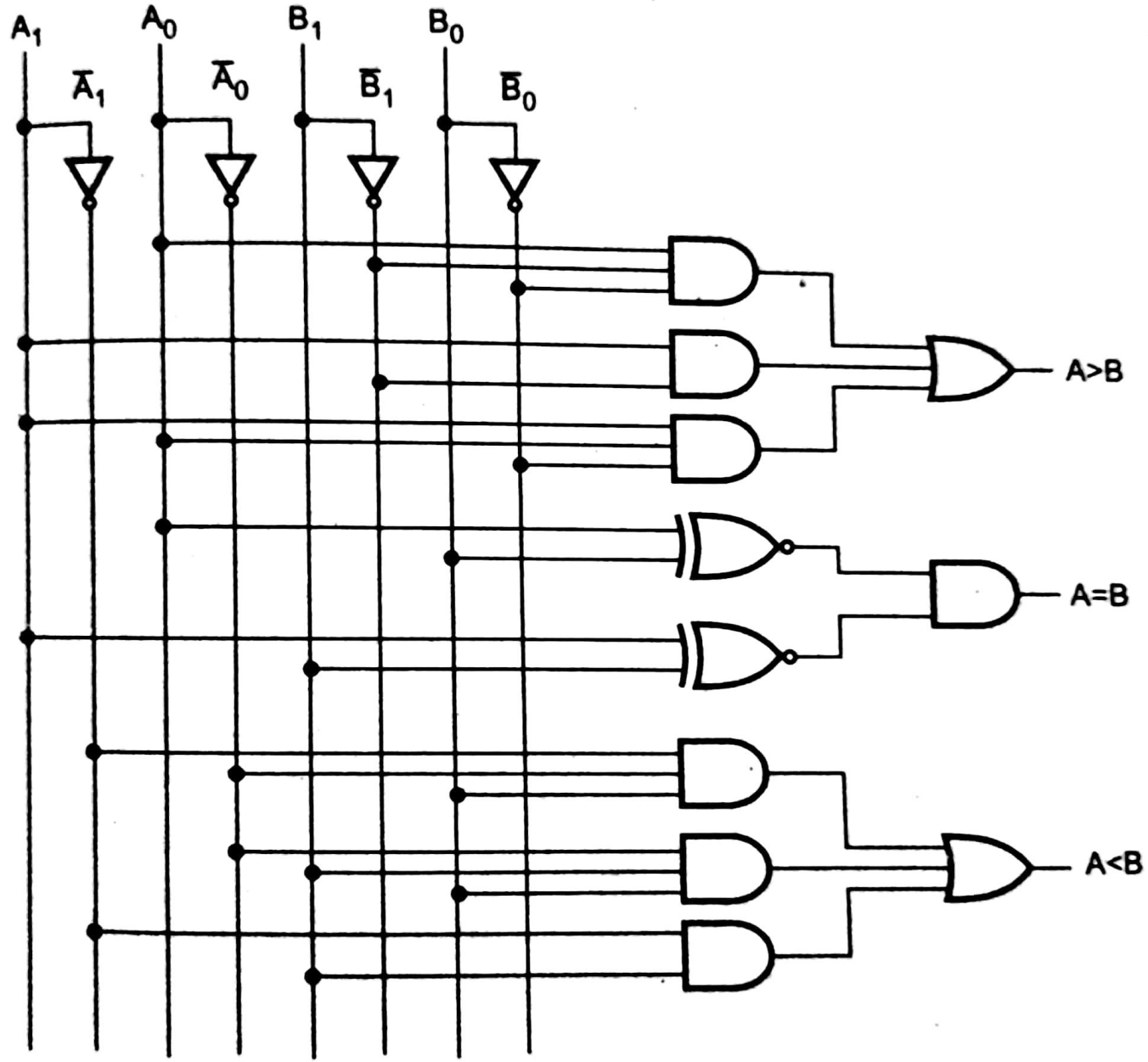
Design 2-bit comparator using gates.

Truth table:

Inputs				Outputs		
A_1	A_0	B_1	B_0	A>B	A=B	A<B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

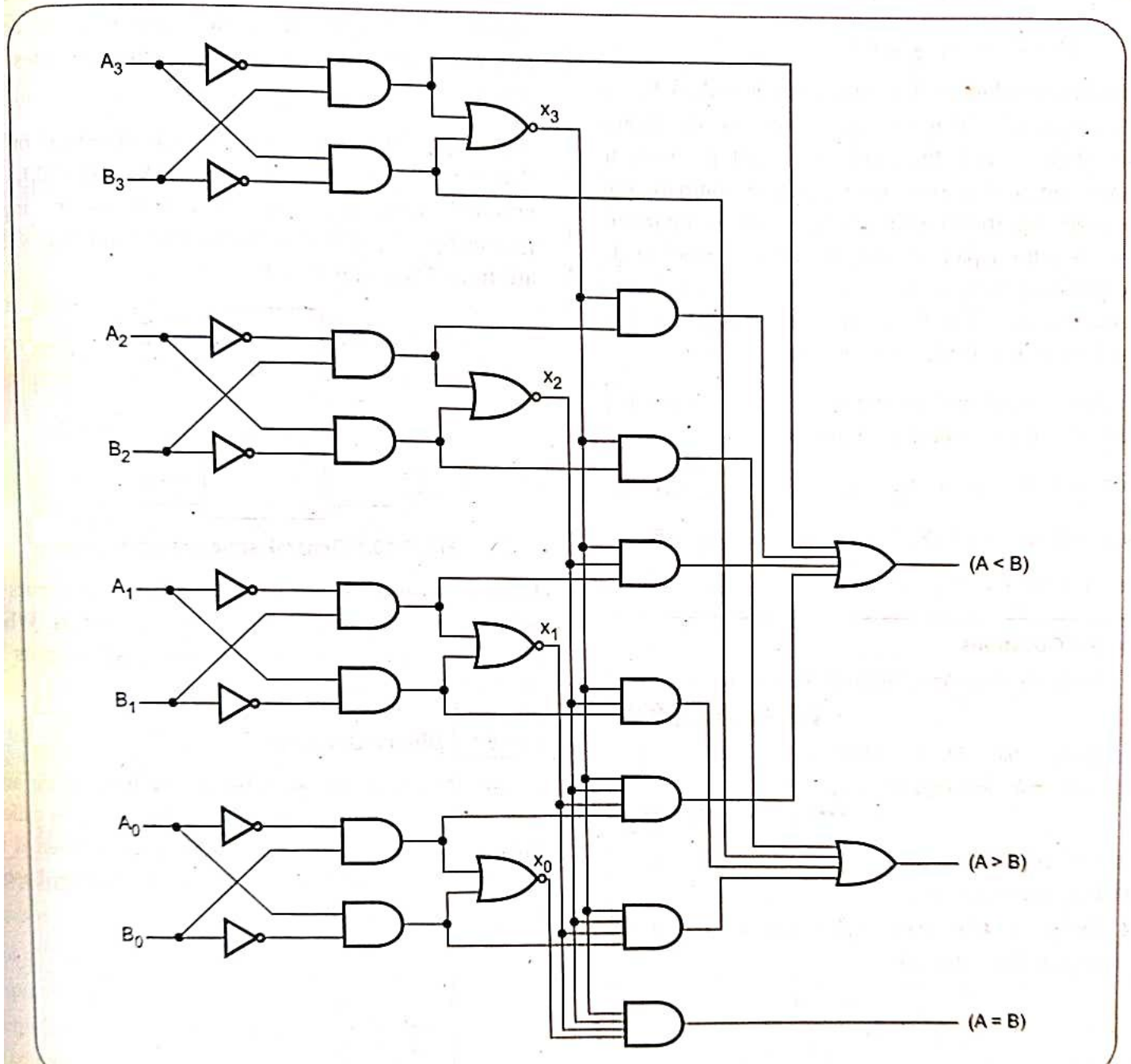
K-map Simplification:

B_1B_0 / A_1A_0	$\overline{B_1}\,\overline{B_0}$ 00	$\overline{B_1}B_0$ 01	B_1B_0 11	$B_1\overline{B_0}$ 10
00 $\overline{A_1}\,\overline{A_0}$	0	1	3	2
01 $\overline{A_1}A_0$	4	5	7	6
11 A_1A_0	12	13	15	14
10 $A_1\overline{A_0}$	8	9	11	10


A1
A0
B1
B0
A1
A0
B1
B0
A>B
A=B
A<B

.1 4-bit Magnitude Comparator

AU : May-12



Code Conversion

Design a 4-bit binary to BCD converter.

Step 1: Truth Table for the code converter

Binary code				BCD code				
B_3	B_2	B_1	B_0	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	0	0	0	0	1	0
0	0	1	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
0	1	0	1	0	0	1	0	1
0	1	1	0	0	0	1	1	0
0	1	1	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	0	0	1	0	1	0	0	1
1	0	1	0	1	0	0	0	0
1	0	1	1	1	0	0	0	1
1	1	0	0	1	0	0	1	0
1	1	0	1	1	0	0	1	1
1	1	1	0	1	0	1	0	0
1	1	1	1	1	0	1	0	1

Step 2: K-map simplification for each BCD output

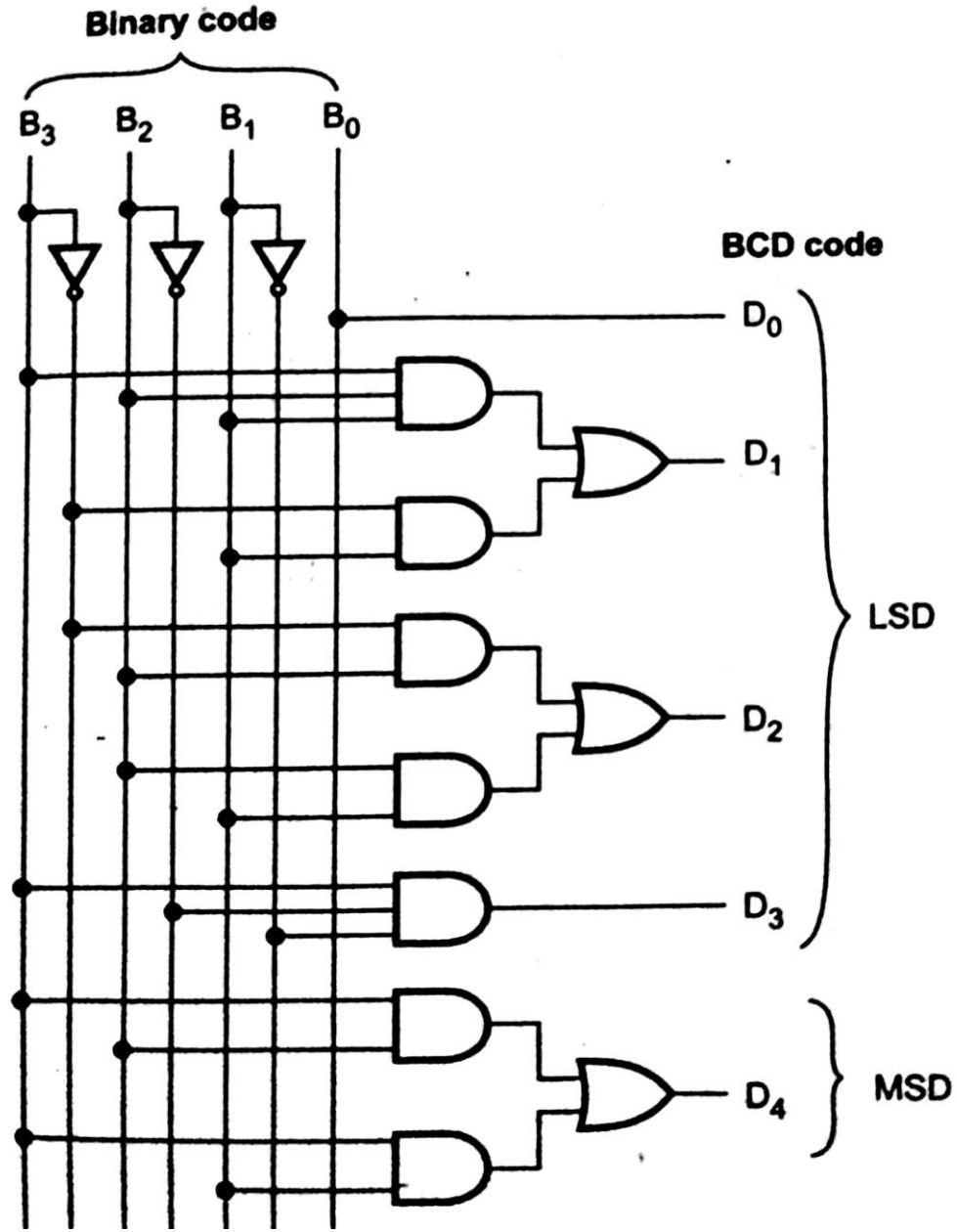
$$D_0 = B_0$$

$$D_1 = B_3B_2\overline{B_1} + \overline{B_3}\,B_1$$

$$D_2 = \overline{B_3}B_2 + B_2\,B_1$$

$$D_3 = B_3\overline{B_2}\,\overline{B_1}$$

$$D_4 = B_3B_2 + B_3B_1$$


Binary code
B3
B2
B1
B0
BCD code
D0
D1
D2
D3
D4
LSD
MSD

Design a logic circuit to convert BCD to gray code.

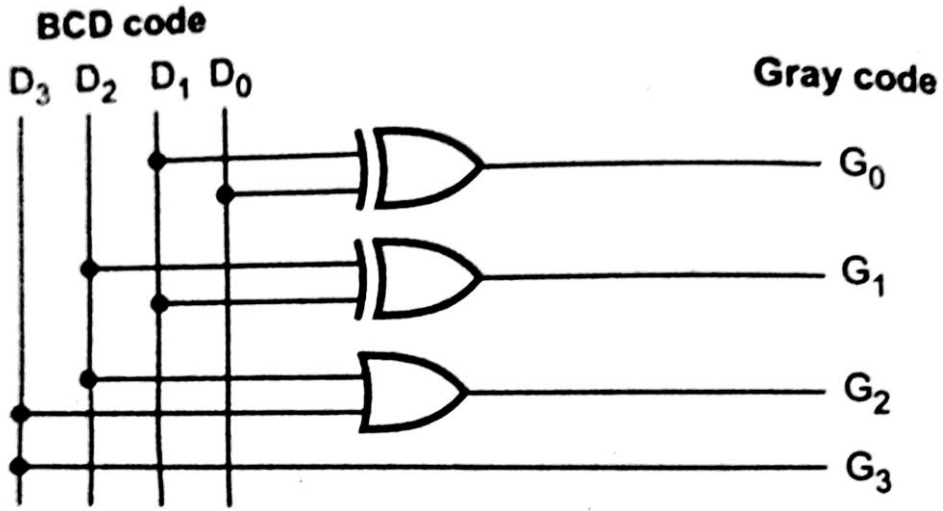
Step 1: Truth Table for the code converter

BCD code				Gray code			
D_3	D_2	D_1	D_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

Step 2: K-map simplification for each Gray output

$G_0 = D_1 \oplus D_0$
$G_1 = D_2 \oplus D_1$
$G_2 = D_2 + D_3$
$G_3 = D_3$

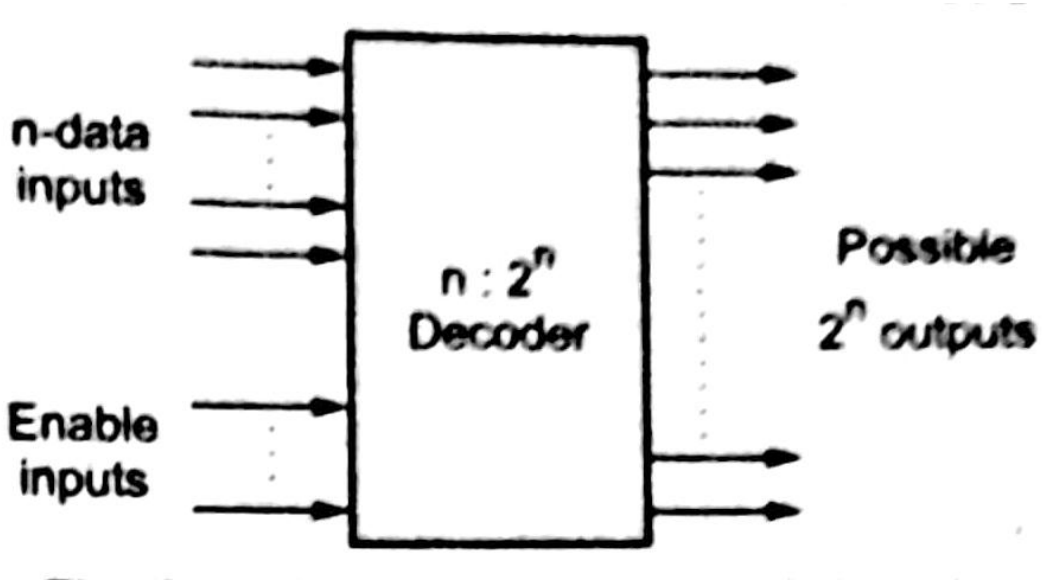
Logic diagram:



1. Design a logic circuit to convert the 8421 BCD to Excess-3 code.
2. Design and implement a 8421 to gray code converter. Realize using NAND gates only.
3. Design a gray to BCD code converter.

Decoders

- A decoder is a multiple-input, multiple-output logic circuit which converts coded inputs into coded outputs, where the input and output codes are different.
- The encoded information is presented as n inputs producing 2^n possible outputs.
- The 2^n output values are from 0 to 2^n -1.
- Usually, a decoder is provided with enable inputs to activate decoded output based on data inputs. When any one enable input is unasserted, all outputs of decoder are disabled.

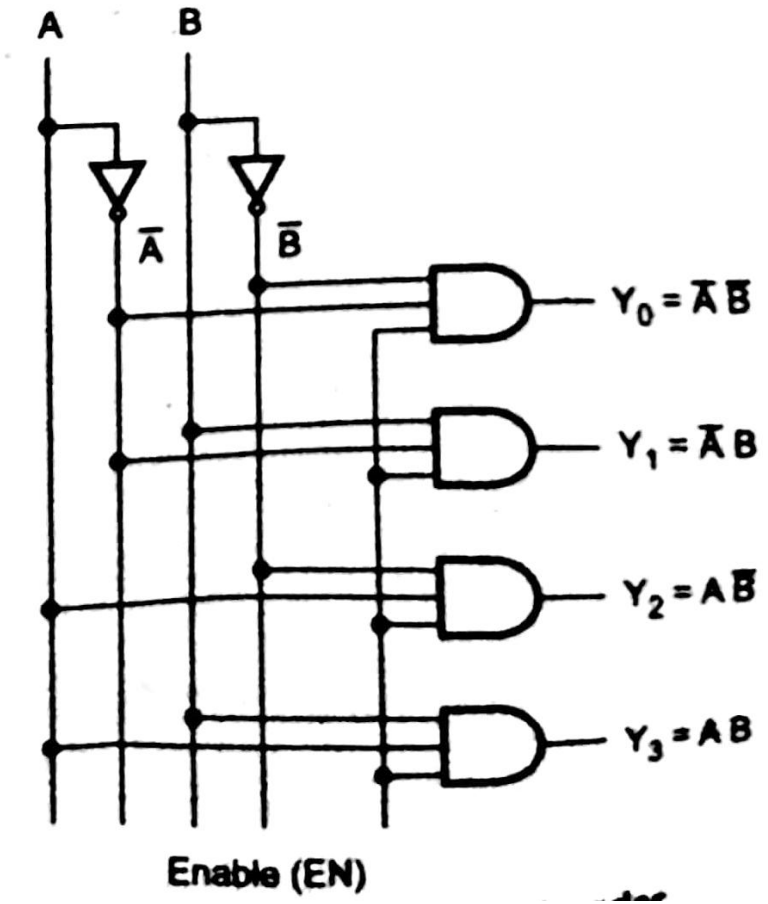


Binary Decoder

- A decoder which has an n-bit binary input code and a one activated output out of 2^n output code is called binary decoder.
- A binary decoder is used when it is necessary to activate exactly one of 2^n outputs based on an n-bit input value.

Inputs			Outputs			
EN	**A**	**B**	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

Truth Table for a 2 to 4 decoder

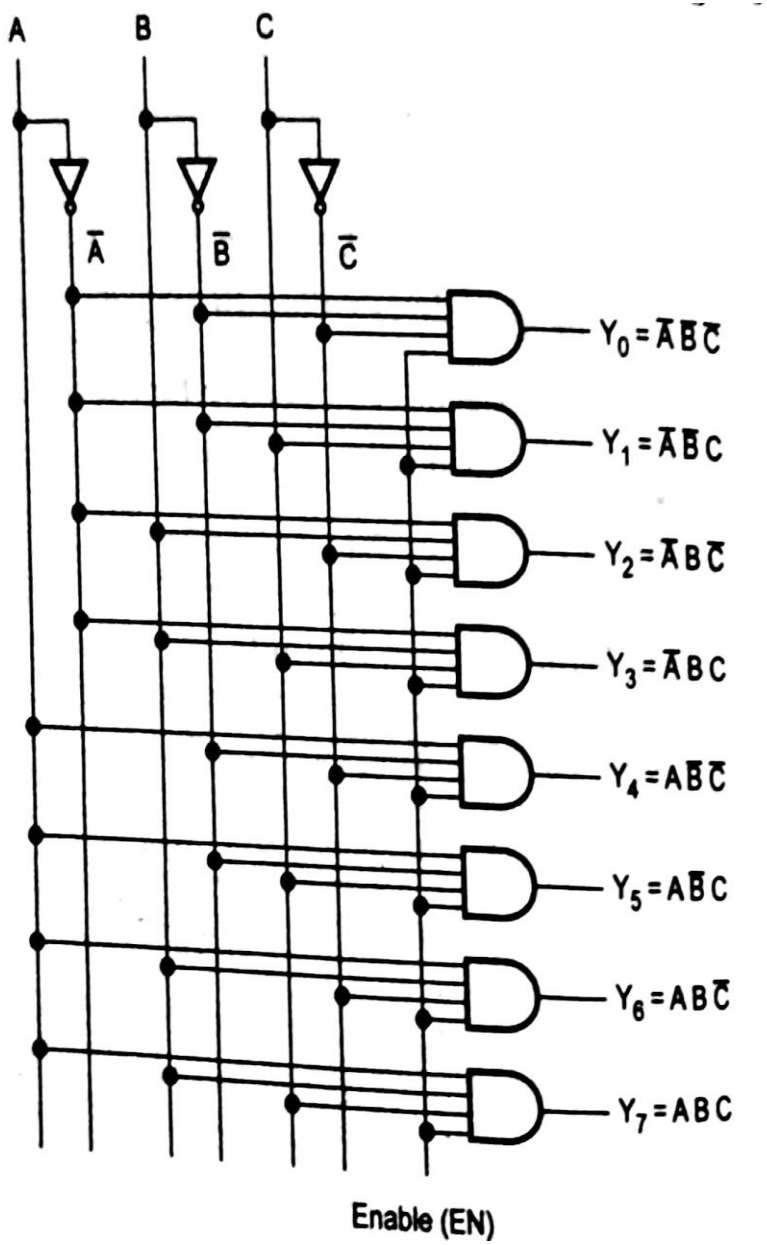


- 2 inputs are decoded into four outputs, each output representing one of the minterms of the 2 input variables.
- The two inverters provide the complement of the inputs, and each one of four AND gates generates one of the minterms.
- If enable input is 1(EN=1), one, and only one, of the outputs Y_0 to Y_3, is active for a given input.
- The output Y_0 is active, i.e. Y_0=1 when inputs A=B=0, the output Y_1 is active when inputs A=0 and B=1.
- If enable input is 0, i.e. EN=0, then all the outputs are 0.

Draw the circuit for 3 to 8 decoder and explain.

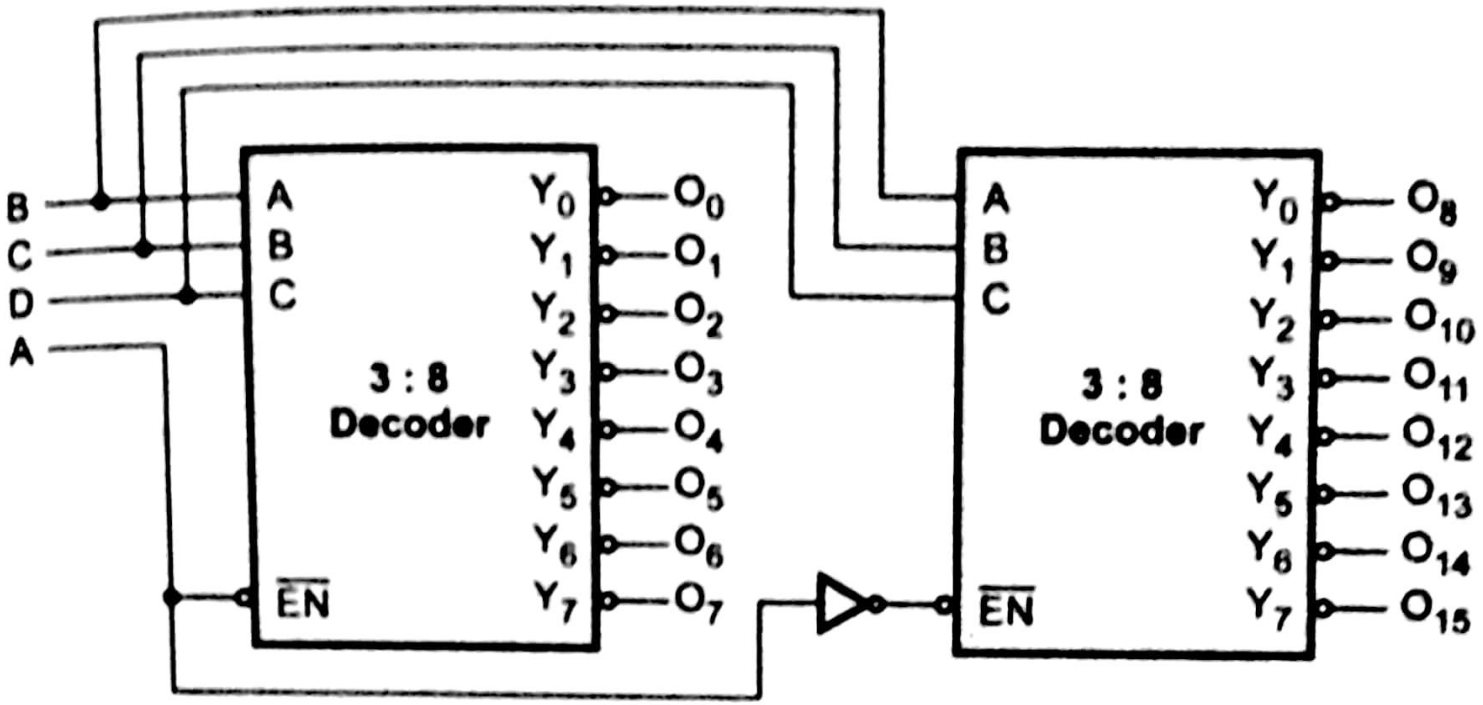
- In this, 3 inputs are decoded into eight outputs, each output represent one of the minterms of the 3 input variables.
- The three inverters provide the complement of the inputs, and each one of the eight AND gates generates one of the minterms.
- Enable input is provided to activate decoded output based on data inputs A,B and C.

Inputs				Outputs							
EN	**A**	**B**	**C**	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

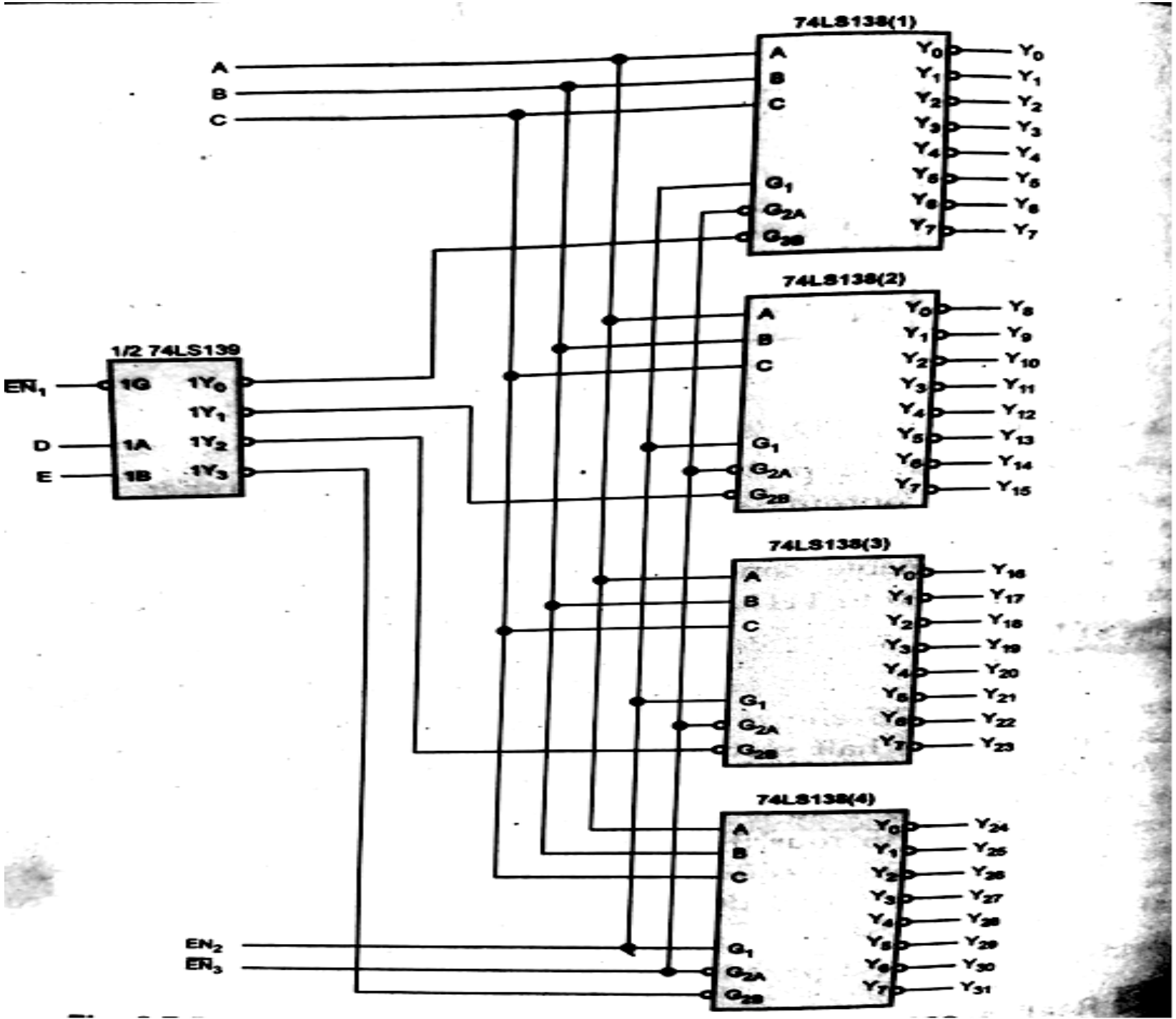


Expanding Cascading Decoders

- Binary decoder circuits can be connected together to form a larger decoder circuit.
- The figure shows 4x16 decoder using two 3x8 decoder.
- Here, one input line (D) is used to enable /disable the decoders.
- When D=0, the top decoder is enabled and the other is disabled.
- Thus the bottom decoder outputs are all 1s and the top eight outputs generate minterms 0000 to 0111.
- When D=1, the enable conditions are reversed and thus bottom decoder outputs generate minterms 1000 to 1111, while the outputs of the top decoder are all 1s.



Design 5-to-32 decoder using one 2-to-4 and four 3-to-8 decoder ICs.

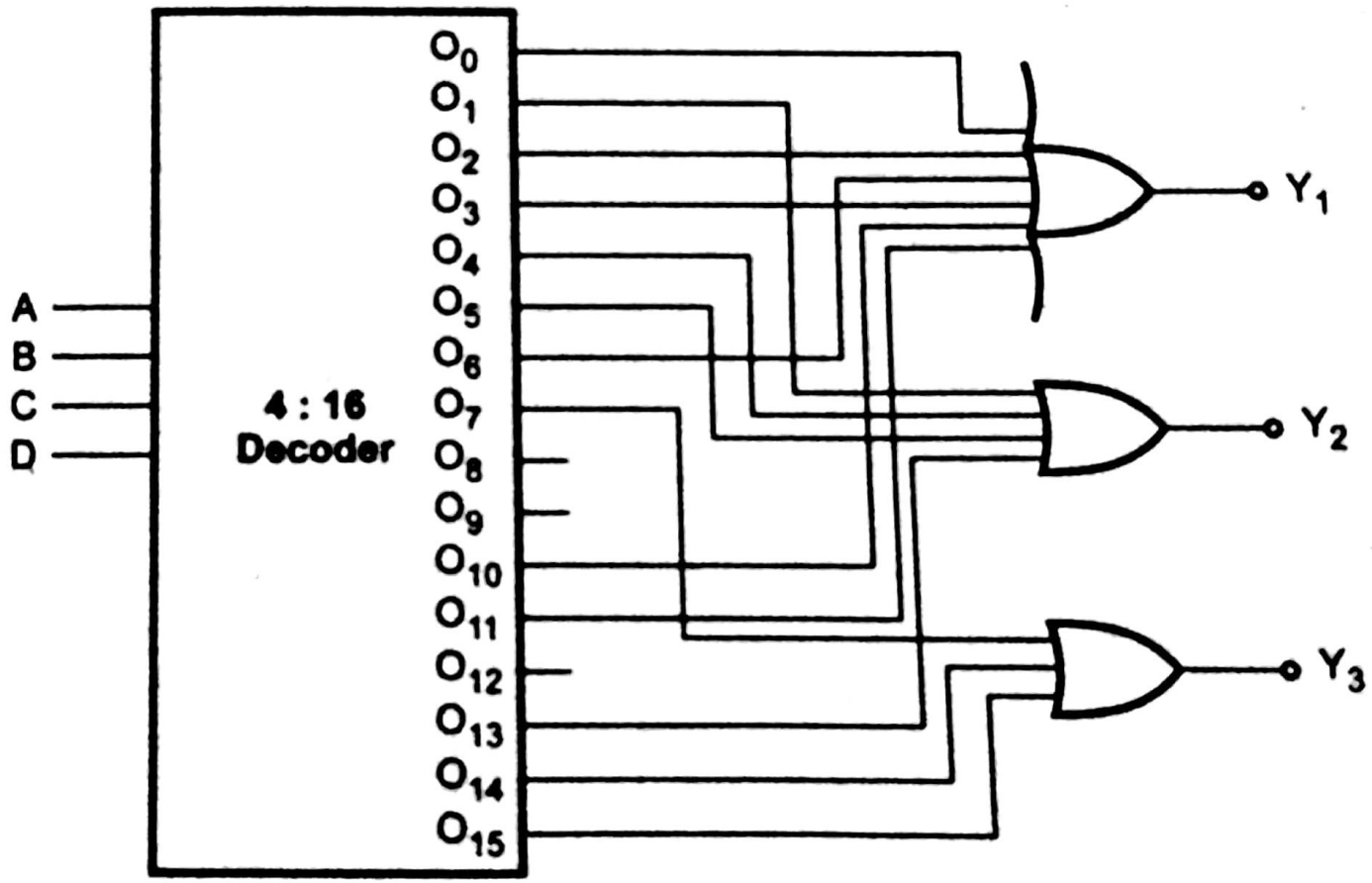


- Implement the following multiple output combinational logic using a 4 line to 16 line decoder.

$$Y_1 = \bar{A}\bar{B}\bar{C}\bar{D} + \bar{A}\bar{B}CD + \bar{A}B\bar{C}\bar{D} + \bar{A}BC\bar{D} + A\bar{B}\bar{C}\bar{D} + A\bar{B}CD$$

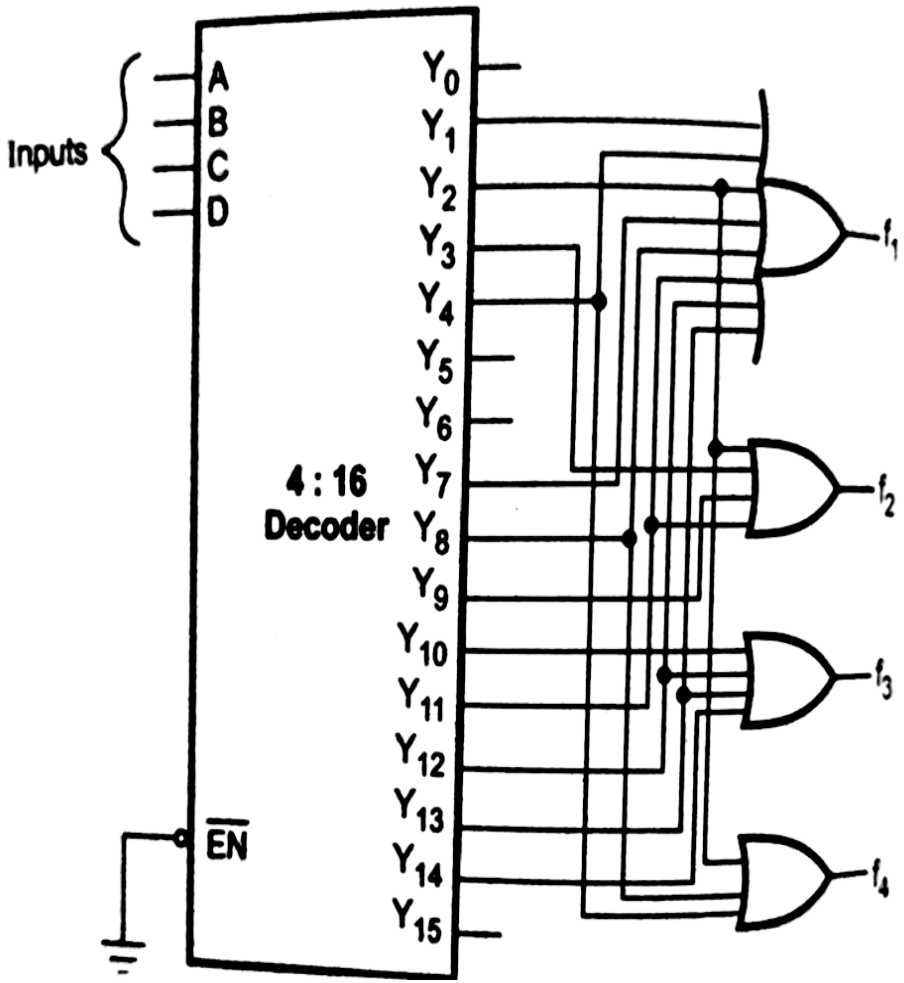
$$Y_2 = \bar{A}\bar{B}\bar{C}D + \bar{A}B\bar{C}\bar{D} + \bar{A}B\bar{C}D + AB\bar{C}D$$

$$Y_3 = \bar{A}BCD + ABC\bar{D} + ABCD$$



- Implement the following multiple output combinational logic function using a 4 line to 16 line decoder.

$$f_1 = \Sigma m(1,2,4,7,8,11,12,13)\ ; f_2 = \Sigma m(2,3,9,11); f_3 = \Sigma m(10,12,13,14)$$
$$f_4 = \Sigma m(2,4,8)$$

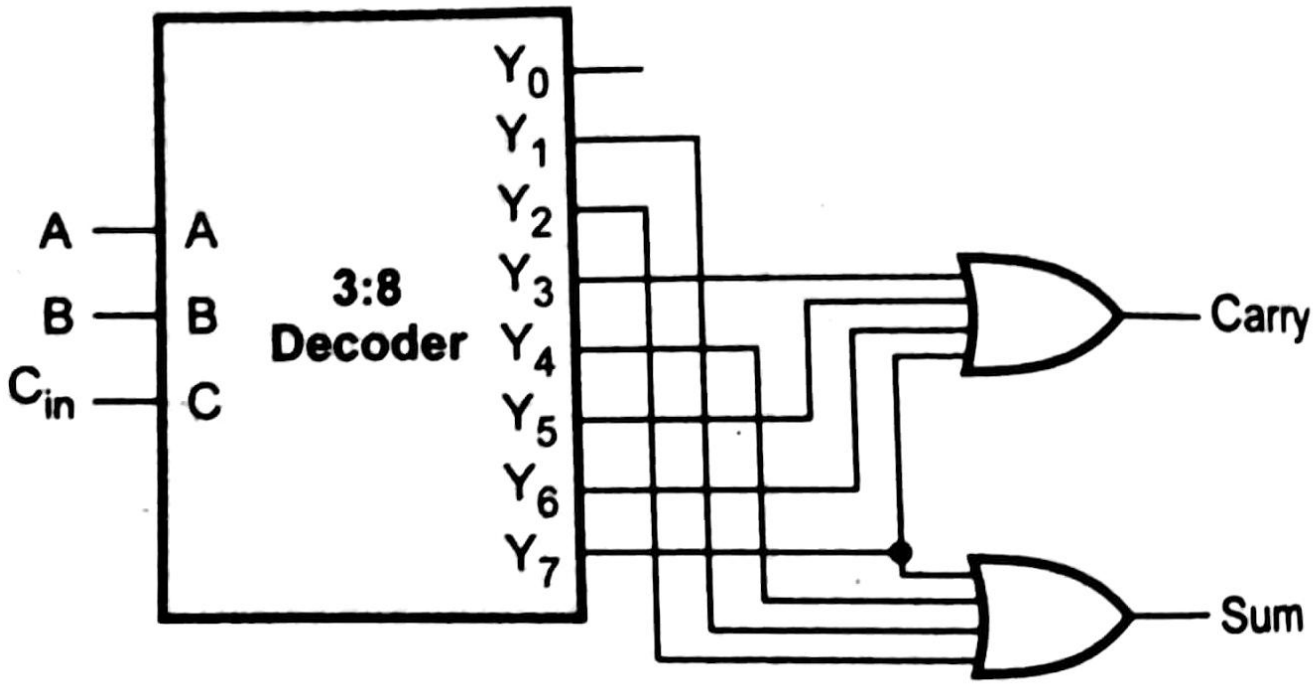


Design and implement a full adder circuit using a 3:8 decoder.

Solution:

Truth table of full adder

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



Applications of Decoders:

- Code converters
- Implementation of combinational circuits
- Address decoding
- BCD to 7-segment decoder

Decoder ICs:

3:8 Decoder – 74138

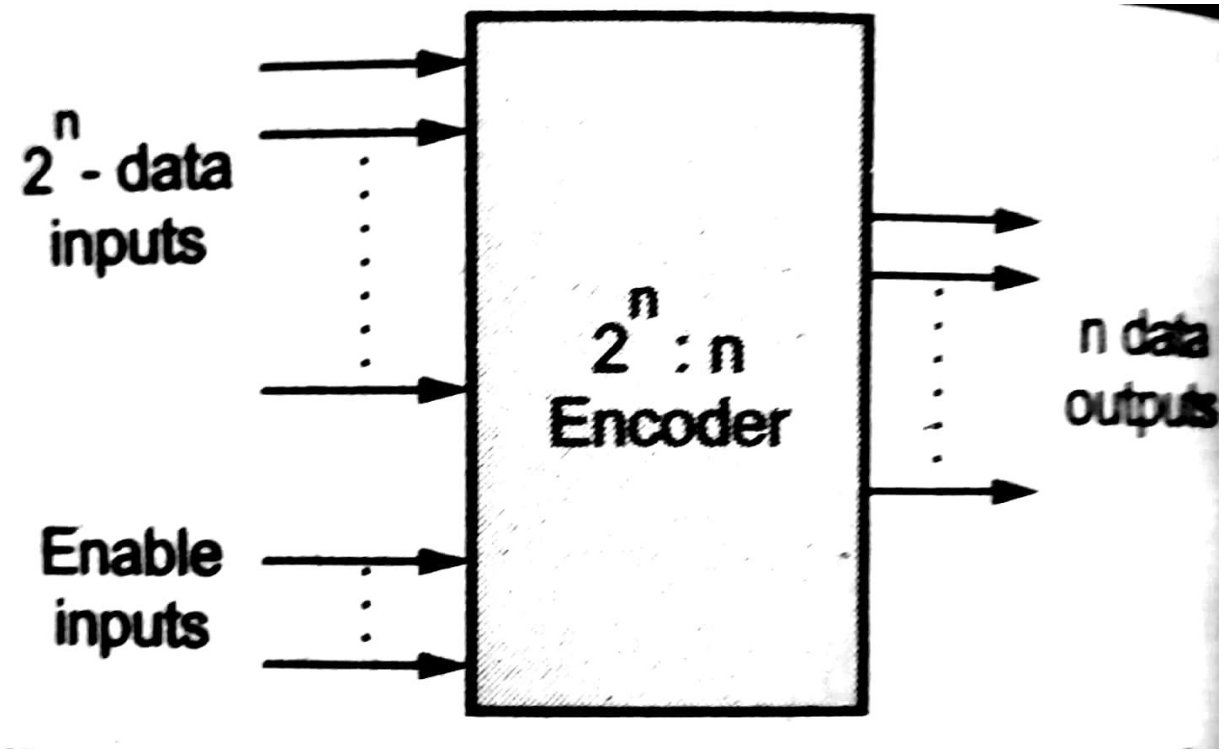
Dual 2:4 Decoder – 74139

BCD to decimal decoder – 7442

BCD to 7-segment decoder – 7447

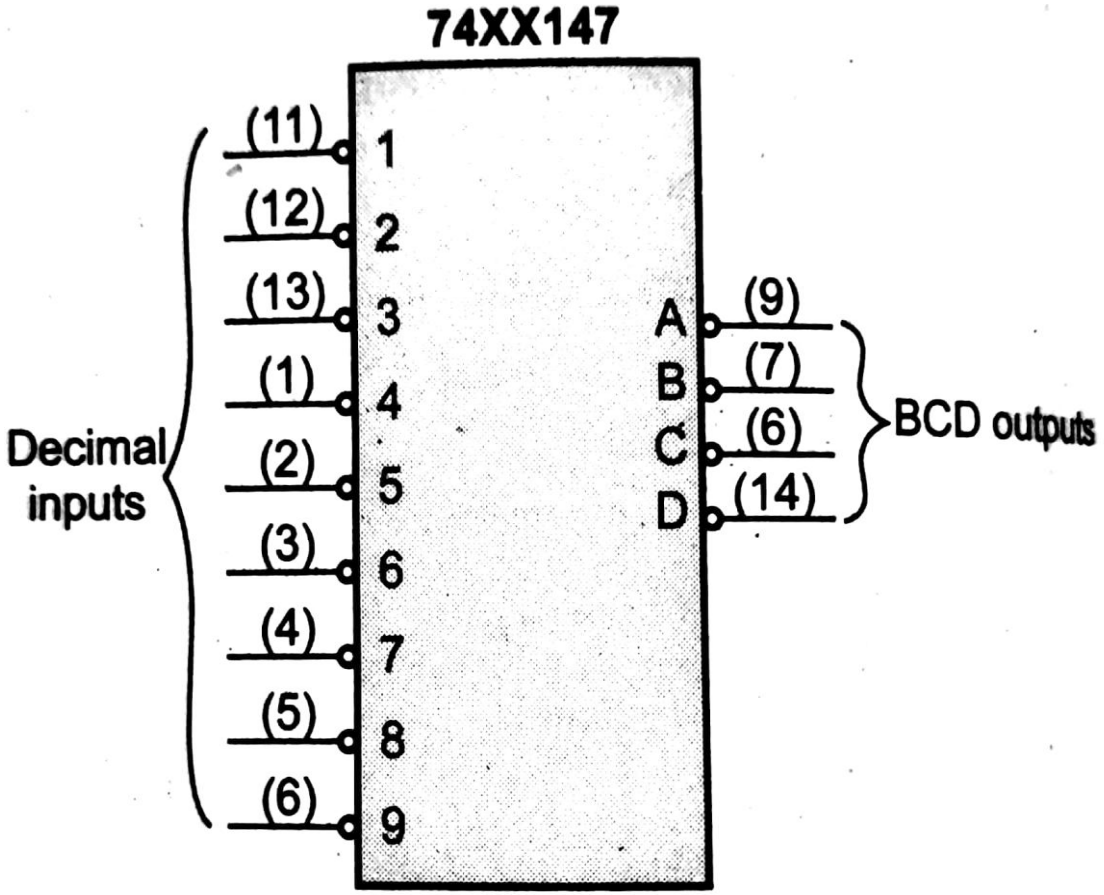
Encoders

- An encoders is a digital circuit that performs the inverse operation of a decoder.
- An encoder has 2^n (or fewer) input lines and n output lines.
- In encoder the output lines generate the binary code corresponding to the input value.



Decimal to BCD Encoder

- The decimal to BCD encoder, usually has ten input lines and four output lines.
- The decoded decimal data acts as an input for encoder and encoded BCD output is available on the four output lines.
- In IC 74XX147, it has nine input lines and four output lines,
- Both the input and output lines are asserted active low.
- It is important to note that there is no input line for decimal zero, when this condition occurs, all output lines are 1.



Truth Table for Decimal to BCD encoder

Decimal Value	Inputs									Outputs			
	1	2	3	4	5	6	7	8	9	D	C	B	A
0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	0
2	X	0	1	1	1	1	1	1	1	1	1	0	1
3	X	X	0	1	1	1	1	1	1	1	1	0	0
4	X	X	X	0	1	1	1	1	1	1	0	1	1
5	X	X	X	X	0	1	1	1	1	1	0	1	0
6	X	X	X	X	X	0	1	1	1	1	0	0	1
7	X	X	X	X	X	X	0	1	1	1	0	0	0
8	X	X	X	X	X	X	X	0	1	0	1	1	1
9	X	X	X	X	X	X	X	X	0	0	1	1	0

X denotes don’t care condition

Priority Encoder

- A priority encoder is an encoder circuit that includes the priority function.
- In priority encoder, if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.
- D_3 input with highest priority and D_0 input with lowest priority. When D_3 input is high, regardless of other inputs output is $11(Y_1\ Y_0 = 11)$
- The D_2 has the next priority. Thus, when D_3=0 and $D_2 = 1$, regardless of other two lower priority input, output is 10.
- The output for D_1 is generated only if higher priority inputs are 0 and so on.
- The output V (a valid output indicator) indicates, one or more of the inputs are equal to 1. If all inputs are 0, V is equal to 0, and the other two outputs (Y_1 and Y_0) of the circuit are not used.

Inputs				**Outputs**		
D_0	D_1	D_2	D_3	Y_1	Y_0	**V**
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Truth table of 4-bit priority encoder

By K-map Simplification:

For Y_1

D_0D_1 \ D_2D_3	00	01	11	10
00	X 0	1 1	1 3	1 2
01	0 4	1 5	1 7	1 6
11	0 12	1 13	1 15	1 14
10	0 8	1 9	1 11	1 10

For Y_0

D_0D_1 \ D_2D_3	00	01	11	10
00	X 0	1 1	1 3	0 2
01	1 4	1 5	1 7	0 6
11	1 12	1 13	1 15	0 14
10	0 8	1 9	1 11	0 10

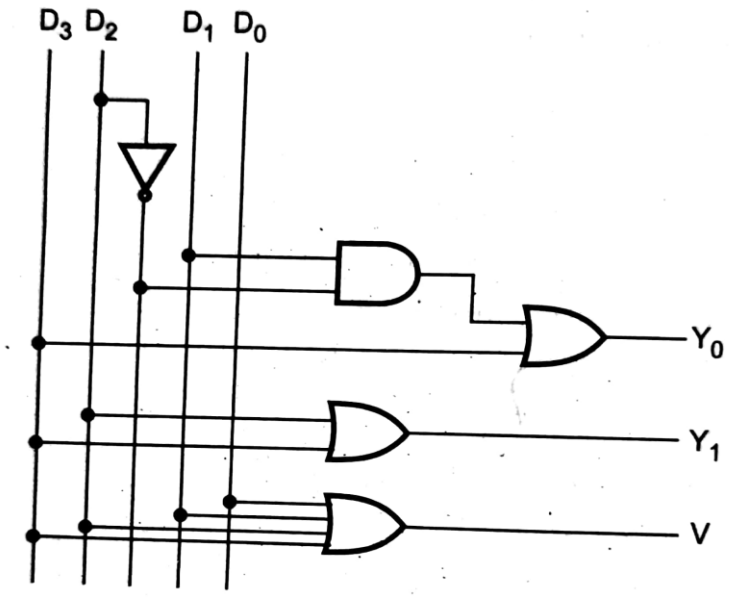
For V

D_0D_1 \ D_2D_3	00	01	11	10
00	0 0	1 1	1 3	1 2
01	1 4	1 5	1 7	1 6
11	1 12	1 13	1 15	1 14
10	1 8	1 9	1 11	1 10

$$Y_1 = D_2 + D_3$$

$$Y_0 = D_3 + D_1\overline{D_2}$$

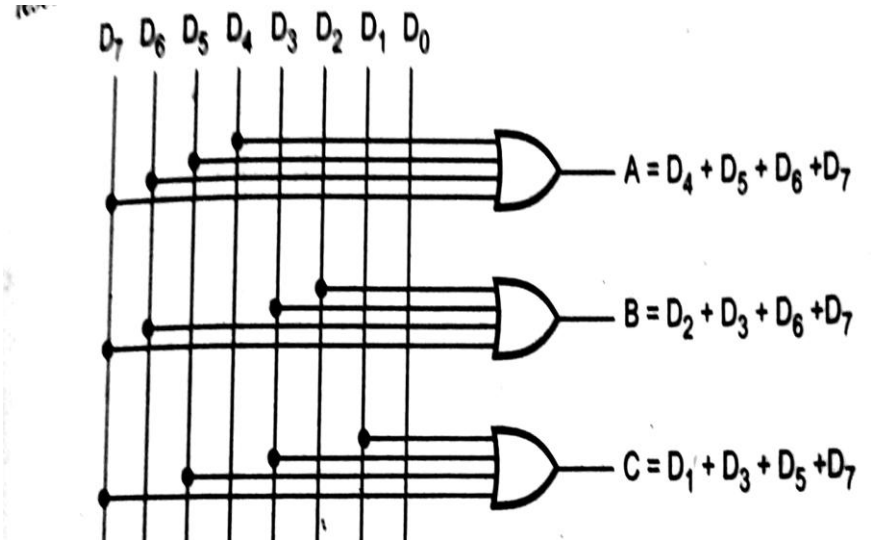
$$V = D_0 + D_1 + D_2 + D_3$$



Octal to Binary Encoder

- It has eight inputs, one for each octal digit, and three outputs that generate the corresponding binary code.
- In encoders it is assumed that only one input has a value of 1 at any given time; otherwise the circuit is meaningless.
- The circuit has one more ambiguity that when all inputs are 0s the outputs are 0s. The zero output can also be generated when D_0=1. this ambiguity can be resolved by providing an additional output that specifies the valid condition.

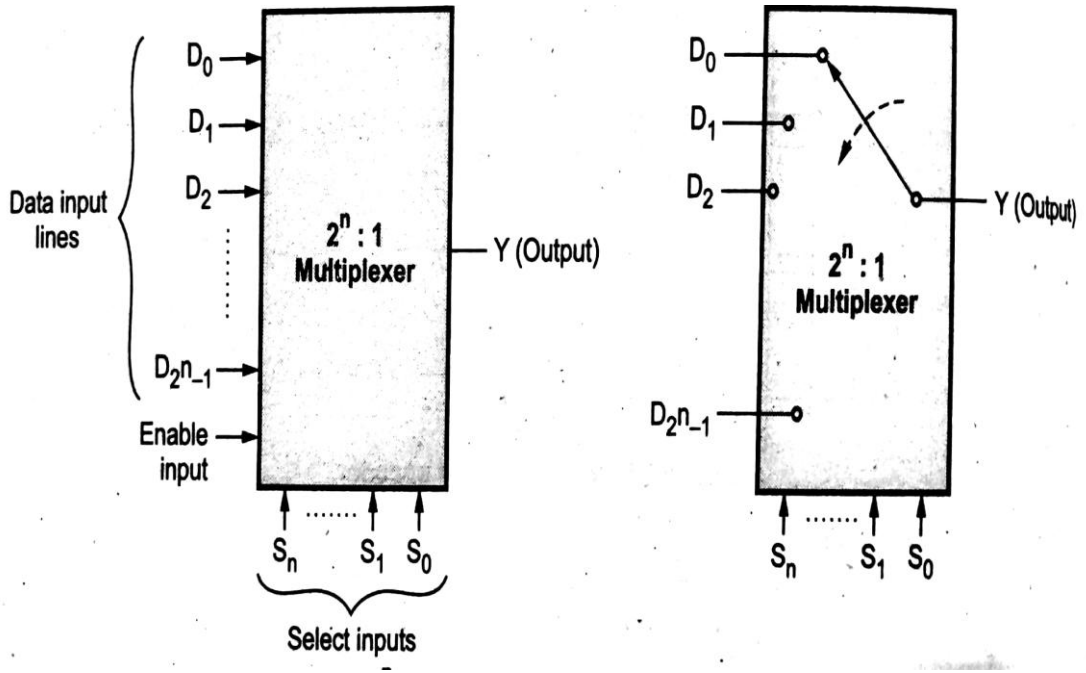
Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1



Encoder ICs:
74147-Decimal to BCD encoder
74148-8-input priority encoder

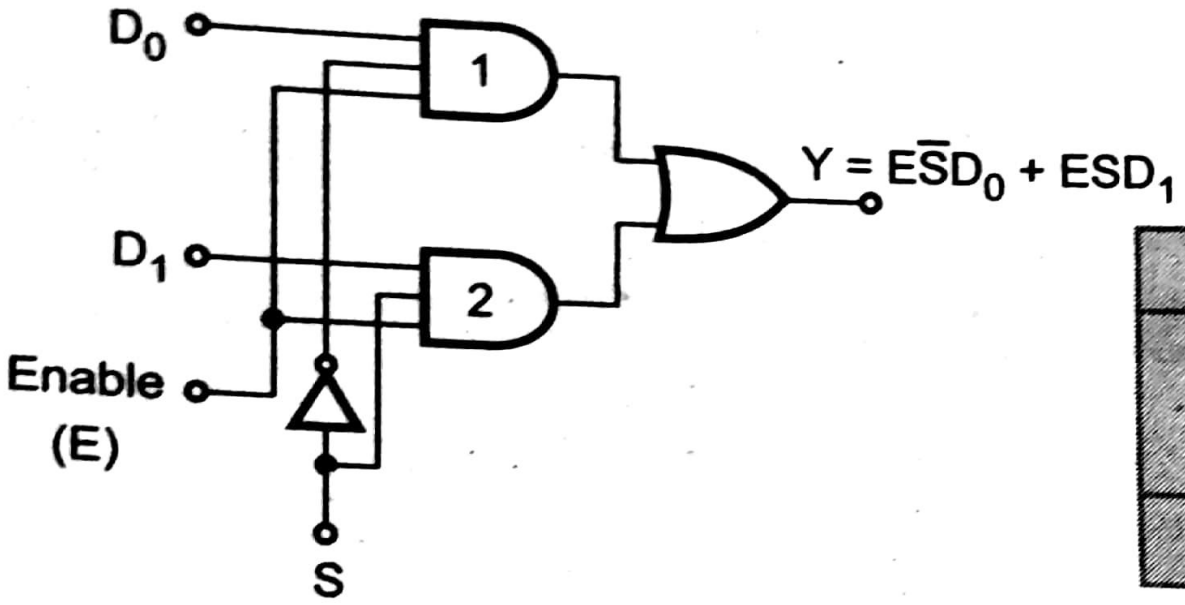
Multiplexers

- In digital systems, many times it is necessary to select single data line from several data-input lines, and the data from the selected data line should be available on the output.
- This digital circuit which does this task is called a multiplexer.
- It is a digital switch.
- It allows digital information from several sources to be routed onto a single output line.
- The basic multiplexer has several data-input lines and a single output line.
- The selection of a particular input line is controlled by a set of selection lines.
- Since multiplexer selects one of the input and routes it to output, it is also known as data selector.
- Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.
- Therefore, multiplexer is 'many into one' and it provides the digital equivalent of an analog selector switch.



2:1 Multiplexer:

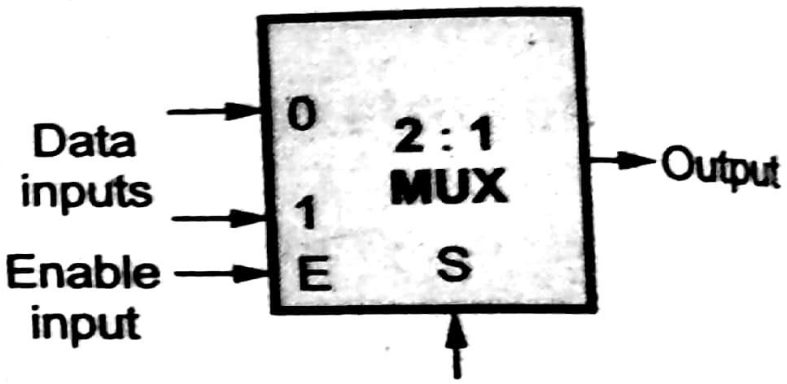
- D_0 is applied as an input to one AND gate and D_1 is applied as an input to another AND gate.
- Enable input is applied to both gates as one input.
- Selection line S is connected as second input to second AND gate.
- An inverted S is applied as second input to first AND gate.
- Outputs of both AND gates are applied as inputs to OR gate.



(a) Logic diagram

E	S	Y
1	0	D_0
1	1	D_1
0	X	0

(b) Function table



(c) Logic symbol

Working:

- When E=0,output is 0,i.e.,Y=0 irrespective of any input condition.
- When E=1, the circuit works as follows:

a. When S=0, the inverted S, that is gate 1 gets applied as second input to first AND gate. Since S is applied directly as input to second AND gate, its output goes zero irrespective of first input. Since the second input of first AND gate is 1, its output is equal to its first input, that is D_0. Hence $Y= D_0$

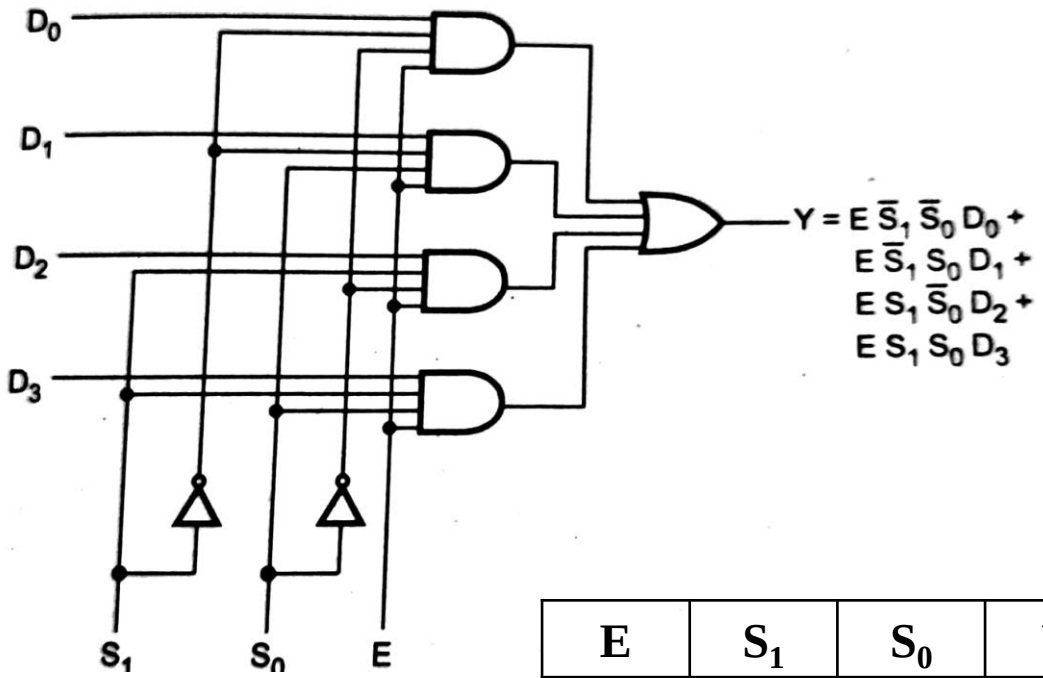
b. Exactly opposite is the case when S=1. In this case, second AND gate output is equal to its first input D_1 and first AND gate output is 0.Hence $Y= D_1$.

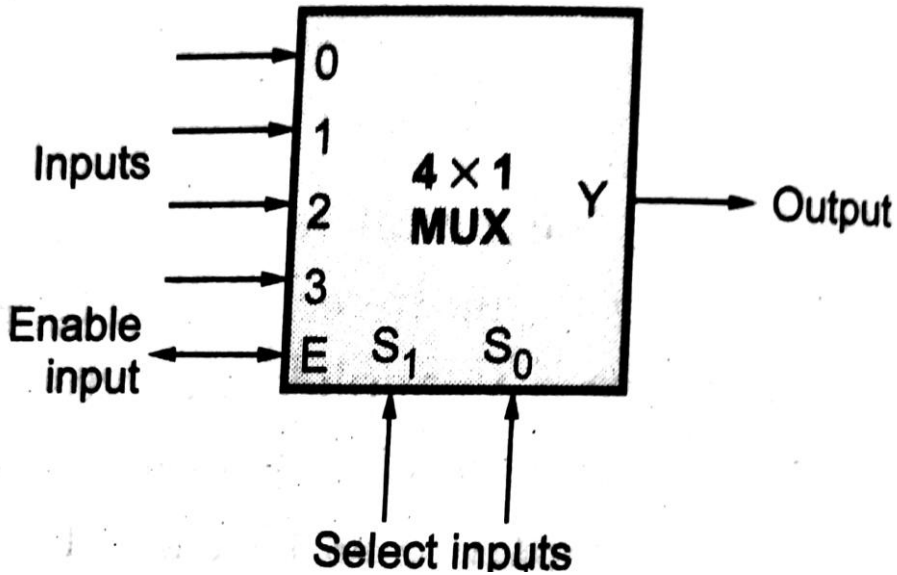
Enable (E)	Select (S)	D_1	D_0	Output Y	
1	0	X	0	0	
1	0	X	1	1	$E\bar{S}D_0$
1	1	0	X	0	
1	1	1	X	1	ESD_1
0	X	X	X	0	

Truth Table for 2:1 multiplexer

$Y= E\bar{S}D_0 + ESD_1$

4:1 Multiplexer:

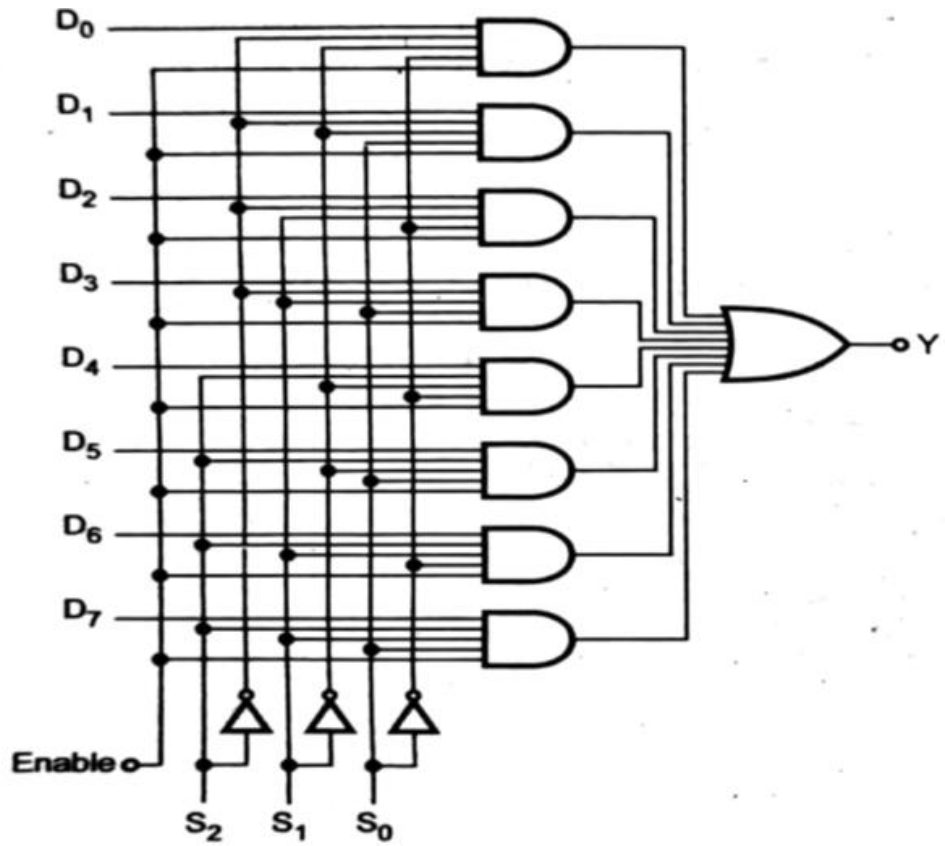




E	S_1	S_0	Y
1	0	0	D_0
1	0	1	D_1
1	1	0	D_2
1	1	1	D_3
0	X	X	0

Function Table

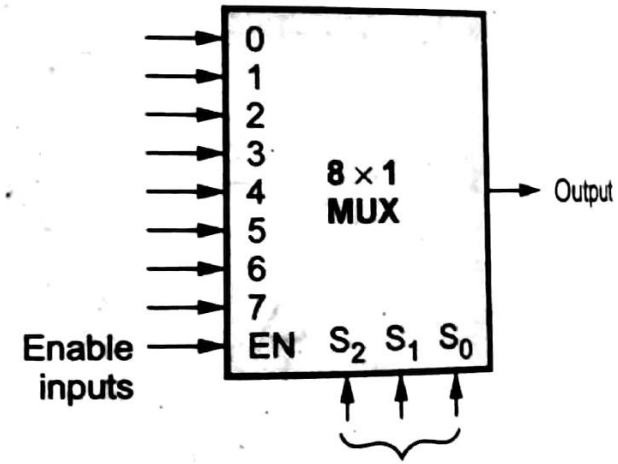
8:1 Multiplexer:



(a) Logic diagram

S_2	S_1	S_0	Y
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

(b) Function Table



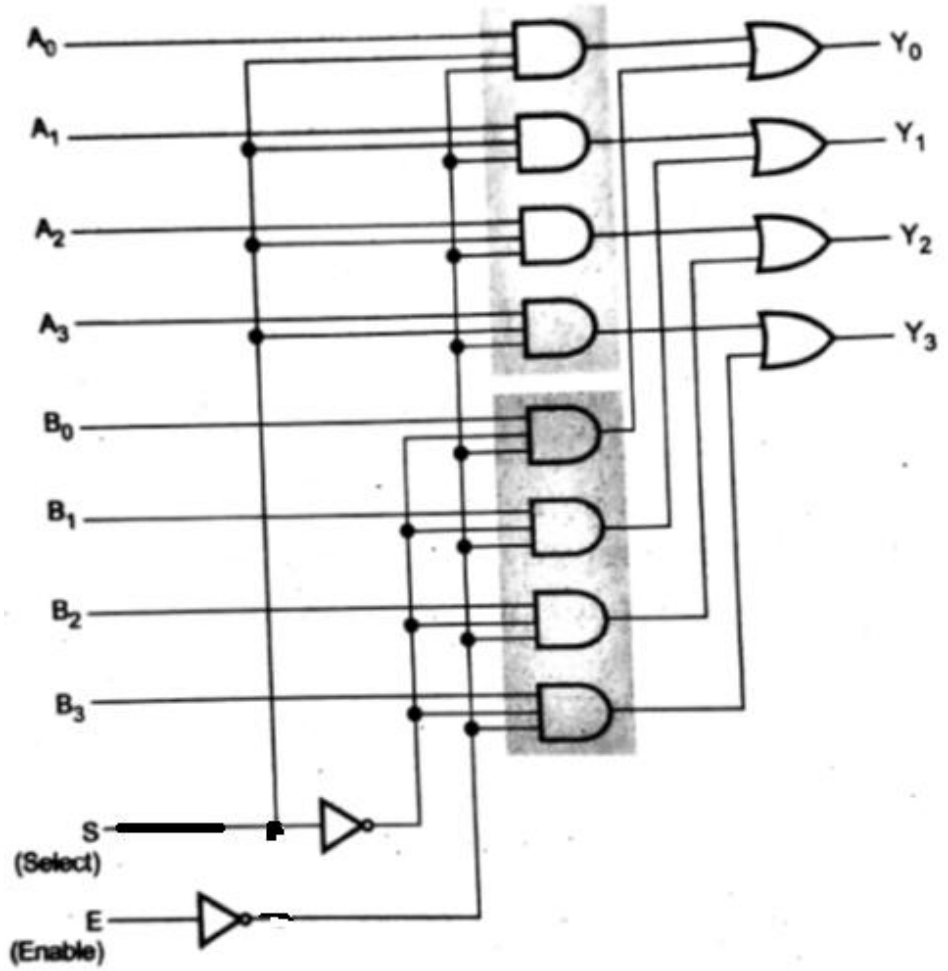
(c) Logic symbol

Quadruple 2 to 1 Multiplexer

- In some cases, two or more multiplexers are enclosed within one IC package.
- The figure shows quadruple 2-to-1 line multiplexer, i.e., four multiplexers, each capable of selecting one of two input lines.

E	S	Output Y
1	X	All 0s
0	1	Select A
0	0	Select B

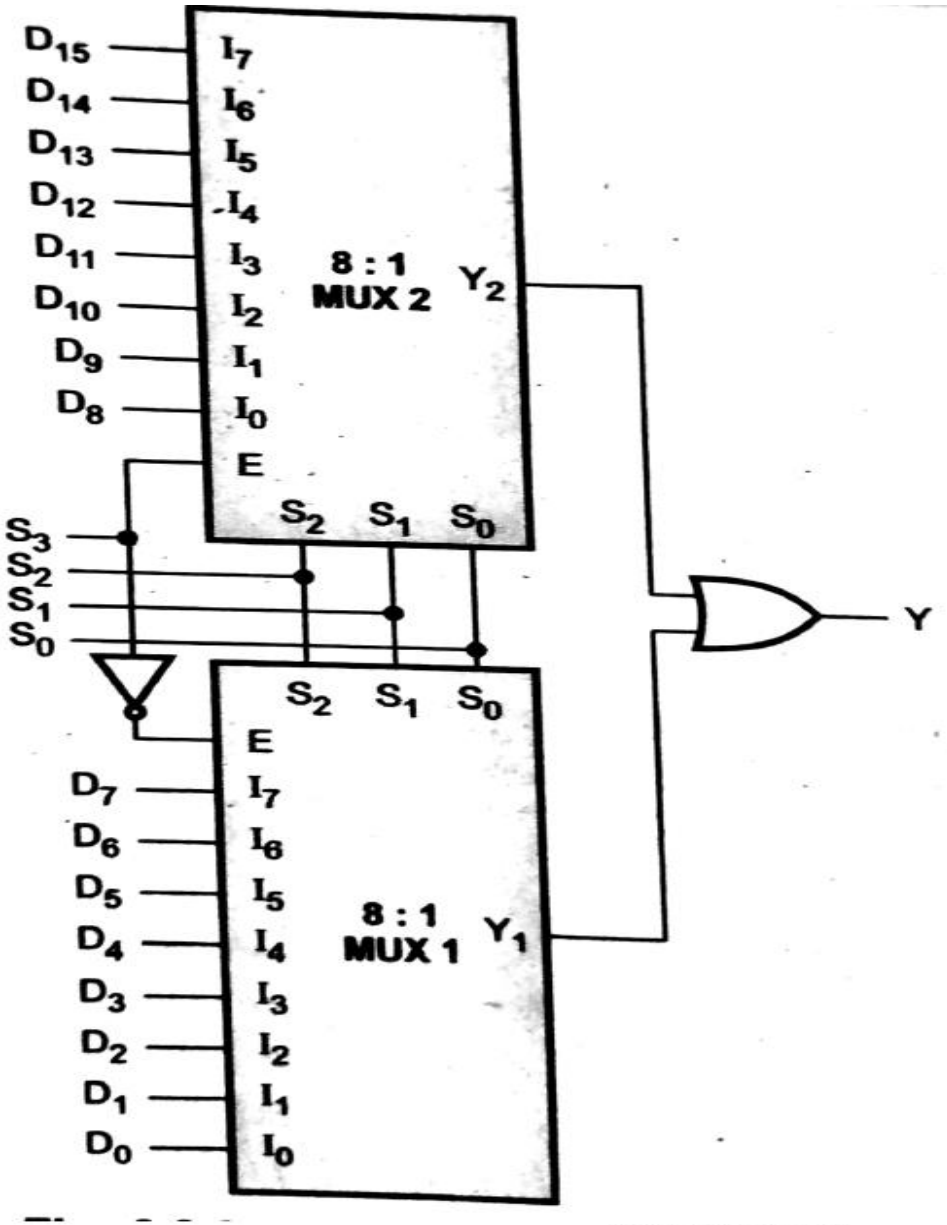
Function Table



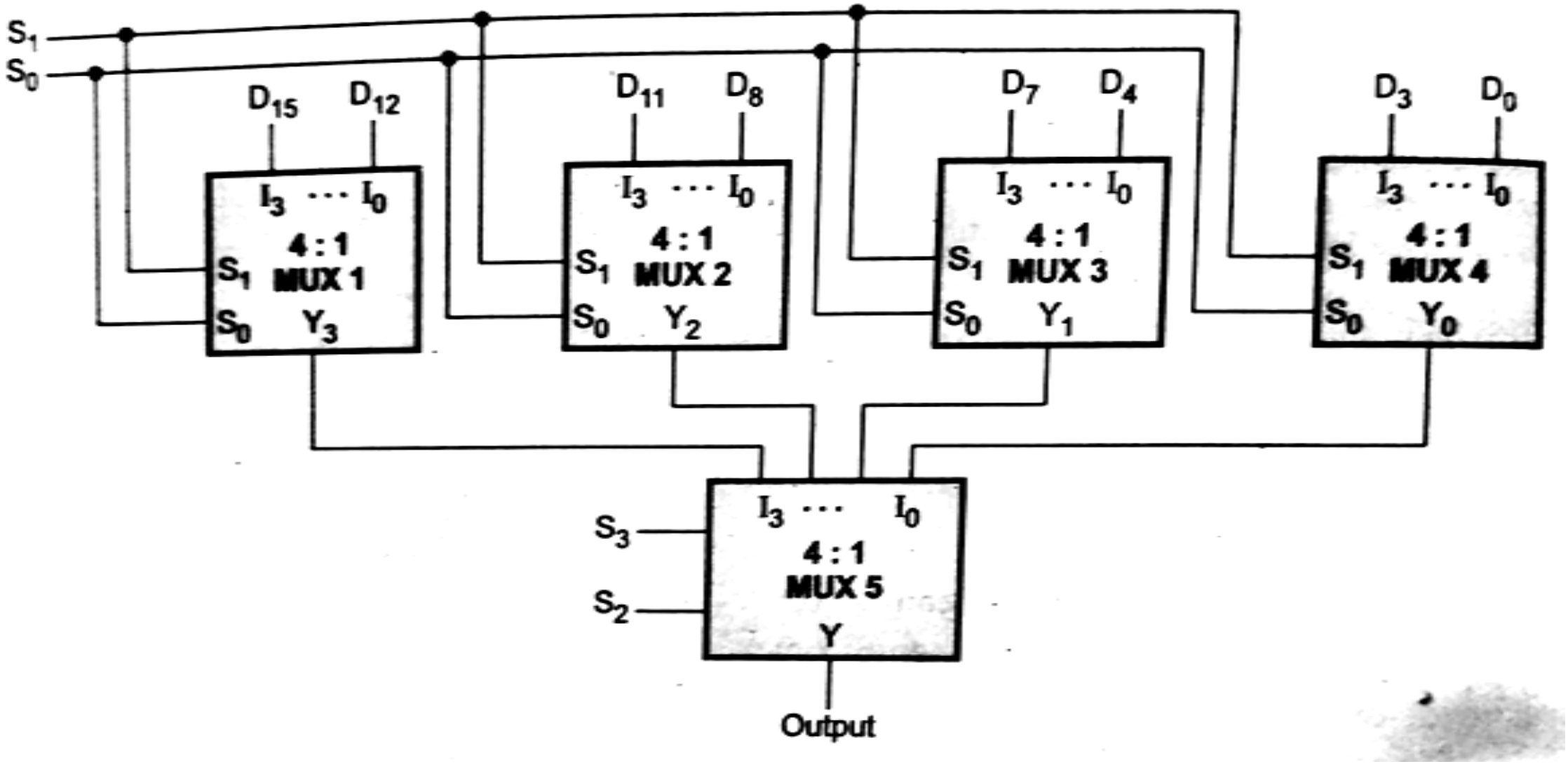
Expanding Multiplexers

- It is possible to expand range of inputs for multiplexer beyond the available range by interconnecting several multiplexers in cascade.
- The circuit with two or more multiplexers connected to obtain the multiplexer with more number of inputs is known as multiplexer tree.

Design 16:1 multiplexer using 8:1 multiplexer.



Design 16:1 multiplexer using 4:1 multiplexers.



Implementation of combinational logic using MUX

1. Implement the given function using 8:1 multiplexer. F(A,B,C)=Σm(1,3,5,6)

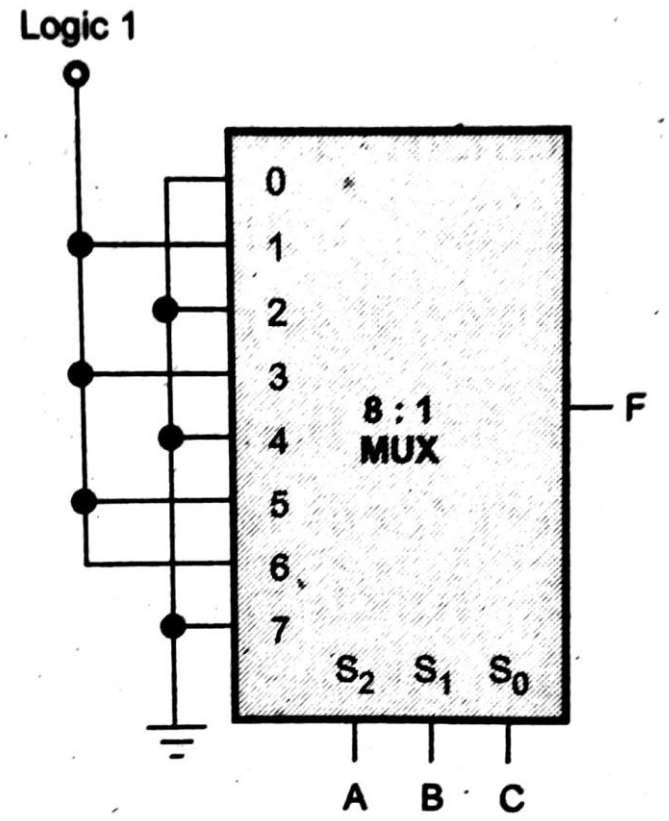
Solution:

Step 1: Select the multiplexer. Here, Boolean expression has 3 variables, thus we require $2^3 = 8:1$ multiplexer.

Step 2: Connect inputs corresponds to the present minterms to logic 1.

Step 3: Connect remaining inputs to logic 0.

Step 4: Connect input variables to select lines of MUX.



2. Implement the following Boolean function using 8:1 multiplexer

$$F(A,B,C,D)=\bar{A}B\bar{D}+ACD+\bar{B}CD+\bar{A}\bar{C}D$$

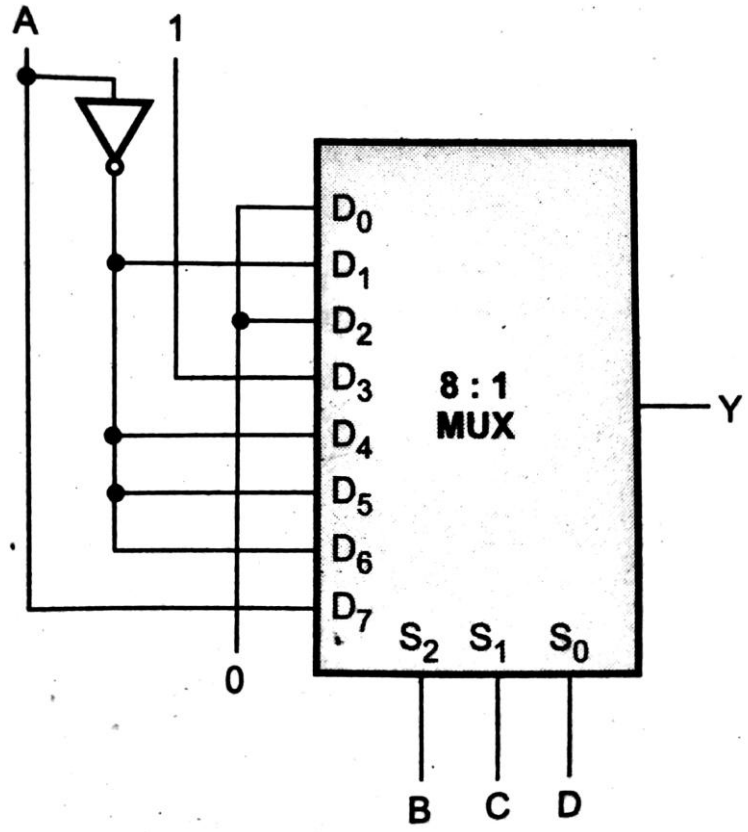
Solution:

Step 1: Express the Boolean function in the minterm form.

$$F(A,B,C,D)=\Sigma m(1,3,4,5,6,11,15)$$

Step 2: Implement it using implementation table.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	0	1	2	3	4	5	6	7
A	8	9	10	11	12	13	14	15
	0	$\bar{A}$	**0**	**1**	$\bar{A}$	$\bar{A}$	$\bar{A}$	**A**



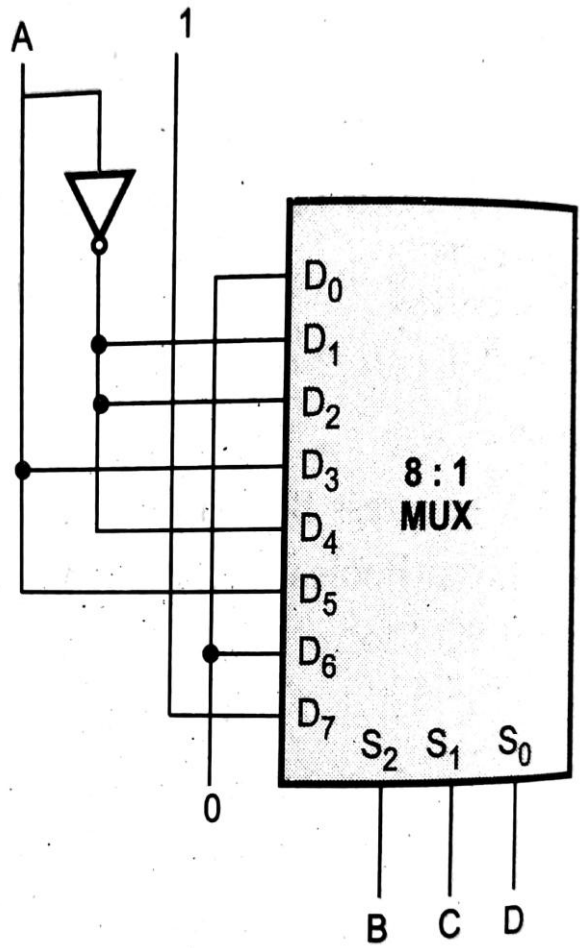
Multiplexer Implementation

3. Implement the following Boolean function with 8:1 multiplexer.

F(A,B,C,D)=πM(0,3,5,6,8,9,10,12,14)

Solution: Here, instead of minterms, maxterms are specified. Thus, we have to circle the maxterms which are not included in the Boolean function.

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{A}$	0	1	2	3	4	5	6	7
A	8	9	10	11	12	13	14	15
	0	$\overline{A}$	$\overline{A}$	**A**	$\overline{A}$	A	**0**	**1**



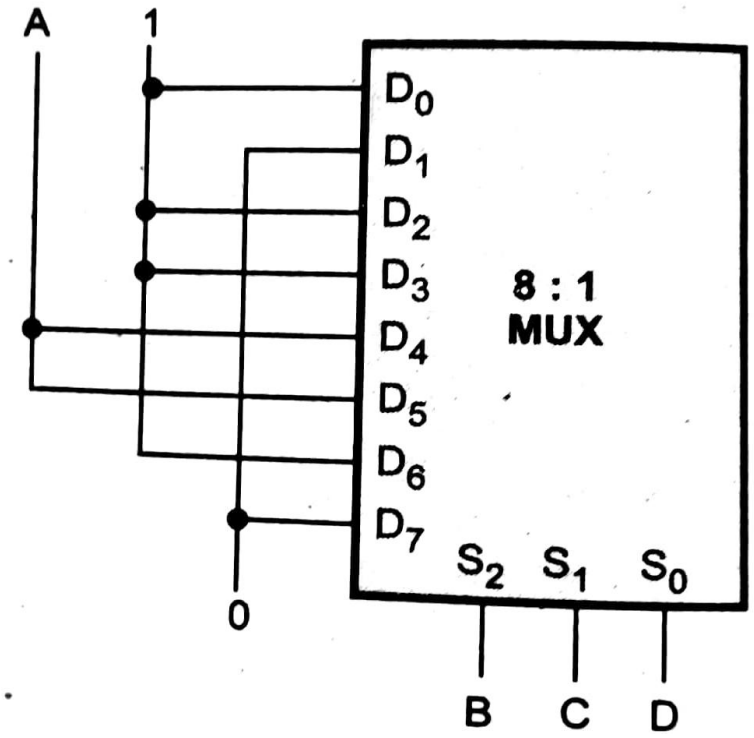
4. Implement the following Boolean function with 8:1 multiplexer.

$$F(A,B,C,D)=\Sigma m(0,2,6,10,11,12,13)+d(3,8,14)$$

Solution:

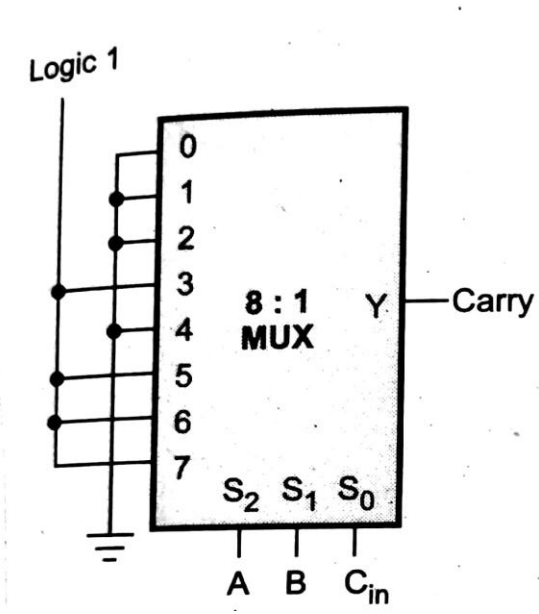
In the given Boolean function three don't care conditions are also specified. We know that don't care conditions can be treated as either 0s or 1s. Here, don't cares are treated as 1s.

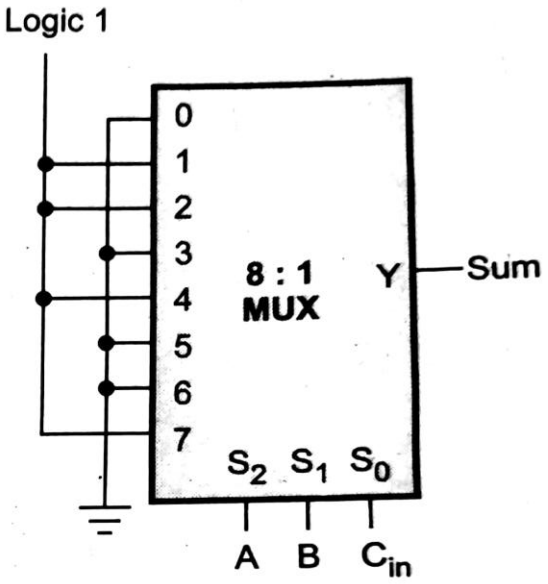
	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{A}$	0	1	2	3	4	5	6	7
A	8	9	10	11	12	13	14	15
	1	**0**	**1**	**1**	***A***	***A***	**1**	**0**



5. Implement full adder circuit using 8:1 multiplexer.

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





6. Implement full adder circuit using quadruple 2 to 1 multiplexer.

Solution:

Implementation Table

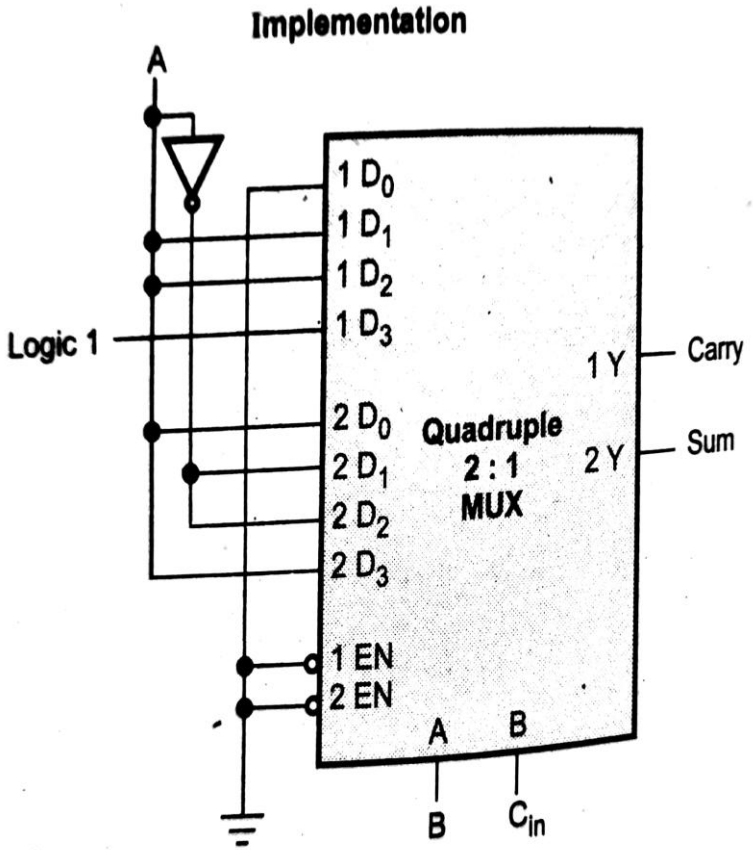
Inputs			Outputs	
A	**B**	**C_{in}**	**Carry (C_{out})**	**Sum**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sum:

	D_0	D_1	D_2	D_3
$\overline{A}$	0	1	2	3
A	4	5	6	7
	A	$\overline{A}$	$\overline{A}$	**A**

Carry:

	D_0	D_1	D_2	D_3
$\overline{A}$	0	1	2	3
A	4	5	6	7
	0	***A***	***A***	**1**

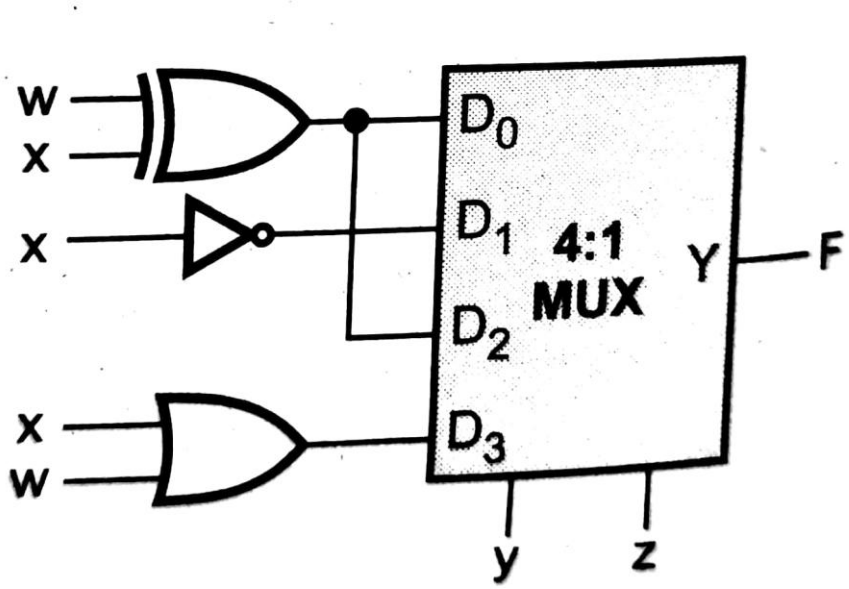


7. Realize F(w,x,y,z)=Σ(1,4,6,7,8,9,10,11,15) using 4 to1 MUX.

Solution:

	D_0	D_1	D_2	D_3
$\bar{w}\bar{x}$	0	1	2	3
$\bar{w}x$	4	5	6	7
$w\bar{x}$	8	9	10	11
wx	12	13	14	15

$$D_0=\bar{w}x+w\bar{x}$$
$$=w\oplus x$$
$$D_1=\bar{w}\bar{x}+w\bar{x}$$
$$=\bar{x}$$
$$D_2=\bar{w}x+w\bar{x}$$
$$=w\oplus x$$
$$D_3=\bar{w}x+w\bar{x}+wx$$
$$=w+x$$



Applications of Multiplexer:

- They are used as a data selector to select one out of many data inputs.
- They can be used to implement combinational logic circuit.
- They are used in time multiplexing systems.
- They are used in frequency multiplexing systems.
- They are used in A/D and D/A converter.
- They are used in data acquisition systems.

Multiplexer ICs:

74150→16:1 multiplexer

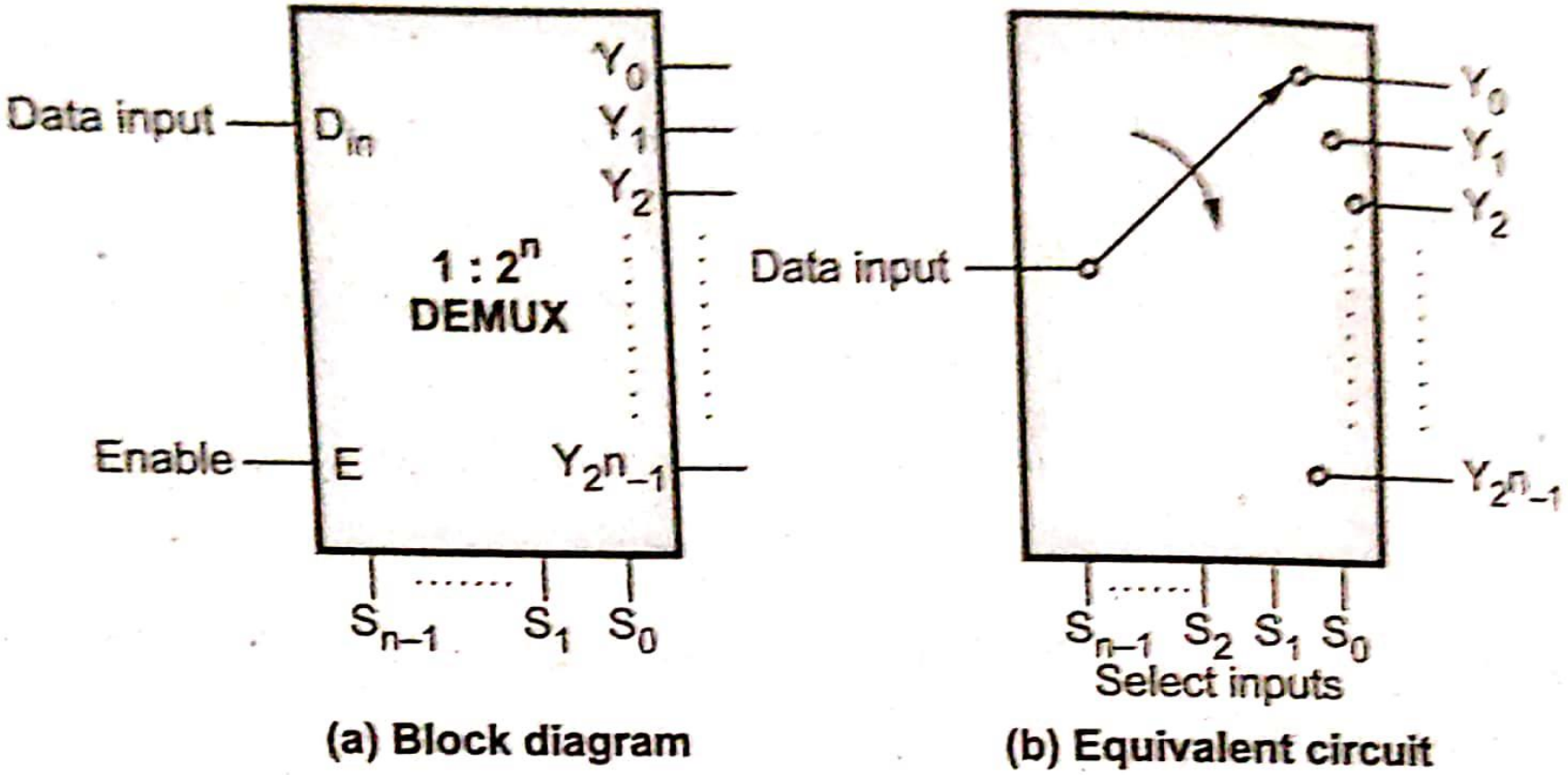
74151→8:1 multiplexer

74153→Dual 4:1 multiplexer

74157→Quad 2-input multiplexer

Demultiplexers

- A demultiplexer is a circuit that receives information on a single line and transmits this information on one of 2^n possible output lines.
- The selection of specific output line is controlled by the values of n selection lines.
- It has one input data line, 2^n output lines, n select lines and one enable input.

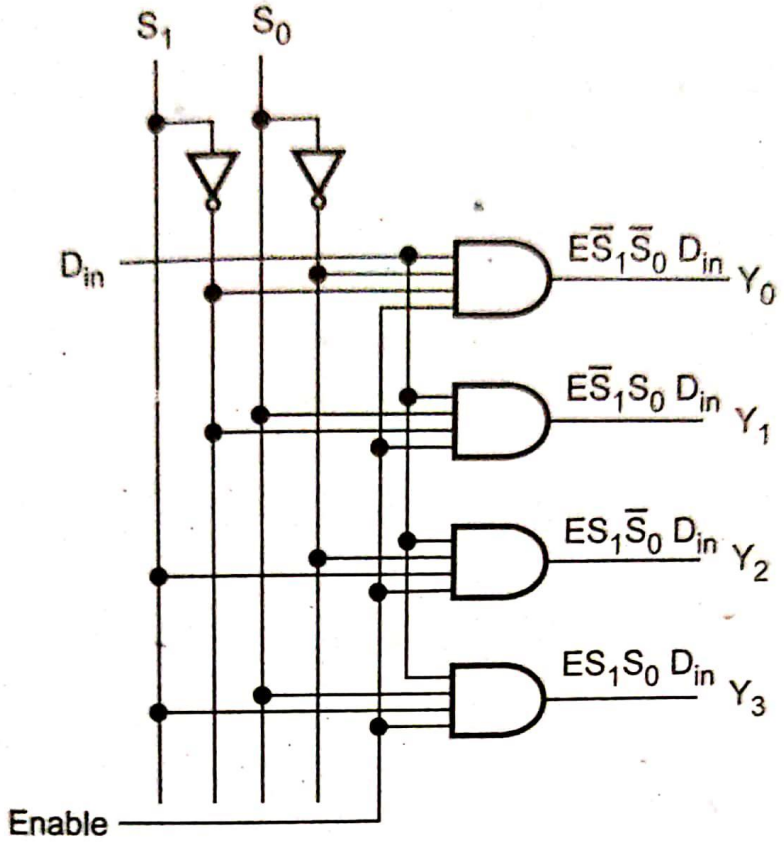


Types of Demultiplexers

1:4 Demultiplexer:

- The single input variable $\mathbf{D_{in}}$ has a path to all four outputs, but the input information is directed to only one of the output lines depending on the select inputs.
- Enable input should be high to enable demultiplexer.

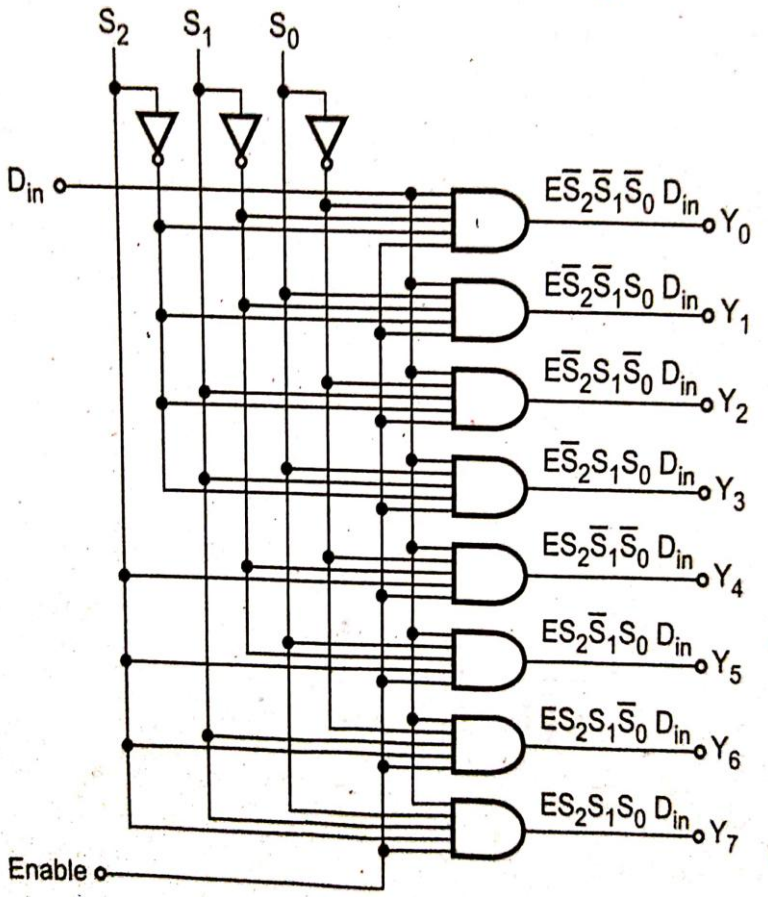
Enable (E)	S_1	S_0	D_{in}	Y_0	Y_1	Y_2	Y_3	
0	X	X	X	0	0	0	0	
1	0	0	0	0	0	0	0	D_{in} is routed to Y_0
1	0	0	1	1	0	0	0	$Y_0 = E\,\overline{S}_1\,\overline{S}_0\,D_{in}$
1	0	1	0	0	0	0	0	D_{in} is routed to Y_1
1	0	1	1	0	1	0	0	$Y_1 = E\,\overline{S}_1\,S_0\,D_{in}$
1	1	0	0	0	0	0	0	D_{in} is routed to Y_2
1	1	0	1	0	0	1	0	$Y_2 = E\,S_1\,\overline{S}_0\,D_{in}$
1	1	1	0	0	0	0	0	D_{in} is routed to Y_3
1	1	1	1	0	0	0	1	$Y_3 = E\,S_1\,S_0\,D_{in}$



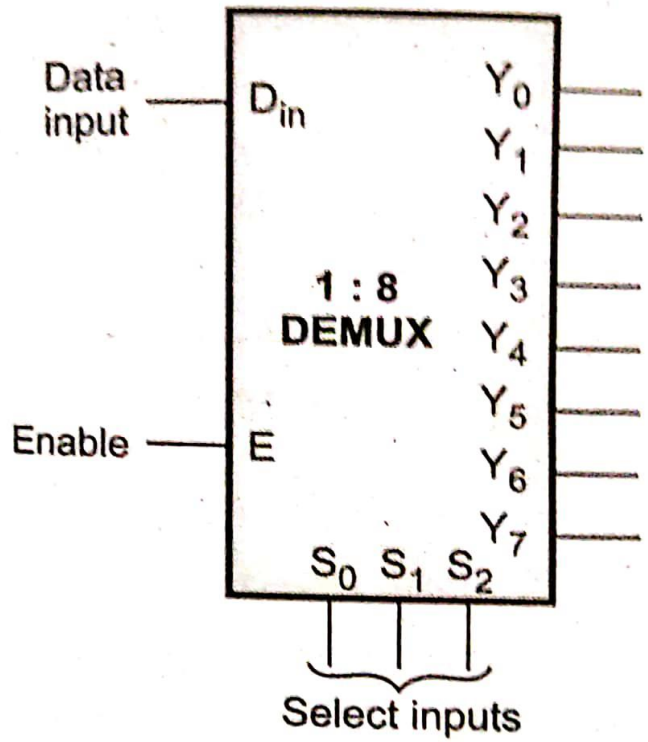
1:8 Demultiplexer:

The single input data $\mathbf{D_{in}}$ has a path to all eight outputs, but the input information is directed to only one of the output lines depending on the select inputs.

Enable	Select inputs			Outputs								
E	S_2	S_1	S_0	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	
0	X	X	X	0	0	0	0	0	0	0	0	
1	0	0	0	D_{in}	0	0	0	0	0	0	0	$E\,\bar{S}_2\,\bar{S}_1\,\bar{S}_0\,D_{in}$
1	0	0	1	0	D_{in}	0	0	0	0	0	0	$E\,\bar{S}_2\,\bar{S}_1\,S_0\,D_{in}$
1	0	1	0	0	0	D_{in}	0	0	0	0	0	$E\,\bar{S}_2\,S_1\,\bar{S}_0\,D_{in}$
1	0	1	1	0	0	0	D_{in}	0	0	0	0	$E\,\bar{S}_2\,S_1\,S_0\,D_{in}$
1	1	0	0	0	0	0	0	D_{in}	0	0	0	$E\,S_2\,\bar{S}_1\,\bar{S}_0\,D_{in}$
1	1	0	1	0	0	0	0	0	D_{in}	0	0	$E\,S_2\,\bar{S}_1\,S_0\,D_{in}$
1	1	1	0	0	0	0	0	0	0	D_{in}	0	$E\,S_2\,S_1\,\bar{S}_0\,D_{in}$
1	1	1	1	0	0	0	0	0	0	0	D_{in}	$E\,S_2\,S_1\,S_0\,D_{in}$



Logic Symbol:



Expanding Demultiplexer:

To provide larger output needs we can cascade two or more demultiplexer to get demultiplexer with more number of output lines. Such a connection is known as demultiplexer tree.

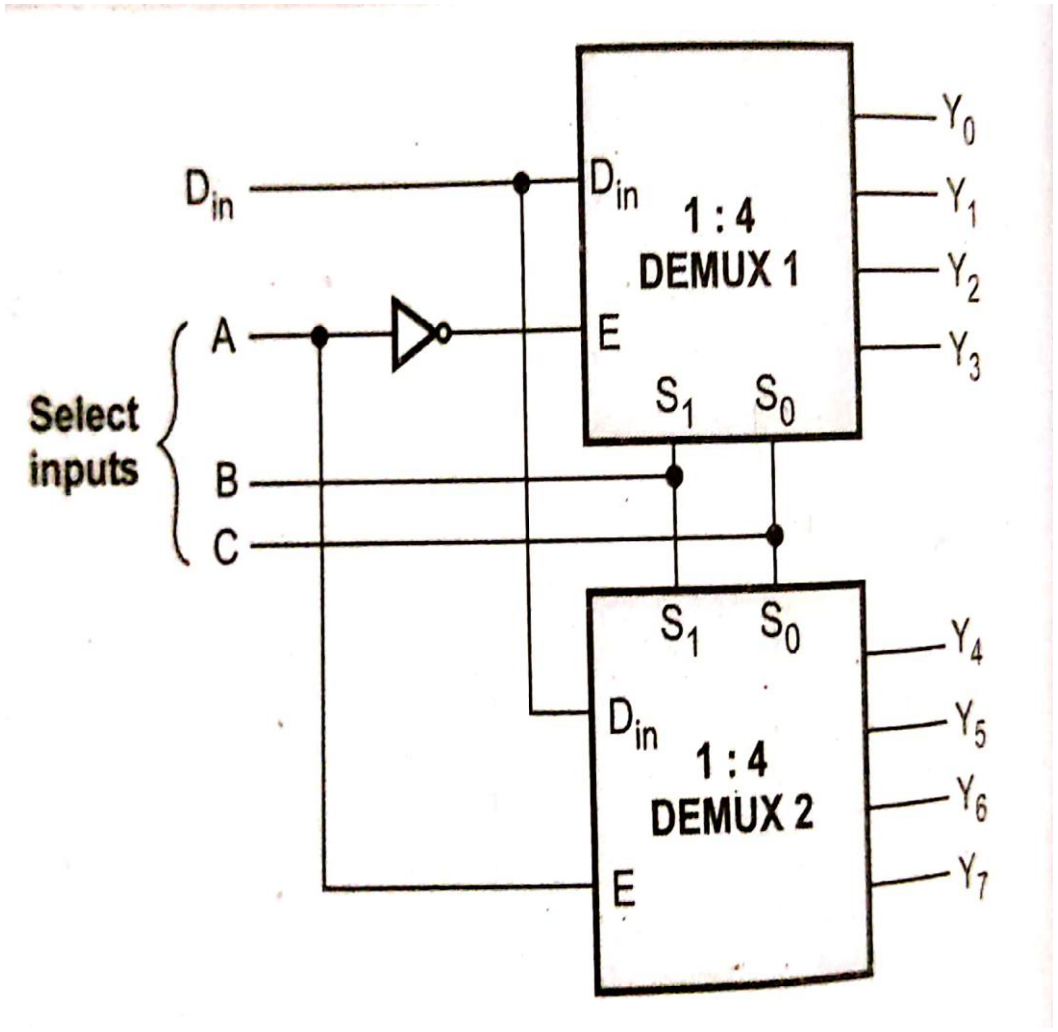
1. Design 1:8 demultiplexer using two 1:4 demultiplexers.

Solution:

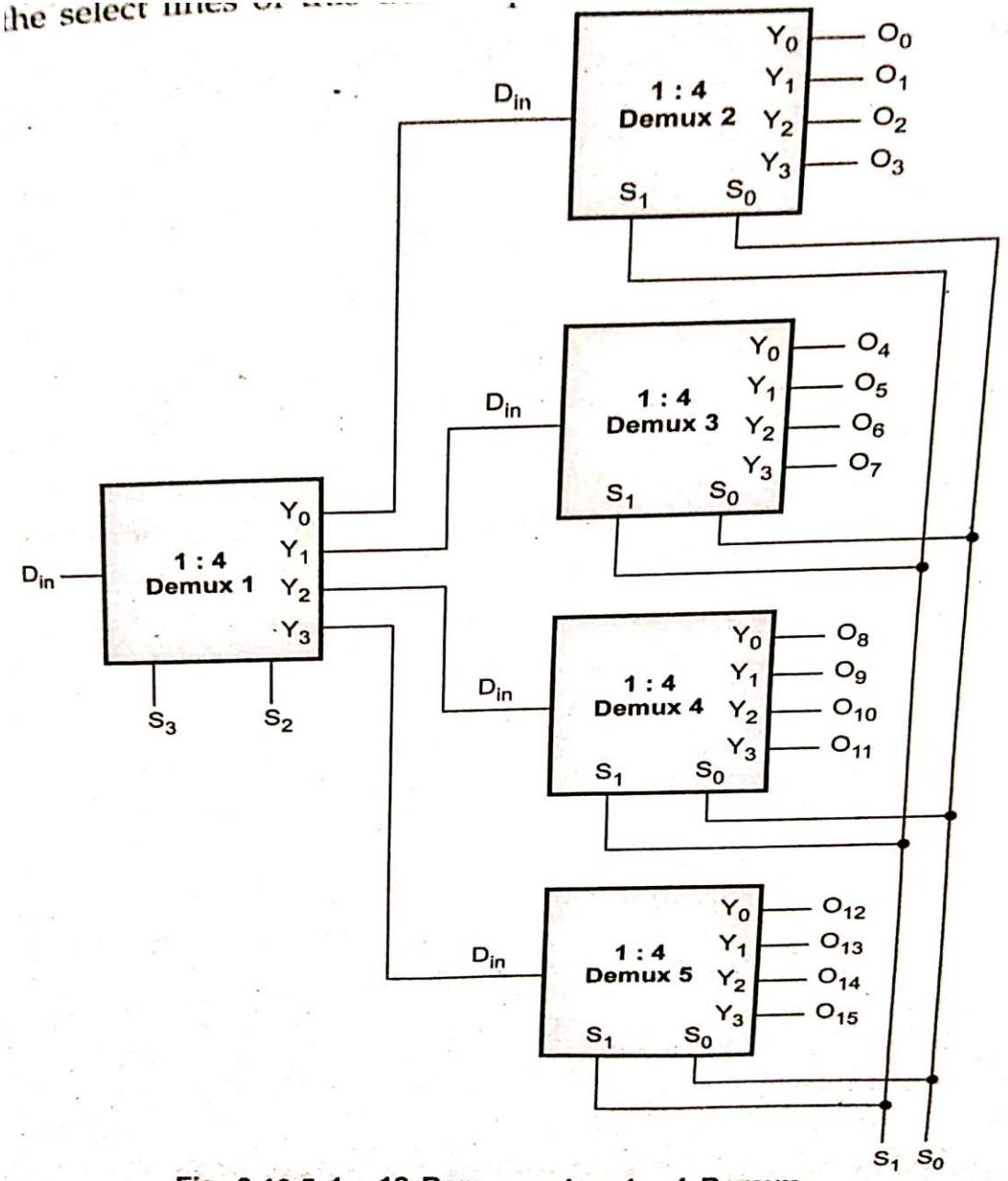
Step 1: Connect D_{in} signal to D_{in} input of both the demultiplexers.

Step 2: Connect select lines B and C to select lines S_1 and S_0 of the both demultiplexer.

Step 3: Connect most significant select line (A) such that when A=0 DEMUX 1 is enabled and when A=1 DEMUX 2 is enabled.



Implement 1:16 demultiplexer using 1:4 demultiplexer.

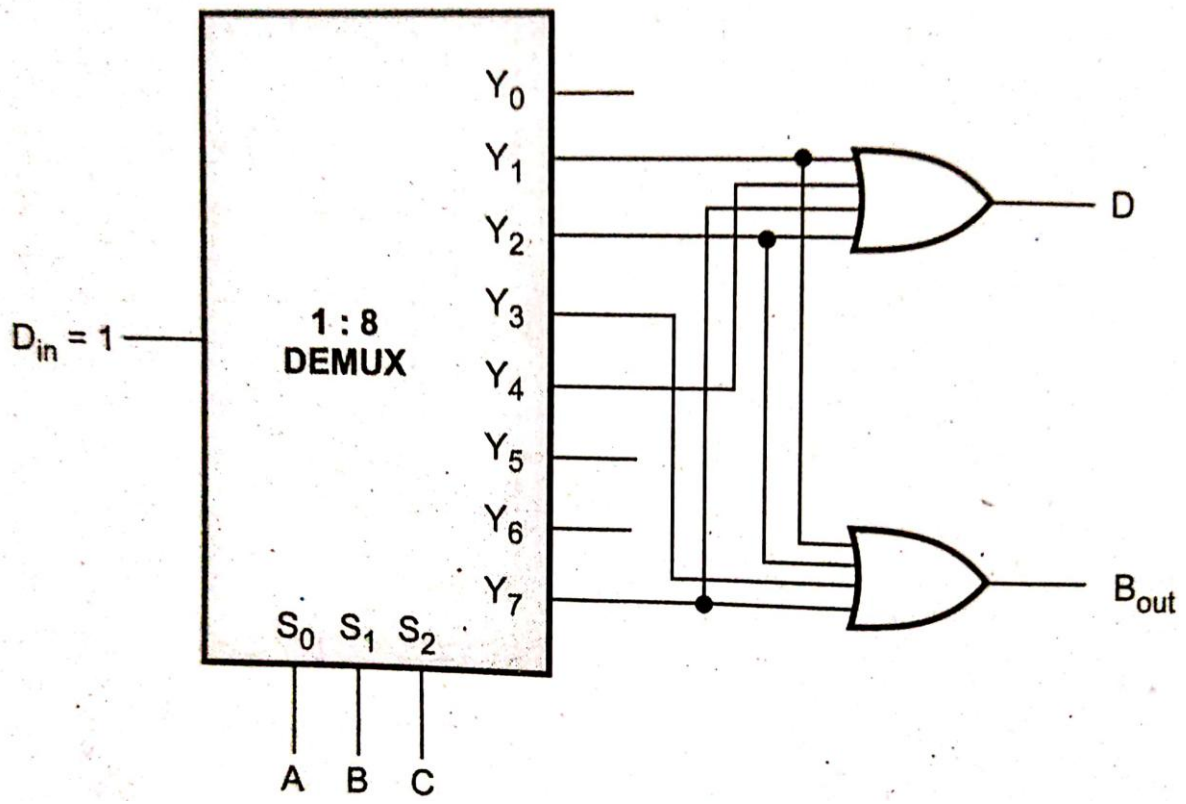


Implementation of Combinational Logic using Demultiplexer

Implement full subtractor using demultiplexer.

Solution:

Inputs			Outputs	
A	**B**	**B_{in}**	**D**	**B_{out}**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



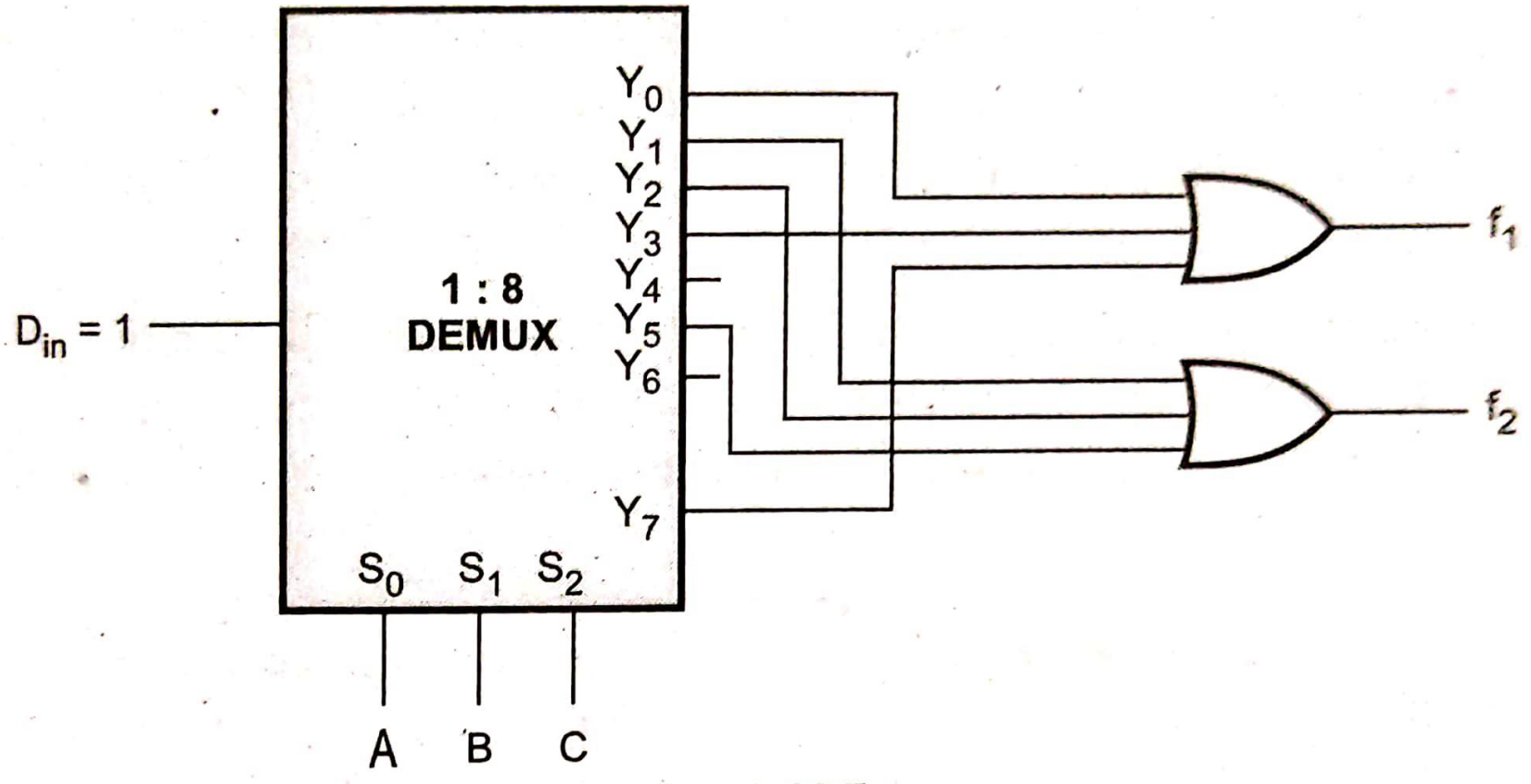
B_{out}=Σm(1,2,3,7)

D=Σm(1,2,4,7)

Implement the following functions using demultiplexer.

$F_1(A,B,C)=\Sigma m(0,3,7)$

$F_2(A,B,C)=\Sigma m(1,2,5)$



Applications of Demultiplexer:

- It can be used as a decoder.
- It can be used as a data distributor.
- It is used in time division multiplexing at the receiving end as a data separator.
- It can be used to implement Boolean expression.

Demultiplexer ICs:

74154→1:16 Demultiplexer

74155→Dual 1:4 Demultiplexer

Introduction to Hardware Description Language

Various modeling techniques in HDL

- Gate-level modeling/Structural modeling
- Dataflow modeling
- Behavioral modeling

Structure of HDL module

- Each module consists of a declaration and a body.
- Declaration → name, inputs and outputs of the module are listed.
- Body→ relationship between the inputs and outputs
- A module is a basic building block of Verilog HDL.
- Modules can represent pieces of hardware ranging from simple gate to complete systems.
- The structure of module is,

  module<module name> <port list>;
  <declares>
  <module items>
  endmodule

Operators in Verilog HDL

- Boolean logic→(!,&&,||)
- Unary reduction logical→(&, |,^)
- Bitwise logical→(~&, ~|,~^,~)
- Relational→(>,>=,<,<=,==,!=)
- Binary arithmetic→(+,-,*,/,%)
- Other →(<<,>>)

Structure of the Data Flow Description

Verilog HDL code for full adder	Verilog HDL code for multiplexer with active low enable

module full_add (A,B,Cin,Cout,Sum);
input A,B,Cin;
output Sum,Cout;
assign Sum=(A^B) ^Cin;
assign Cout=(A & B)|(Cin & A)|(Cin & B);
endmodule
	module mux 2x1(D0,D1,S,Enbar,Y);
	input D0,D1,S,Enbar;
	output Y;
	Wire I1,I2,I3,I4;
	assign #10 Y=I3|I4;
	assign #10 I3=D0 & I1 & I2;
	assign #10 I4=D1 & S & I2;
	assign #10 I1=~S;
	assign #10 I2=~Enbar;
	
	endmodule

Structure of the Behavioral Description

Verilog HDL code for full adder	Verilog HDL code for multiplexer

module full_add (A,B,Cin,Cout,Sum);
input A,B,Cin;
output Sum,Cout;
Reg Sum,Cout;
always@(A,B,Cin)
begin
Sum=(A^B) ^Cin;
Cout=(A & B)|(Cin & A)|(Cin & B);
end
endmodule
	module mux 2x1(D0,D1,S,Enbar,Y);
	input D0,D1,S,Enbar;
	output Y;
	Reg Y;
	always@(S,D0,D1,Enbar)
	begin
	if(Enbar==0 & S==1)
	begin
	Y=D0;
	end
	else if(Enbar==0 & S==0)
	Y=D1;
	else
	Y=1'bz;
	end
	endmodule

Structural/ Gate Level Description

Verilog HDL code for full adder	Verilog HDL code for multiplexer

module full_addER (A,B,Cin,Cout,Sum);
input A,B,Cin;
output Sum,Cout;
Wire S0,C0,C1;
 	
full adder
HA H1(A,B,S0,C0);
HA H2(S0,Cin,Sum,C1);
or (Cout,C0,C1)
endmodule
 	
module HA(A,B,S,C);
input A,B;
output S,C;
xor(S,A,B);
and(C,A,B);
endmodule
	module mux 2x1(D0,D1,S,Enbar,Y);
	input D0,D1,S,Enbar;
	output Y;
	and #7 (I3,D0,I1,I2);
	or #7 (Y,I3,I4);
	and #7 (I4,D1,S,I2);
	not #7 (I1,S);
	not #7 (I2,Enbar);
	endmodule

Kongunadu College of Engineering And Technology

(Autonomous)

Namakkal – Trichy Main Road, Thottiam

Department of Computer Science and Engineering

24EC304-Digital Logic and Computer Organization

Presented By

Ms.SUGANYA S

UNIT – III
COMPUTER FUNDAMENTALS

Functional Units of a Digital Computer: Von Neumann Architecture - Operation and Operands of Computer Hardware Instruction - Instruction Set Architecture (ISA): Memory Location, Address and Operation - Instruction and Instruction Sequencing - Addressing Modes, Encoding of Machine Instruction - Interaction between Assembly and High-Level Language.

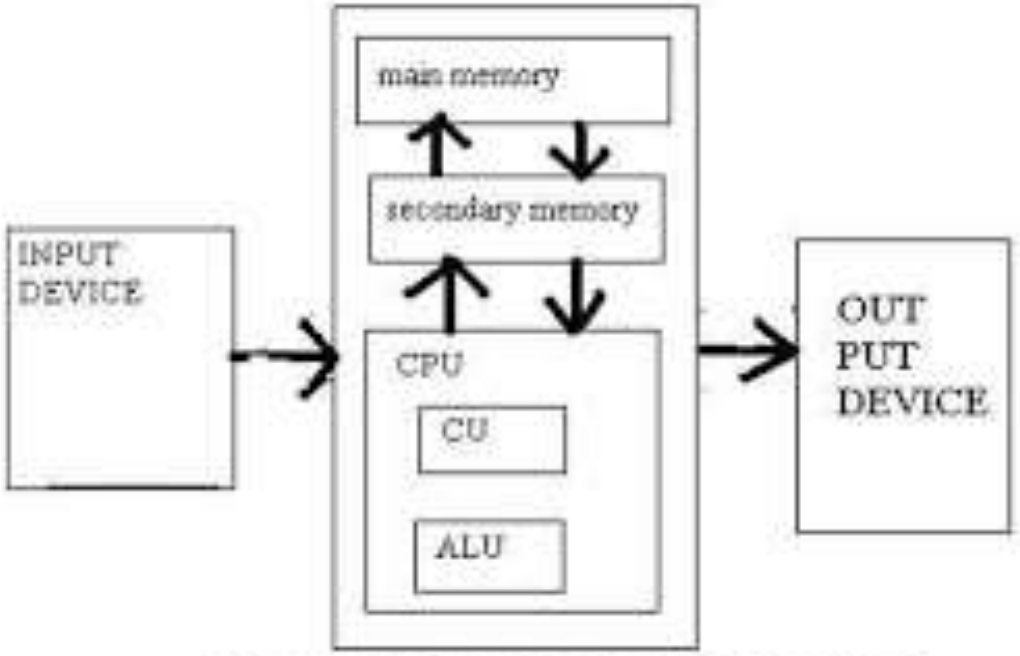
Functional Units of a Digital Computer

- A computer is an electronic device that processes data.
- It's composed of five main functional units that work together.
- These units are essential for fetching data, processing it, and producing output.

Functional Units of a Digital Computer(Contd)

- **Functional Units**:
- **Input Unit**: Takes data and instructions from the outside world. (e.g., Keyboard, Mouse)
- **Memory Unit**: Stores data and instructions.
- **Arithmetic & Logic Unit (ALU)**: Performs arithmetic (+, −, etc.) and logical operations (AND, OR, etc.).
- **Control Unit**: Directs all other units. It's the "brain" of the computer.
- **Output Unit**: Sends processed data to the outside world. (e.g., Monitor, Printer)

Functional Units of a Digital Computer(Contd)

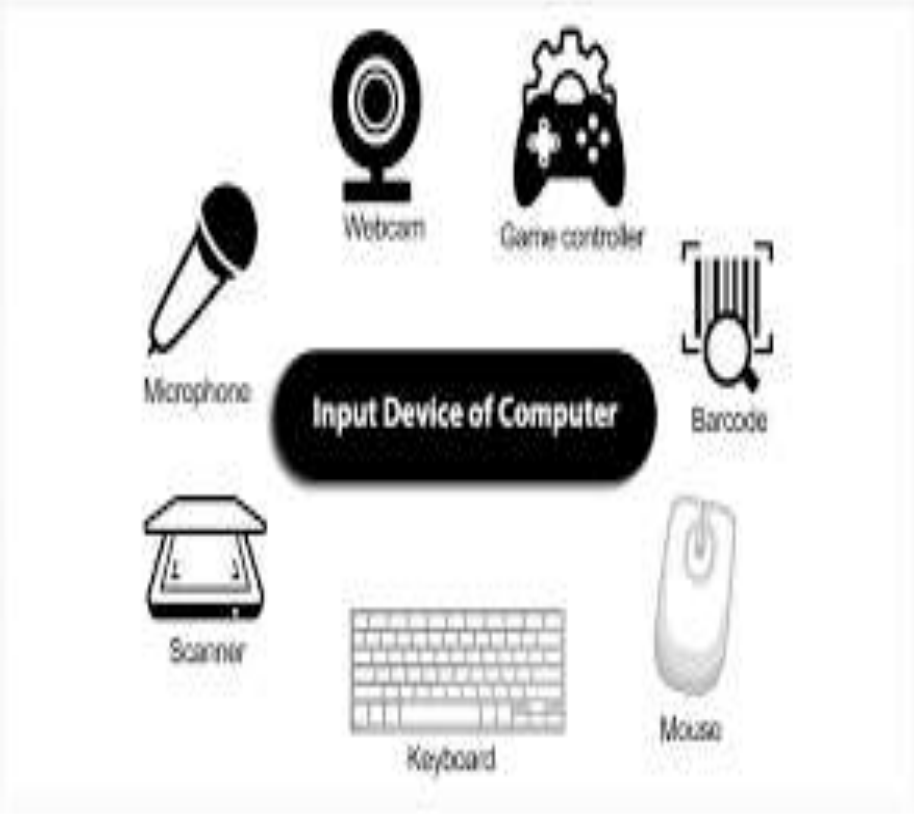


BLOCK DIAGRAM OF A DIGITAL COMPUTER

INPUT UNIT

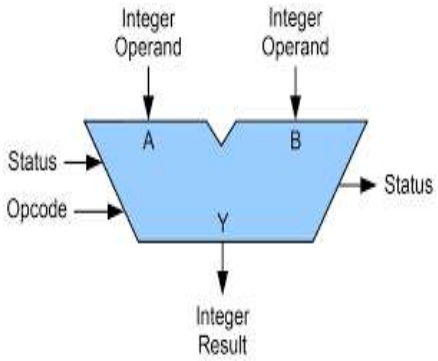
- The input unit is the first point of interaction between the user and the computer. It allows the user to provide instructions and data to the system using input devices such as keyboards, mice, scanners, and microphones. This unit is responsible for converting human-readable information into binary code, which is the language understood by computers. Once the information is digitized, it is passed either to the memory for storage or directly to the central processing unit (CPU) for immediate processing.

INPUT UNIT(Contd)



MEMORY UNIT

- The memory unit plays a pivotal role in storing both data and program instructions. In accordance with the Von Neumann model, a single memory unit holds all data and instructions, eliminating the need for separate storage locations.



MEMORY UNIT (Contd)

- The memory is typically divided into two main categories: primary and secondary memory. Primary memory, or main memory (often RAM), is fast and volatile, serving as the workspace for the CPU during processing. It temporarily holds data that the CPU frequently accesses. Secondary memory, like hard drives and SSDs, offers long-term storage for data and software. Memory is further structured using addresses, allowing the CPU to access specific data quickly and efficiently.

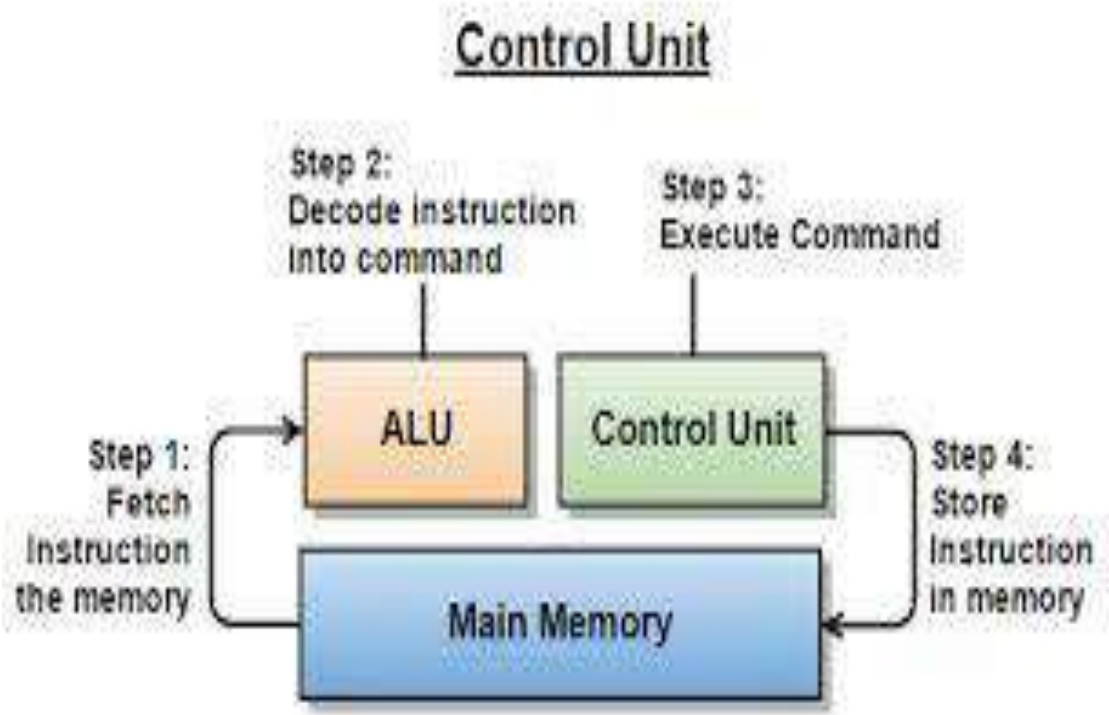
CONTROL UNIT

- The control unit acts as the coordinator of the entire computer system. Its primary function is to interpret instructions fetched from memory and to direct the operations of other components accordingly. While it does not process data itself, it manages the flow of information between the ALU, memory, and input/output devices.

CONTROL UNIT(Contd)

- The control unit executes the instruction cycle, which includes fetching an instruction from memory, decoding it to determine the required action, executing the instruction by signaling other units, and preparing to fetch the next instruction. It uses control signals to maintain synchronization and order among all components.

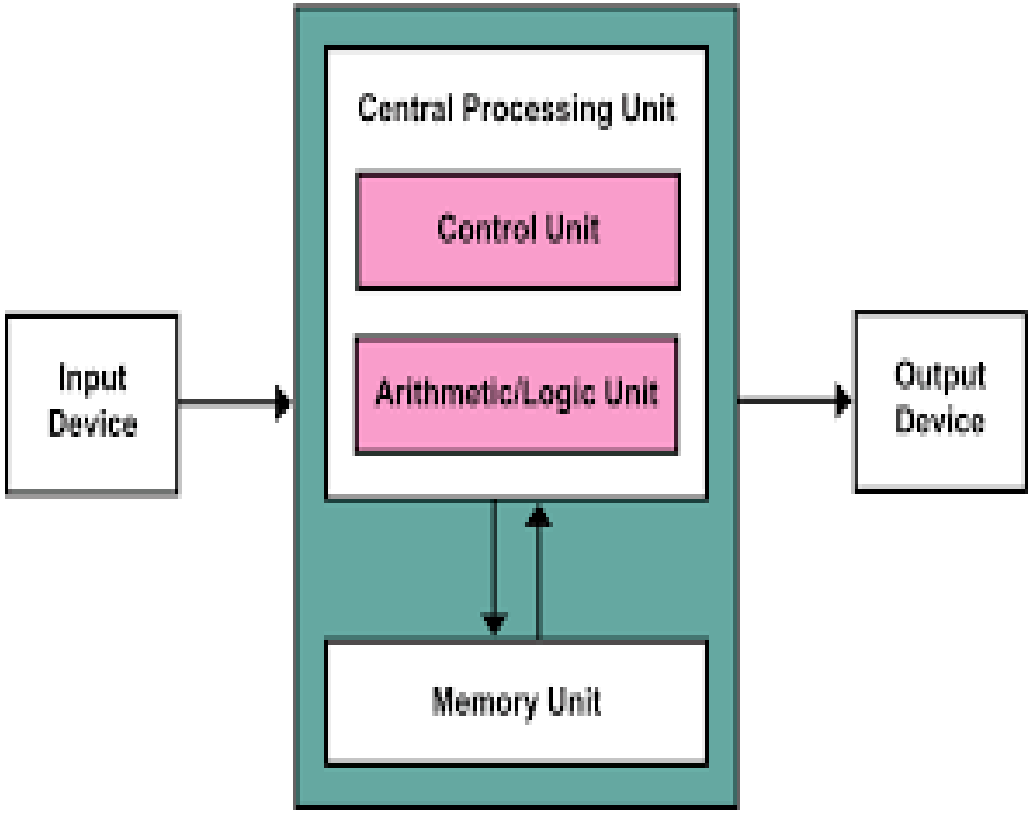
CONTROL UNIT(Contd)



OUTPUT UNIT

- The output unit is responsible for conveying processed data from the computer to the user or to another system.
- This unit converts digital information into human-readable or machine-usable formats.
- Common output devices include monitors, printers, speakers, and communication interfaces.

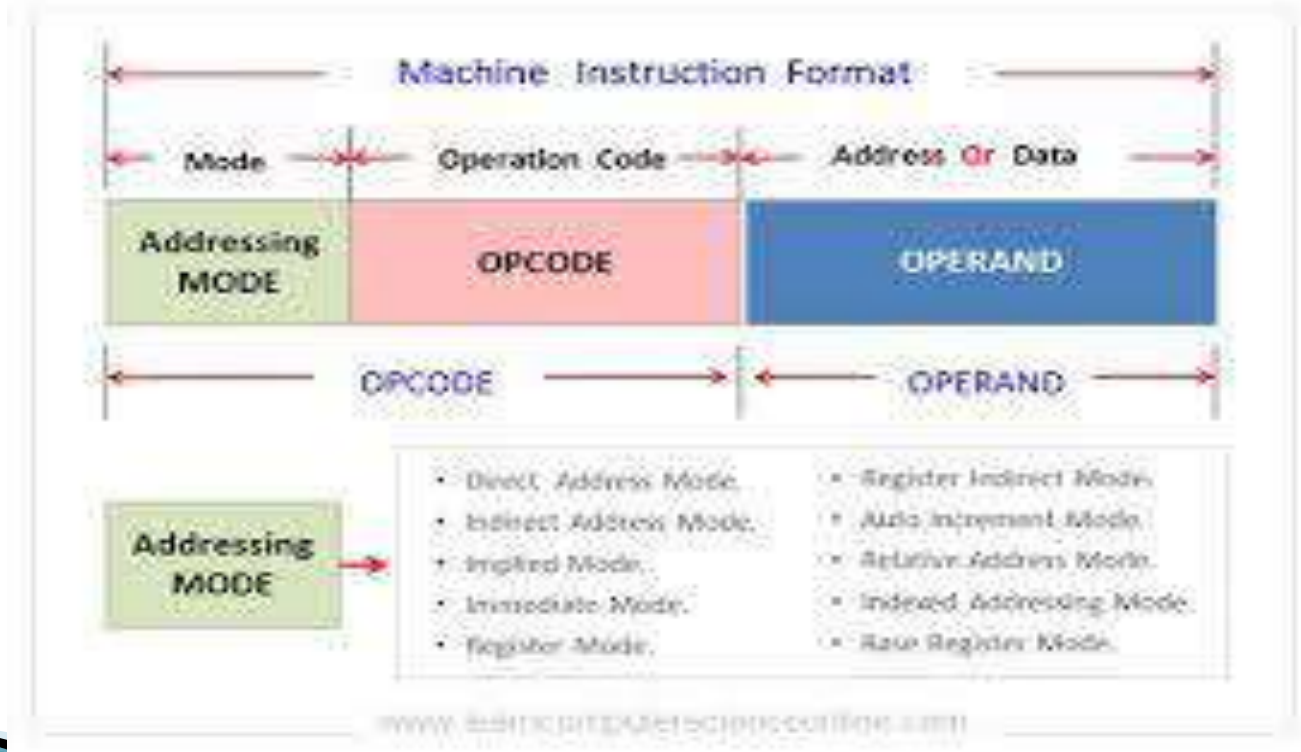
FUNCTIONAL UNIT



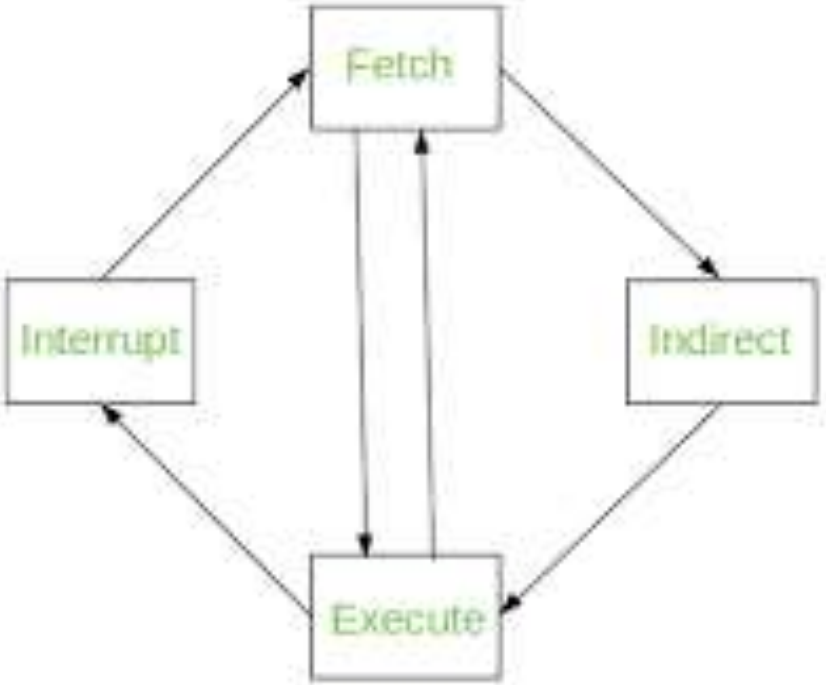
OPERATION AND OPERANDS OF COMPUTER HARDWARE INSTRUCTION

- In the realm of computer architecture, hardware instructions form the core set of commands that a processor can execute. Each instruction specifies an operation to perform, such as arithmetic or data transfer, along with the operands on which the operation acts.

OPERATION AND OPERANDS OF COMPUTER HARDWARE INSTRUCTION(Contd)



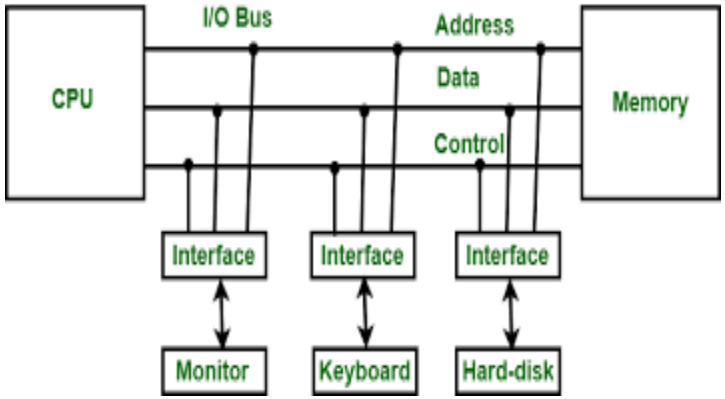
NATURE OF COMPUTER INSTRUCTION



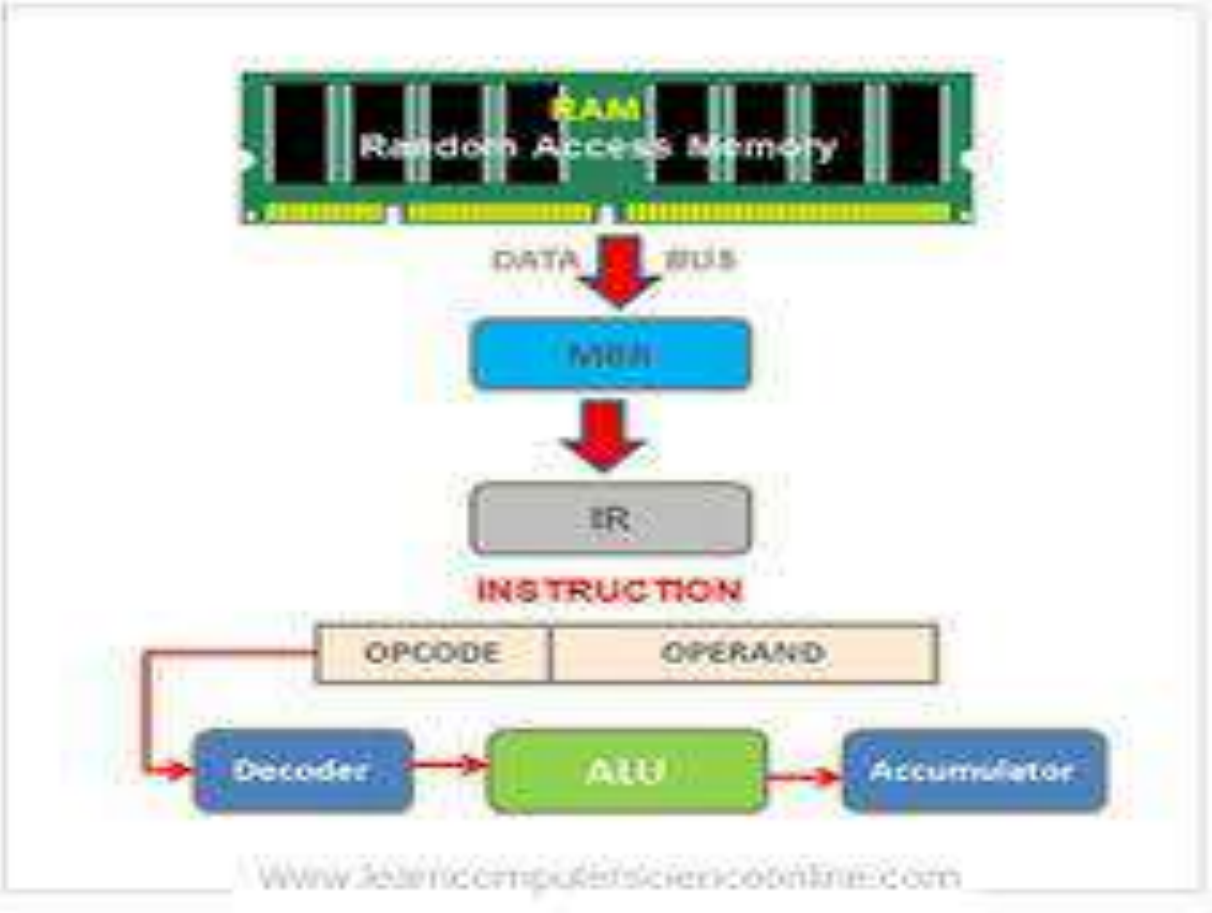
The Instruction Cycle

TYPES OF OPERATION

- Data Movement Operations
- Arithmetic and Logical Operations
- Control Flow Operations
- Input/Output Operations



TYPES OF OPERANDS



INSTRUCTION FORMATS

- Instructions come in various formats, depending on the architecture. A typical instruction format includes fields such as:
- Opcode: Specifies the operation.
- Source operands: Identifies the data inputs.
- Destination operand: Specifies where the result should be stored.
- Addressing mode bits: Indicate how operands should be interpreted.

- Different architectures use different instruction lengths (fixed or variable). For example, RISC (Reduced Instruction Set Computing) typically uses fixed-length instructions, while CISC (Complex Instruction Set Computing) may have variable-length instructions.

VON NEUMANN ARCHITECTURE – OVERVIEW

- The **Von Neumann Architecture** is a design model for digital computers.
- Proposed by John von Neumann in 1945
- Stored Program Concept
- Sequential instruction execution

- Its key feature is the **shared memory** for both data and instructions.
- This is a fundamental concept for how most modern computers operate.

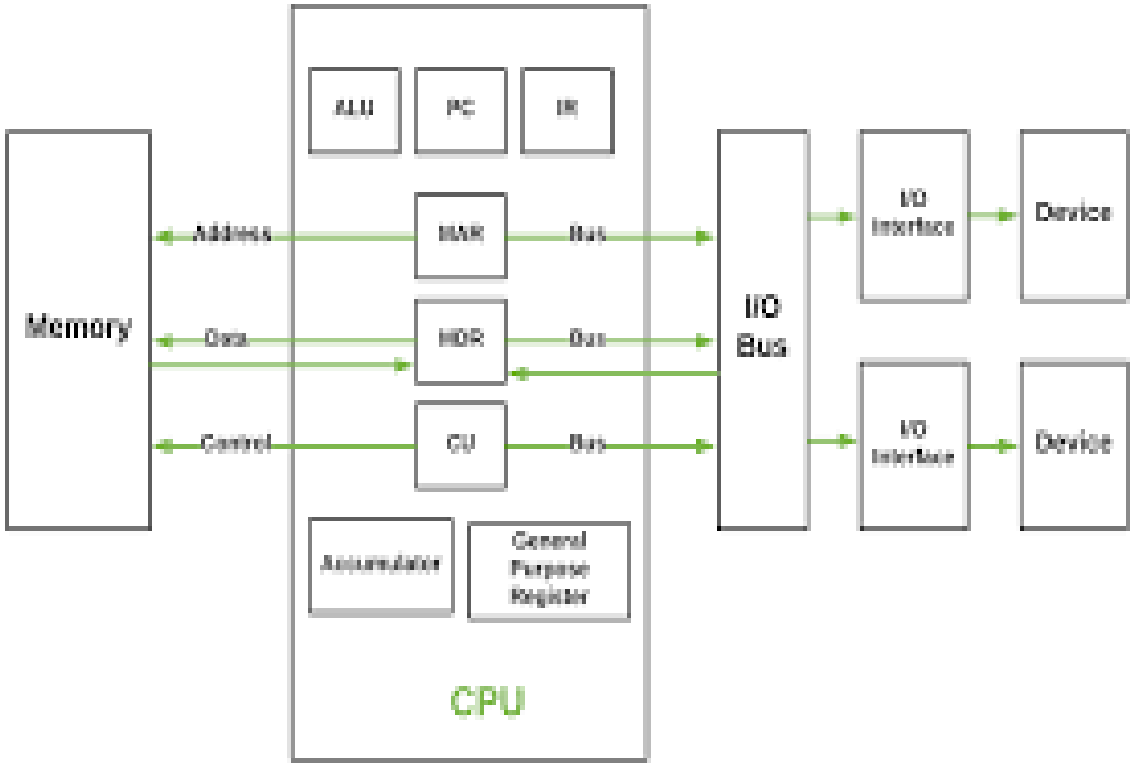
VON NEUMANN ARCHITECTURE – COMPONENTS

- CPU (ALU + CU)
- Memory Unit
- Input/Output Devices
- Buses: Data, Address, Control

VON NEUMANN ARCHITECTURE (Contd)

- **Key Features**:
- **Stored-Program Concept**: The instructions (program) are stored in the memory, just like data. This allows the computer to be reprogrammed.
- **Single Bus**: A single pathway (bus) is used to fetch both instructions and data from memory. This is a potential bottleneck, known as the "Von Neumann Bottleneck."
- **Operation Cycle (Fetch-Decode-Execute)**:
- **Fetch**: The Control Unit fetches the next instruction from memory.
- **Decode**: The Control Unit decodes the instruction to determine what action to perform.
- **Execute**: The Control Unit directs the ALU and other components to perform the action.

VON NEUMANN ARCHITECTURE (Contd)



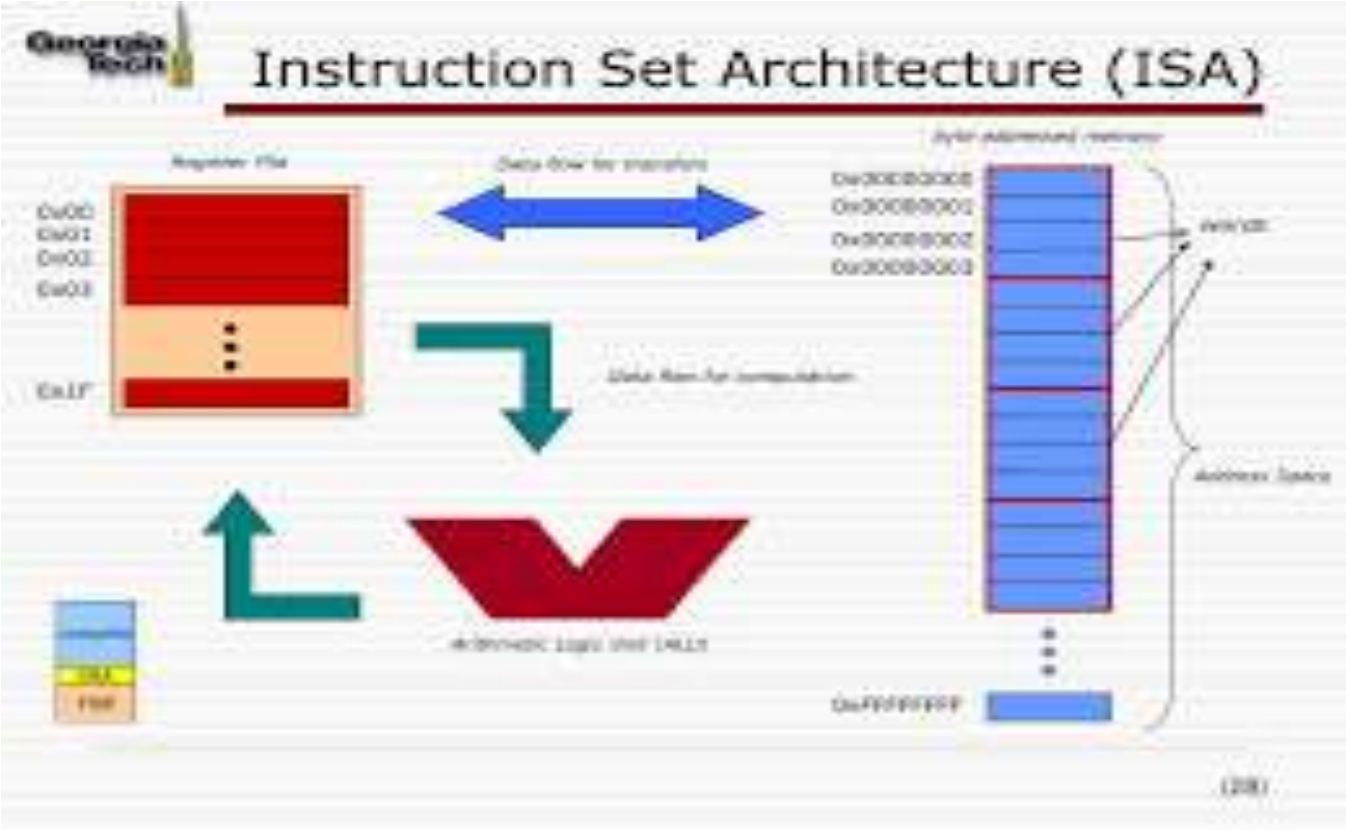
INSTRUCTION AND OPERANDS

- Instruction: Command to CPU
- **Opcode (Operation Code):**
- Specifies the operation to be performed (e.g., ADD, SUB, LOAD, STORE).
- **Operands:**
- The data or address on which the operation is to be performed.
- Operands can be registers, memory locations, or immediate values.
- **Example:** ADD R1, R2, R3
- **Opcode:** ADD
- **Operands:** R1, R2, R3
- **Meaning:** Add the contents of registers R2 and R3 and store the result in register R1.

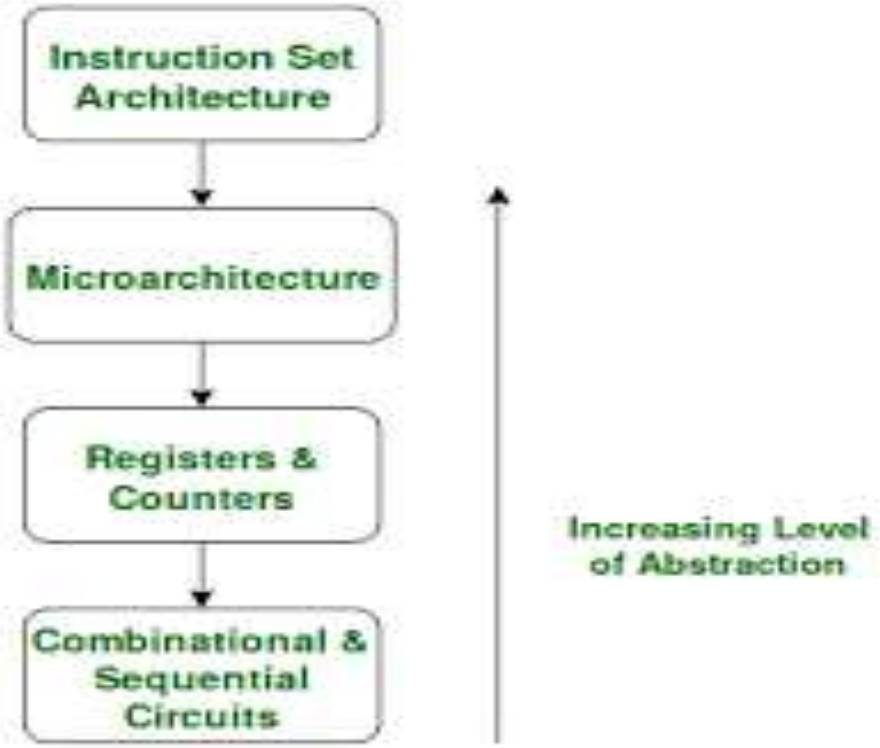
INSTRUCTION SET ARCHITECTURE (ISA)

- The **ISA** is the interface between the software and the hardware.
- It's the complete set of instructions that a particular processor can execute.
- It defines the machine language and the fundamental commands for a processor.
- **Components of ISA:**
- **Instruction Set:** The set of all available opcodes.
- **Registers**: The processor's internal storage locations.
- **Memory Model**: How memory is organized and addressed.
- **Addressing Modes**: How the location of operands is specified.
- **Examples of ISAs:**
- x86-64 (used in most desktops and laptops)
- ARM (used in smartphones and tablets)

ISA (Contd)



ISA (Contd)


Instruction Set Architecture
Microarchitecture
Registers & Counters
Combinational & Sequential Circuits
Increasing Level of Abstraction

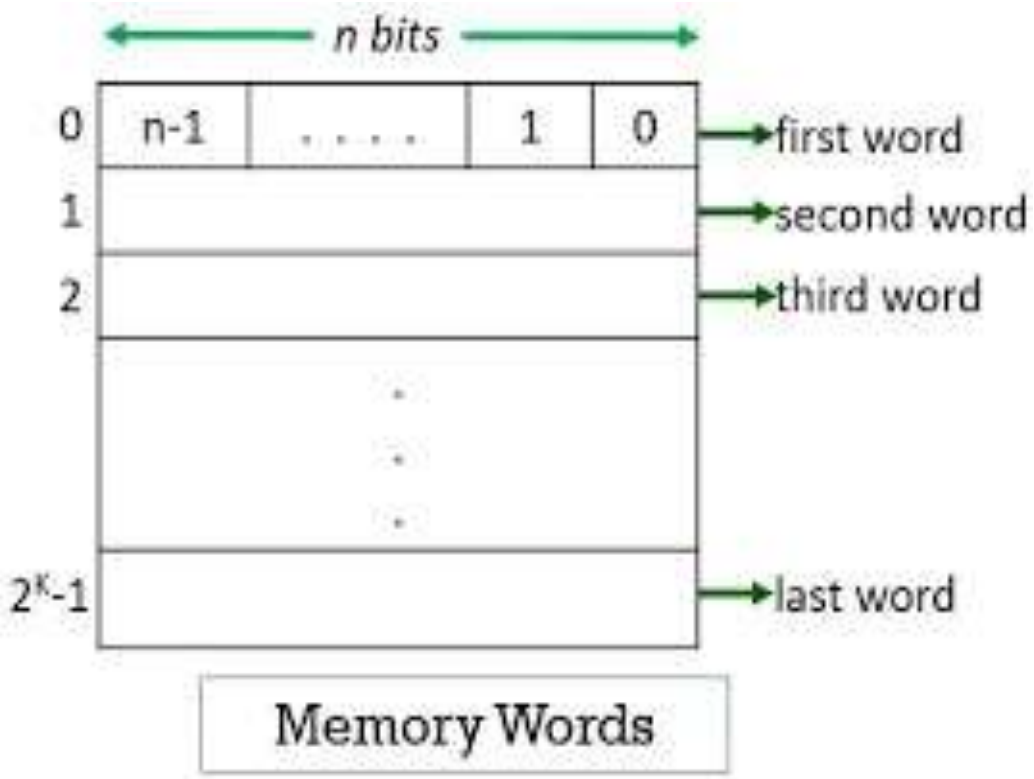
MEMORY LOCATION, ADDRESS, AND OPERATION

- Computer memory is like a large array of storage cells.
- Each cell has a unique number called an **address**.
- The address is used to identify and access the data stored in that location.
- **Operations on Memory**:
- **Read (Load)**: Copying data from a memory location to a processor register.
- **Write (Store)**: Copying data from a processor register to a memory location.

MEMORY LOCATION, ADDRESS, AND OPERATION (Contd)

- Program code (instructions)
- Variables (integers, floats, characters)
- Pointers (addresses of other memory locations)
- Data structures (arrays, structures, objects)
- Different

MEMORY LOCATION, ADDRESS, AND OPERATION (Contd)



MEMORY ADDRESSING MECHANISMS

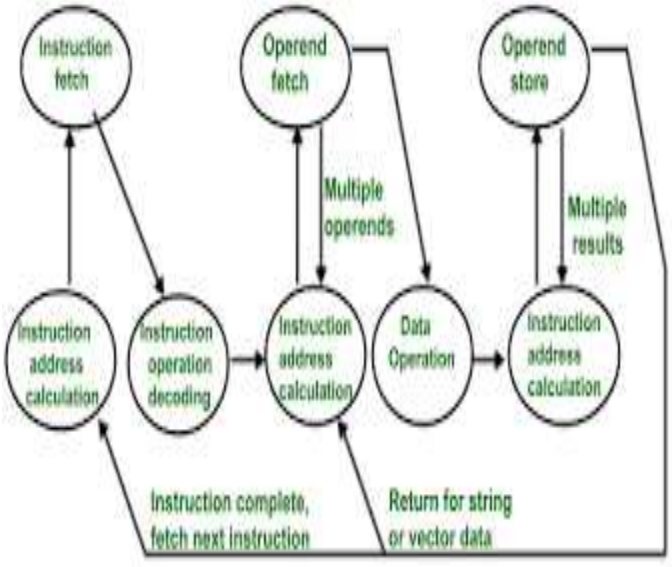
- One of the most critical aspects of ISA is the way it allows programs to access memory—this is defined through addressing modes. Addressing modes determine how the effective address of an operand is calculated. Common addressing modes include:
- **Immediate Addressing**: The operand is part of the instruction itself. Example: MOV R1, #5 (moves the value 5 into register R1).
- **Register Addressing**: The operand is in a register. Example: ADD R1, R2 (adds the contents of R2 to R1).
- **Direct Addressing**: The instruction specifies the memory address directly. Example: LOAD R1, 1000 (loads value from address 1000).
- **Indirect Addressing**: The address of the operand is held in a register or memory. Example: LOAD R1, (R2) (loads from address stored in R2).

INSTRUCTION AND INSTRUCTION SEQUENCING

- Instruction sequencing is a fundamental aspect of computer architecture, governing how a processor retrieves, decodes, and executes instructions in a predefined or dynamically altered order.

INSTRUCTION AND INSTRUCTION SEQUENCING (Contd)

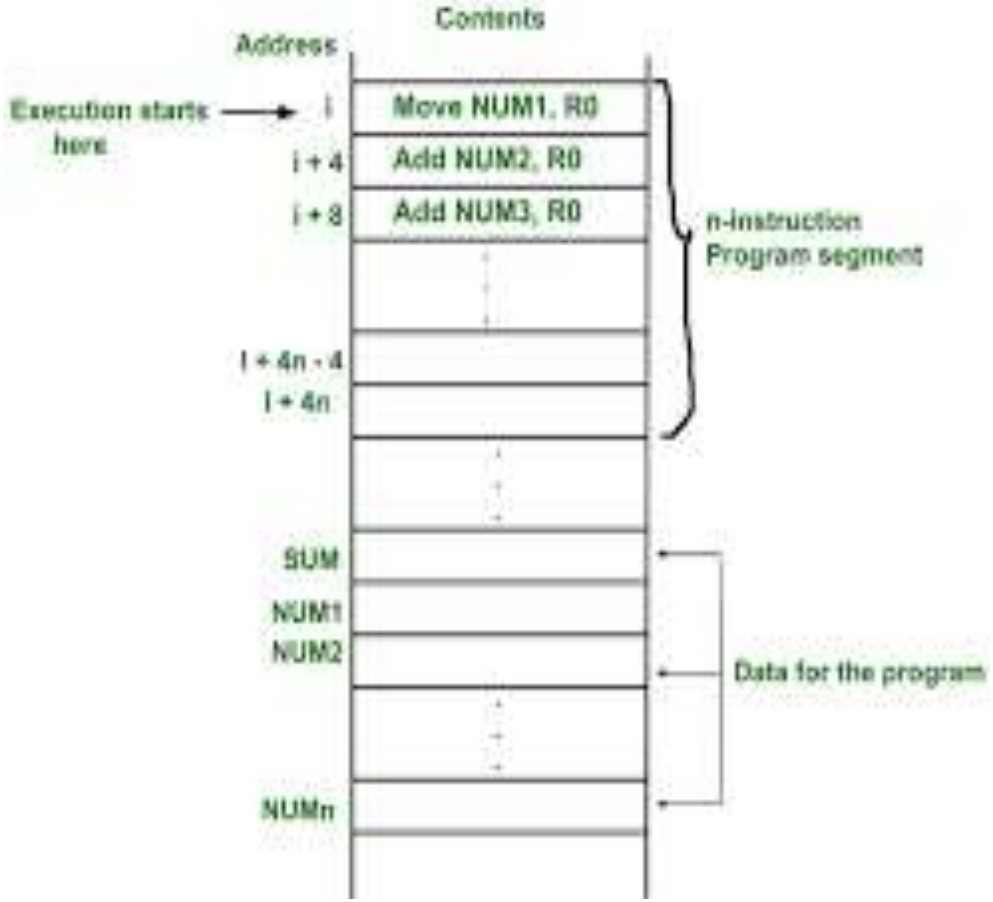
- Understanding Computer Instructions A computer instruction is a binary-encoded command that tells the CPU to perform a specific task.



TYPES OF INSTRUCTIONS

- Instructions can be categorized based on the operation they perform:
- Data Movement Instructions: Move data between registers, memory, and I/O (e.g., MOV, LOAD, STORE).
- Arithmetic and Logical Instructions: Perform calculations (e.g., ADD, SUB, AND, OR).
- Control Flow Instructions: Alter the flow of execution (e.g., JMP, CALL, RET).
- Comparison Instructions: Compare values to set condition flags (e.g., CMP).
- Input/Output Instructions: Interact with peripherals (e.g., IN, OUT).

TYPES OF INSTRUCTIONS (Contd)

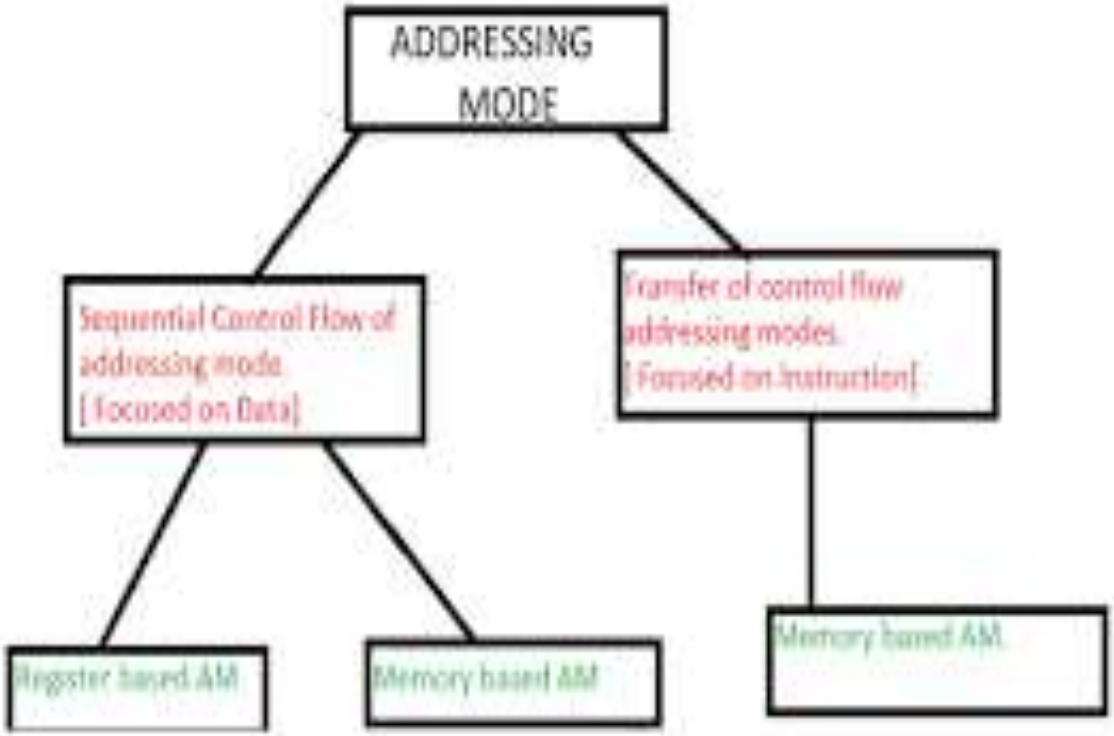


ADDRESSING MODES

- **Addressing Modes** are different ways of specifying the location of an operand.
- They provide flexibility and efficiency in accessing data.
- **Common Addressing Modes**:
- **Immediate**: The operand is a constant value within the instruction itself.
- **Register**: The operand is in a specified register.
- **Direct (Absolute)**: The address of the operand is given directly in the instruction.
- **Indirect**: The instruction contains the address of a memory location, which in turn holds the address of the operand.
- **Indexed**: The address of the operand is calculated by adding a constant offset to a value in an index register.

- Immediate, Direct, Indirect
- Register, Indexed, Base addressing

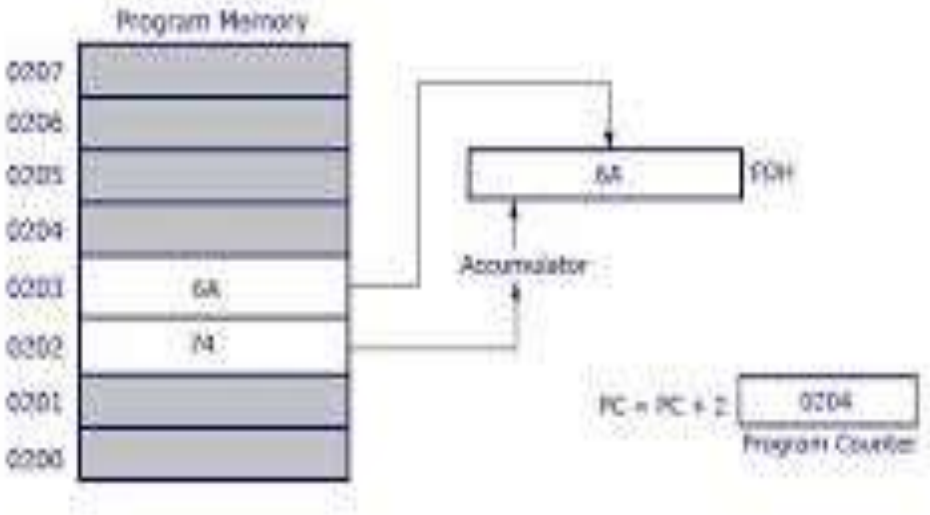
ADDRESSING MODES(Contd)



ADDRESSING MODES(Contd)

Immediate Addressing Mode

Instruction	Opcode	Bytes	Cycles
MOV A, #6AH	74H	2	1



TYPES OF ADDRESSING MODES

- Each addressing mode has specific characteristics suited to certain tasks. The following are some of the most commonly used addressing modes:
- *a. Immediate Addressing*
- In this mode, the operand is specified explicitly in the instruction. It is useful for loading constants directly.
- **Example: MOV R1, #5 (Move the constant value 5 into register R1).**
- **Advantage: Fast and requires no memory access beyond the instruction itself.**
- **Limitation: Operand size is limited by the instruction length.**

TYPES OF ADDRESSING MODES (Contd)

- *Register Addressing*
- Here, the operand is located in a register. The instruction specifies which register to use.
- **Example: ADD R2, R3 (Add the values in R2 and R3).**
- **Advantage: Fastest access since registers are internal to the CPU.**

- *c. Direct (Absolute) Addressing*
- The instruction contains the actual memory address of the operand.
- **Example: LOAD R1, 5000 (Load the data at memory address 5000).**
- **Advantage: Simple and easy to implement.**
- **Limitation: Limited flexibility for programs needing dynamic memory access.**
- 92

TYPES OF ADDRESSING MODES (Contd)

- *Indirect Addressing*
- The instruction refers to a memory location that contains the address of the operand.
- **Example: LOAD R1, (R2) (Use the contents of R2 as a pointer to memory).**
- **Advantage: Supports pointer-based programming.**
- **Limitation: Slower due to multiple memory accesses.**

- *e. Indexed Addressing*
- This mode calculates the effective address by adding a base address and an index value.
- **Example: LOAD R1, 100(R3) (Address = 100 + contents of R3).**
- **Used for: Accessing array elements.**

TYPES OF ADDRESSING MODES (Contd)

- *Relative Addressing*
- This uses the program counter and a constant value to determine the address.
- **Example: JMP +6 (Jump forward by 6 instructions).**
- **Used in: Branching and loops.**

- *h. Stack Addressing*
- Data is implicitly at the top of the stack; no operand address is explicitly given.
- **Common in: Zero-address machines or stack-based processors.**
- **Used for: Function calls, returns, and expression evaluation.**

TYPES OF ADDRESSING MODES (Contd)

- ***Base-Register Addressing***
- Similar to indexed addressing, but focuses more on modular programming where R3 may point to the beginning of a structure.
- **Example: LOAD R1, 0(R3)**

- Significance of Addressing Modes 93
- Addressing modes provide a balance between hardware efficiency and software flexibility. A processor with multiple addressing modes can:
- Support high-level constructs like arrays, pointers, and loops.
- Optimize code size and runtime.
- Enable easier compilation from high-level languages.

ENCODING OF MACHINE INSTRUCTIONS

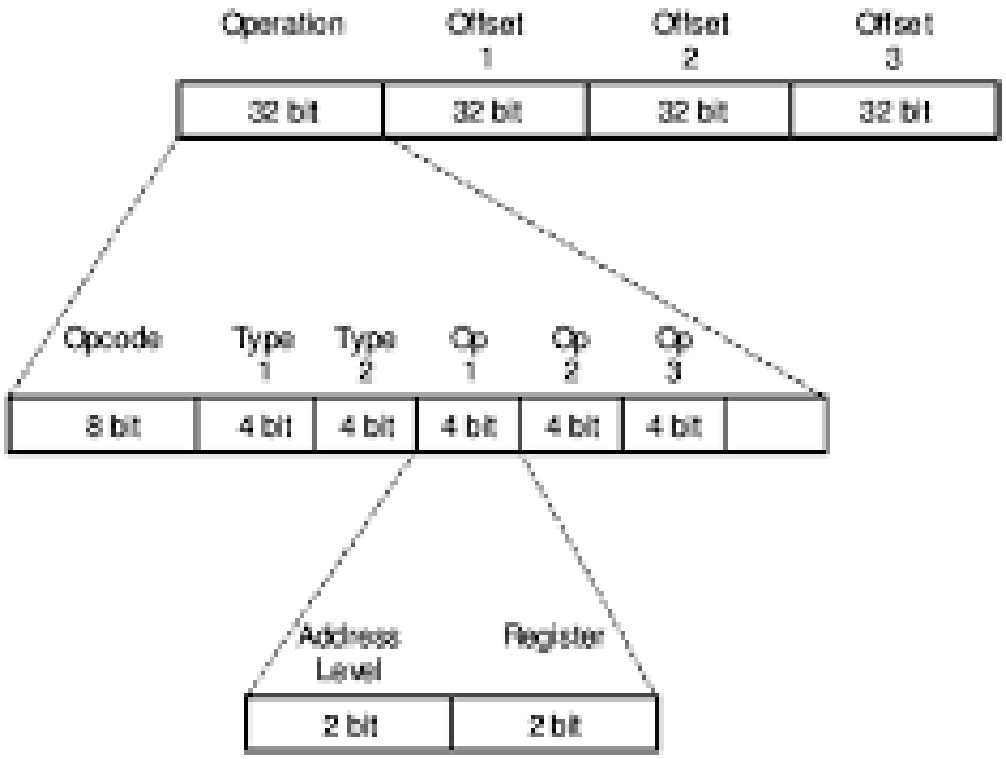
- Instructions are stored in memory as a sequence of bits (0s and 1s).
- This is called **machine code.**
- The process of converting a human-readable instruction (e.g., ADD R1, R2) into its binary representation is called **encoding.**
- **Encoding Process**:
- Each opcode is assigned a specific binary code.
- Each register is also assigned a binary code.
- The final machine instruction is a combination of these binary codes.
- The ISA defines the exact format and length of the encoded instructions.

ENCODING OF MACHINE INSTRUCTIONS(Contd)

Encoding of machine instructions (Home Practice)

- **Provide the encoding for following instructions: [Home Exercise]**
  - I0: R31 = R27 Addition R9;
  - I1: A30 = R27 Division R9;
  - I2: R30 = A27 XOR R9;
  - I3: A53 = A27 Multiplication A9;
  - I4: A80 = R24 Subtraction 127;
  - I5: R8 = 90 AND 256;

ENCODING OF MACHINE INSTRUCTIONS(Contd)



INSTRUCTION FORMAT AND ENCODING

- Instruction encoding defines how each instruction is represented in binary. A machine instruction typically consists of several fields:
- Opcode: Specifies the operation (e.g., ADD, MOV).
- Operands: Specify source and destination data.
- Addressing mode bits: Indicate how to interpret operand references.
- Immediate or displacement values: Provide data directly or support relative addressing.

TYPES OF INSTRUCTION ENCODING (Contd)

- *a. **Fixed-Length Encoding***
- All instructions occupy the same number of bits (e.g., 32 bits).
- Common in: RISC architectures (MIPS, ARM).
- Advantage: Easier to decode; consistent fetch and execute cycles.
- Disadvantage: May waste bits for simple instructions.

TYPES OF INSTRUCTION ENCODING (Contd)

- *b. **Variable-Length Encoding***
- Instructions vary in size based on complexity (1 to 15 bytes in x86). 94
- Common in: CISC architectures (Intel x86).
- Advantage: Efficient memory use; complex operations encoded compactly.
- Disadvantage: Slower decoding, more complex CPU design.

TYPES OF INSTRUCTION ENCODING (Contd)

- *c. Hybrid Encoding*
- Combines features of fixed and variable-length encoding.
- Example: ARM Thumb mode offers both 16-bit and 32-bit instructions.

- Instruction Format Types
- Various formats exist to accommodate the types of operations:
- *- R-Type (Register): All operands are in registers.*
- Example: ADD R1, R2, R3

TYPES OF INSTRUCTION ENCODING (Contd)

- *- **I-Type (Immediate):** Includes an immediate value.*
- Example: ADDI R1, R2, #10

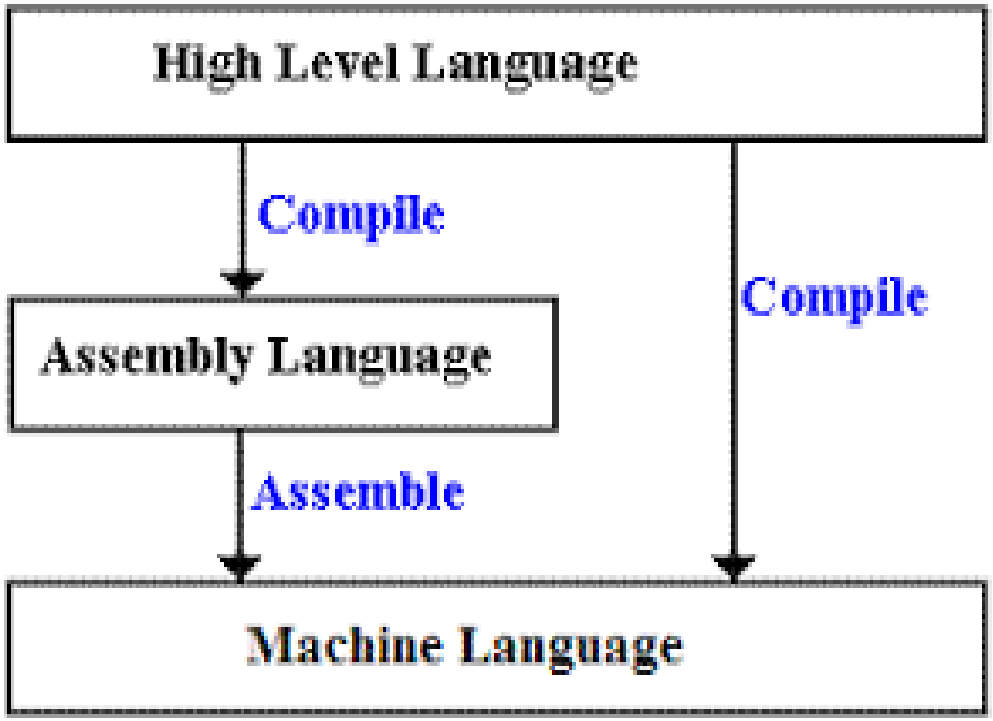
- *- **J-Type (Jump):** Contains a jump address.*
- Example: JMP 1024

- These formats are standard in RISC-based architectures like MIPS and simplify the decoding process.

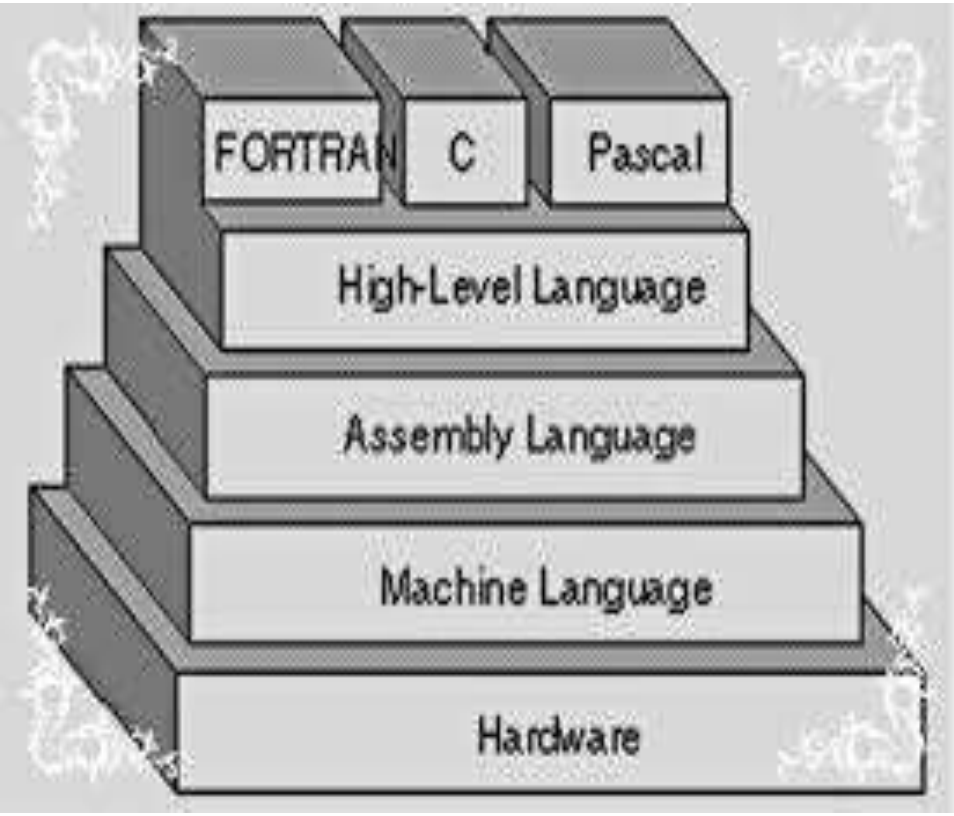
ASSEMBLY & HIGH-LEVEL LANGUAGE INTERACTION

- **High-Level Languages (HLL)** (e.g., C++, Java, Python) are abstract and human-friendly.
- A **compiler** translates HLL code into a lower-level language.
- The final output is often a form of **assembly language**.
- **Assembly Language**:
- A low-level language that is a human-readable representation of machine code.
- Each assembly instruction corresponds to a single machine instruction.
- An **assembler** translates assembly code into machine code.
- **The Bridge**:
- High-level code is compiled into assembly code, which is then assembled into machine code for the processor to execute. This two-step process allows programmers to write powerful applications without needing to know the intricate details of the specific ISA.

ASSEMBLY & HIGH-LEVEL LANGUAGE INTERACTION(Contd)



ASSEMBLY & HIGH-LEVEL LANGUAGE INTERACTION(Contd)



ADVANTAGES OF USING ASSEMBLY WITH HIGH-LEVEL LANGUAGES

- Performance: Assembly enables fine-grained control for time-critical code.
- Hardware Access: Directly control registers, I/O ports, or special CPU instructions.
- Compact Code: Assembly can reduce instruction count and memory footprint.
- Debugging and Analysis: Understand what compiled code does under the hood.

DISADVANTAGES AND CHALLENGES

- Complexity: Writing and understanding assembly is difficult and time-consuming.
- Portability: Assembly is architecture-specific.
- Maintenance: Harder to update or modify than high-level code.
- Security Risks: Poorly written assembly can introduce low-level vulnerabilities.

THANK YOU

Kongunadu College of Engineering And Technology

(Autonomous)

Namakkal – Trichy Main Road, Thottiam

Department of Computer Science and Engineering

24EC304-Digital Logic and Computer Organization

Presented By

Ms.SUGANYA S

UNIT IV
PROCESSOR

Instruction Execution – Building a Data Path – Designing a control Unit – Hardwired Control, Microprogrammed Control – Pipelining – Data Hazard – Control Hazards.

INSTRUCTION EXECUTION

- **INSTRUCTIONS :**
- An instruction is a piece of a program that performs an operation issued by the computer processor.
- Every instruction is defined by the instruction set of the processor.

ELEMENTS OF INSTRUCTION

- Operation code: Specifies the operation to be performed. The operation is specified by binary code, hence the name operation code or simply opcode.
- Source / Destination operand: The source/destination operand field directly specifies the source/destination operand for the instruction.
- Source operand address: The operation specified by the instruction may require one or more source operands.
- Destination operand address: The operation executed by the CPU may produce result. Usually, the result is stored in the destination operand.
- ext instruction address: The next instruction address tells the CPU from where to fetch the next instruction after completion of execution of current instruction.

INSTRUCTION SET

- A list of all the instructions with all their variants that can be executed by a processor is called instruction set. It is a group of commands defined by the processor in machine understandable language.

Format

- An instruction has three fields, namely- 101
- Operation code (Opcode) specifies which type of operation to be performed.
- Mode Field specifies the way the operand or effective address is determined.
- Address Field specifies memory address or a processor register.


Opcode
Mode field
Address field

BUILDING A DATAPATH

- Datapath :
- The Datapath is the pathway that the data takes through the CPU. As the data travels through the data path, the control unit regulates interaction between the data path and the data according to the instruction being executed.
- The data path consists of functional units that perform data processing operations such as addition, subtraction, logical AND, OR, inverting, and shifting.

Datapath Elements

- A data path element is a functional unit used to operate on or hold data within a processor.
- The data path elements are:
- The instruction memory
- The data memory
- The register file
- The arithmetic logic unit (ALU)
- Adders

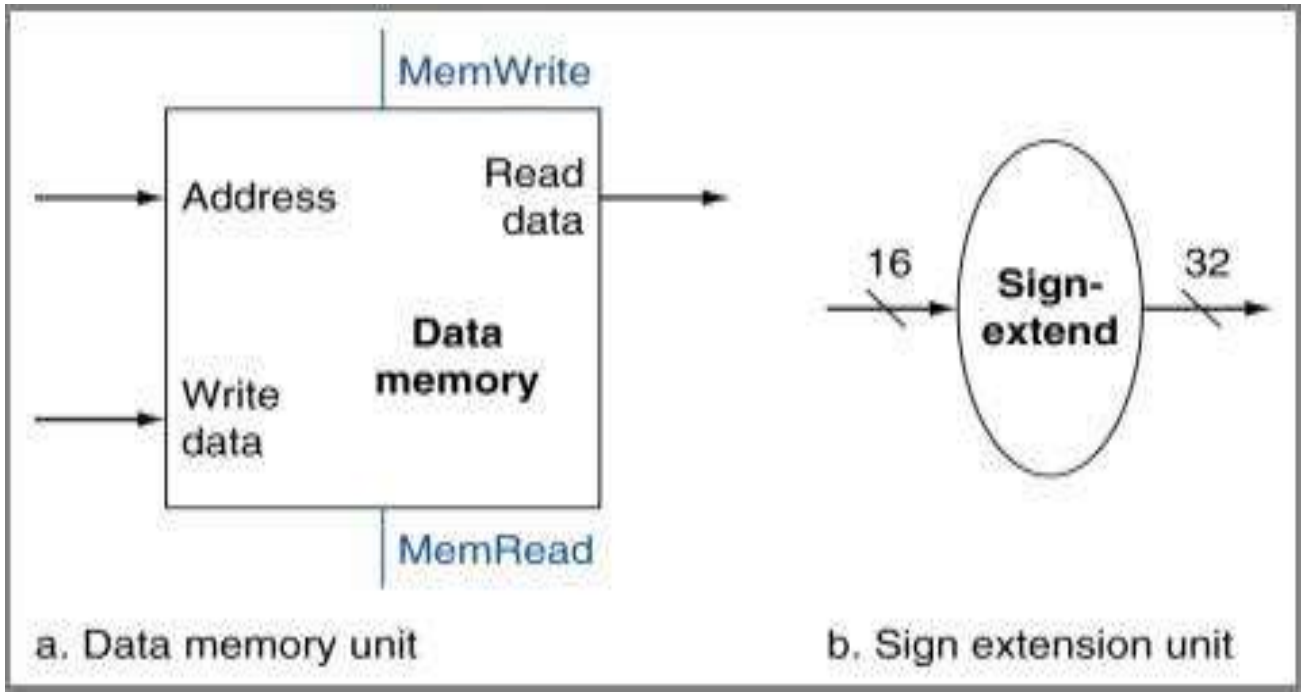
Instruction Memory

- A memory unit that is used to store the instructions of a program and supply instructions given an address

Branch Instructions

- The beq instruction has three operands, two registers that are compared for equality,anda16-bit offset used to compute the branch target address relative to the branch instruction address.

Branch Instructions



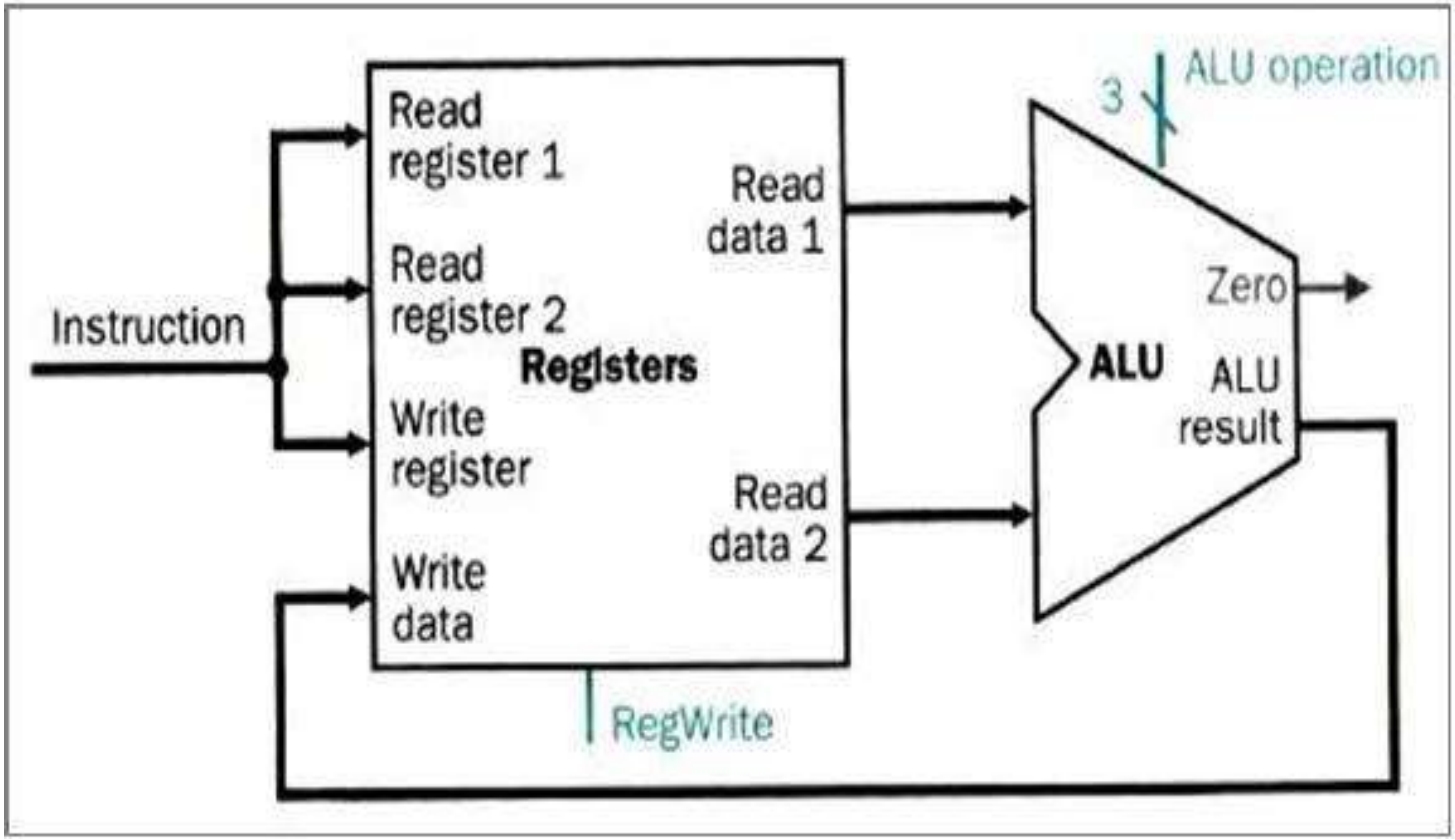
Delayed Branch

- Branches are delayed if the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.
- When the condition is false, the execution looks like a normal branch.
- When the condition is true, a delayed branch first executes the instruction immediately following the branch in sequential instruction order before jumping to the specified branch target address.

Datapath for R-type instructions

- The following additional components are needed for the implementation of the data path for R-format instructions.
- Register file
- ALU
- The ALU accepts the input from the Data Read ports of the register file.
- The register file is written by the ALU in combination with the Reg Write signal.

Datapath for R-type instructions (Contd)



Datapath for Load/Store instruction

- The following additional components are added to build the datapath for load and store instruction.
- Data Memory unit
- Sign Extension unit

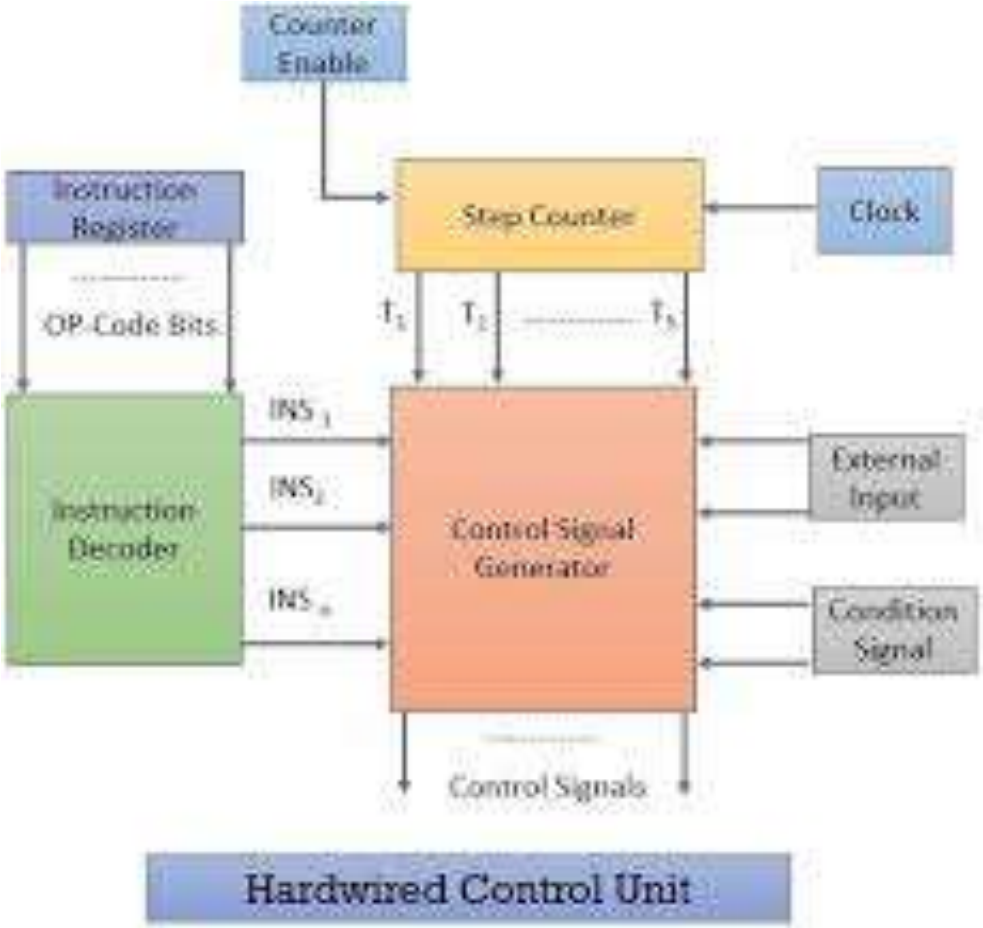
DESIGNING A CONTROL UNIT

- Control units are designed to manage instruction execution, and can be implemented using either hardwired logic or microprogrammed approaches. Hardwired control units use physical hardware like gates and flip-flops to generate control signals, making them fast but inflexible. Microprogrammed control units, on the other hand, store control signals in memory as microinstructions, allowing for flexibility and easier modification, but potentially sacrificing speed.

HARDWIRED CONTROL

- A hardwired control is a mechanism of producing control signals using Finite State Machines (FSM) appropriately. It is designed as a sequential logic circuit. The final circuit is constructed by physically connecting the components such as gates, flip flops, and drums. Hence, it is named a hardwired controller.

HARDWIRED CONTROL (Contd)



HARDWIRED CONTROL (Contd)

- Some of the methods that have come up for designing the hardwired control logic are as follows –
- Sequence Counter Method − This is the most convenient method employed to design the controller of moderate complexity.
- Delay Element Method − This method is dependent on the use of clocked delay elements for generating the sequence of control signals.
- State Table Method − This method involves the traditional algorithmic approach to design the Notes controller using the classical state table method.

HARDWIRED CONTROL- Advantages

- Because of the use of combinational circuits to generate signals, Hardwired Control Unit is fast.
- ▪ It depends on number of gates, how much delay can occur in generation of control signals.
- ▪ It can be optimized to produce the fast mode of operation.
- ▪ Faster than micro- programmed control unit.
- ▪ It does not require control memory.

MICROPROGRAMMED CONTROL

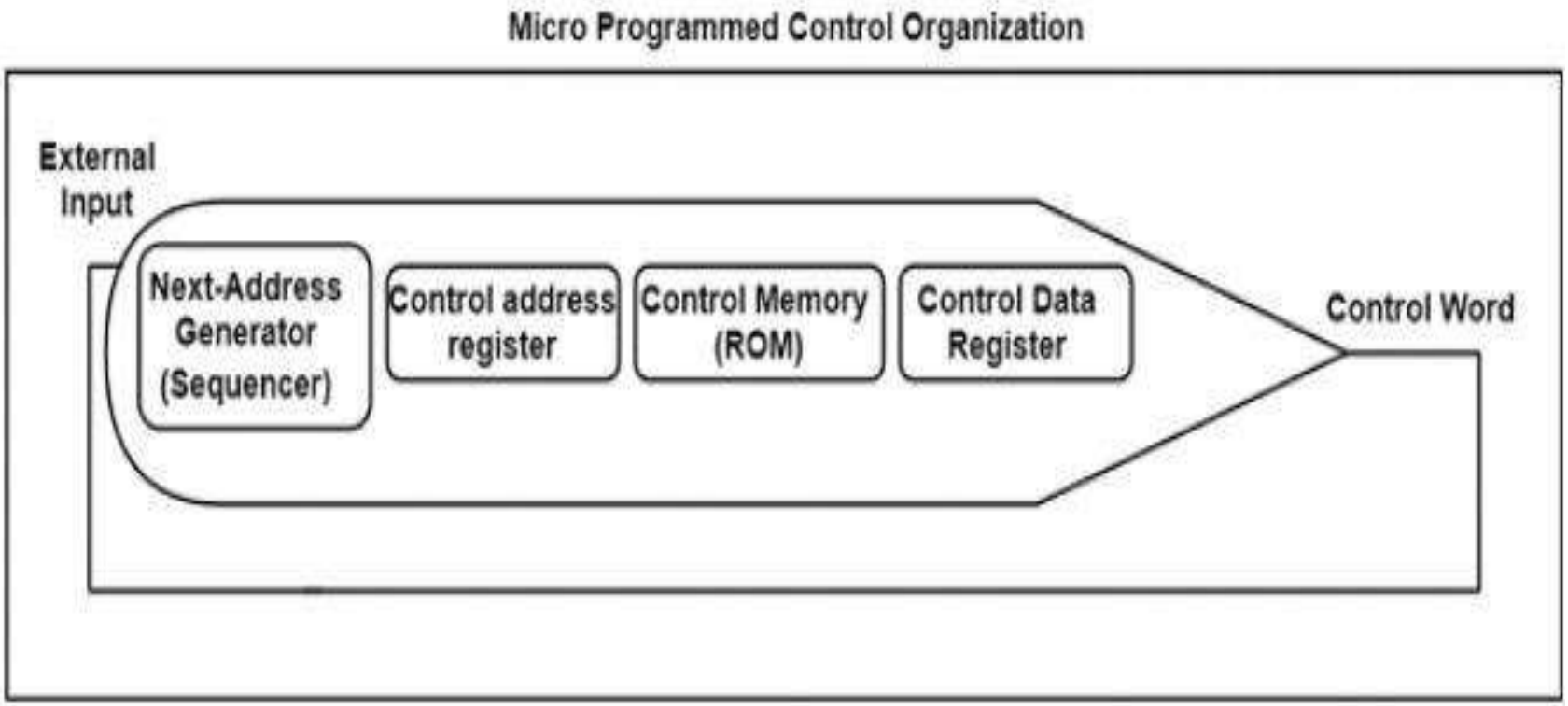
- A control unit whose binary control values are saved as words in memory is called a microprogrammed control unit.
- A controller results in the instructions to be implemented by constructing a definite collection of signals at each system clock beat. Each of these output signals generates one micro-operation including register transfer. Thus, the sets of control signals are generated definite micro-operations that can be saved in the memory.

MICROPROGRAMMED CONTROL(Contd)

- There are the following steps followed by the microprogrammed control are

- It can execute any instruction. The CPU should divide it down into a set of sequential operations. This set of operations are called microinstruction. The sequential micro-operations need the control signals to execute.
- Control signals saved in the ROM are created to execute the instructions on the data direction. These control signals can control the micro-operations concerned with a microinstruction that is to be performed at any time step.

MICROPROGRAMMED CONTROL(Contd)



PIPELINING

- Pipelining is an implementation technique in which multiple instructions are overlapped in execution. This enables the processors to complete the tasks faster.

PIPELINING(Contd)

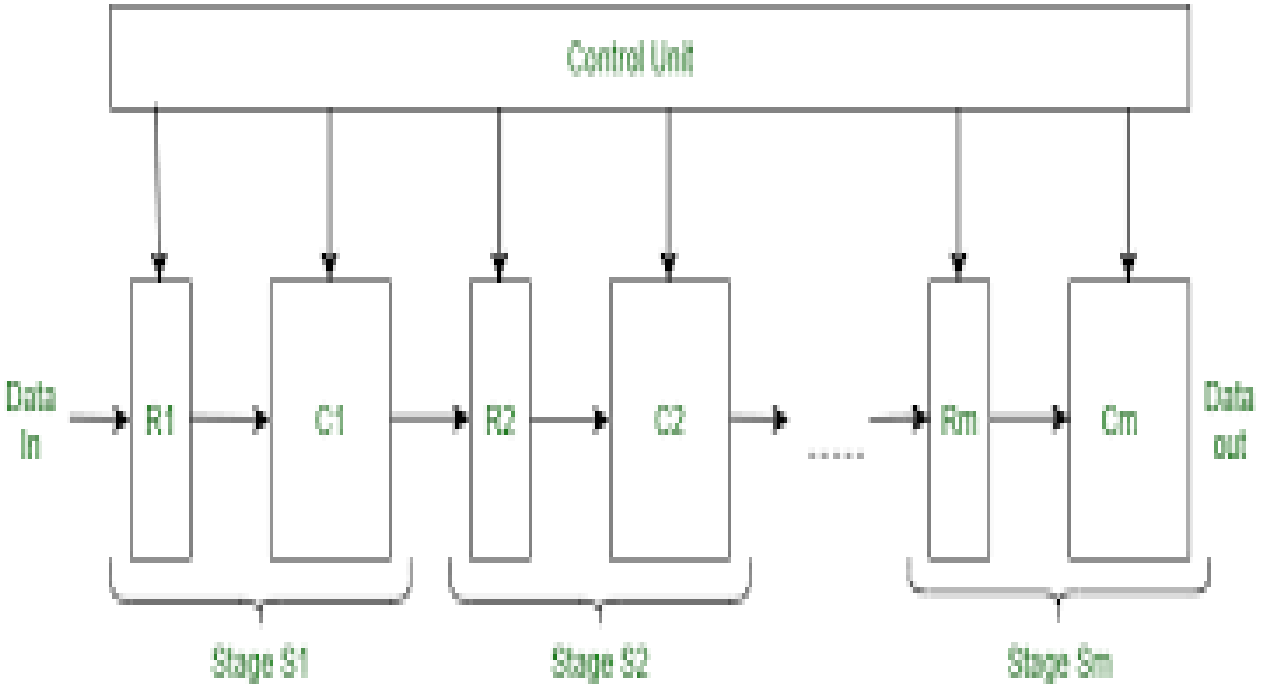
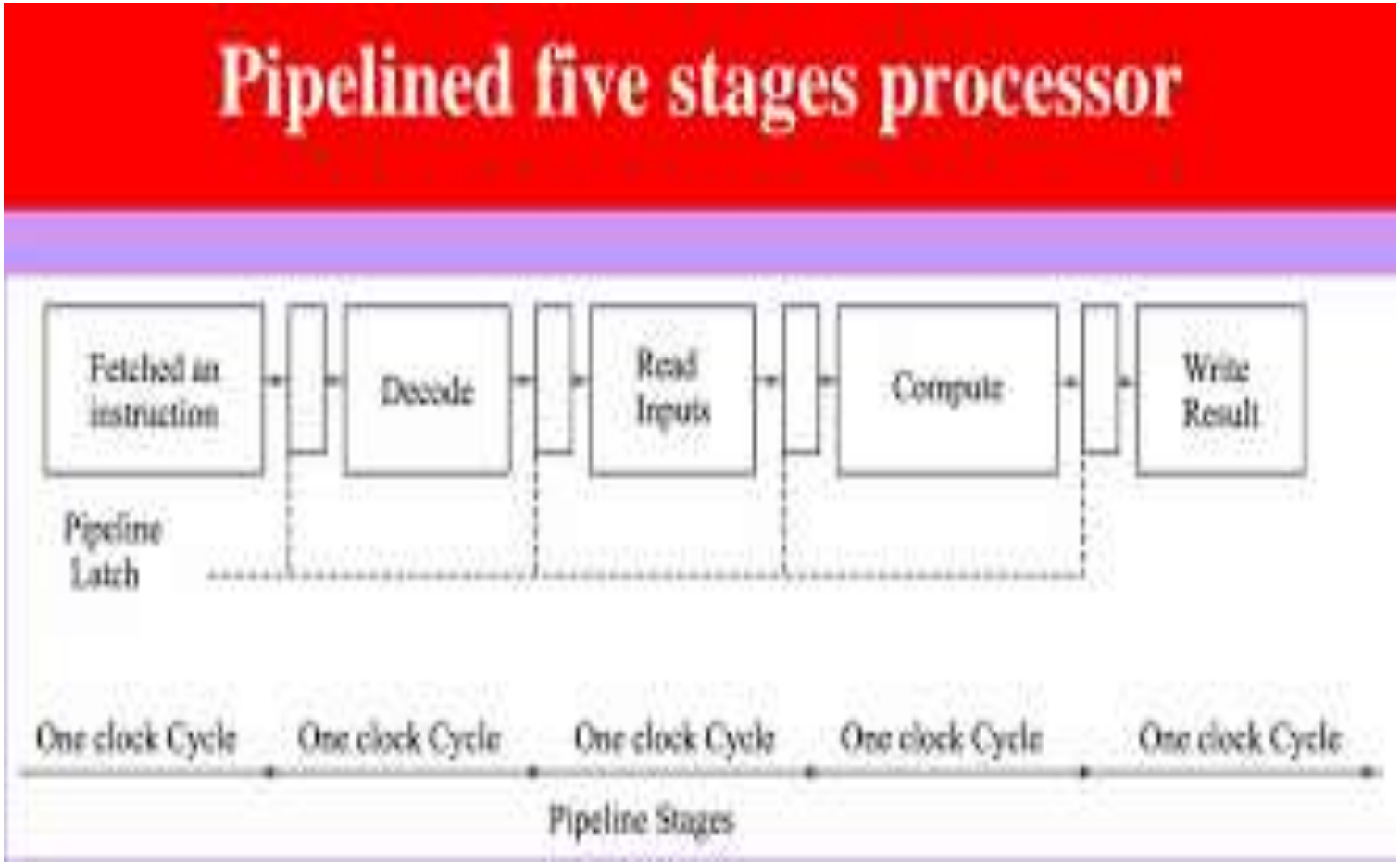


Figure - Structure of a Pipeline Processor

PIPELINING(Contd)

- Pipeline is divided into five stages.
- Each stage completes a part of an instruction in parallel. The stages are connected one to the next to form a pipe like structure. Instructions enter at one end, progress through the stages, and exit at the other end.

PIPELINING(Contd)



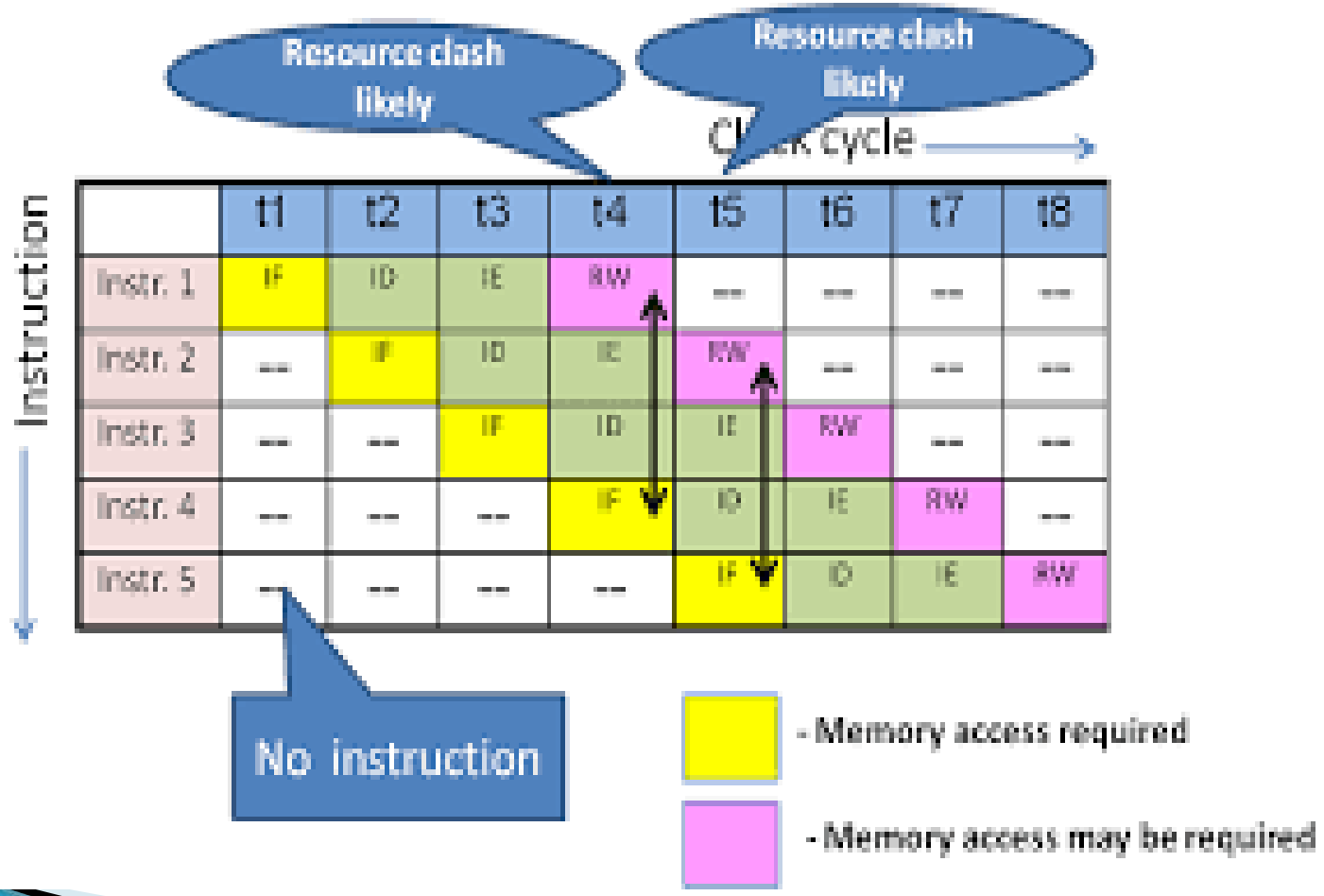
PIPELINING(Contd)

- Stages of a pipeline
- MIPS pipeline classically take the following five steps:
- Fetch instruction from memory (IF)
- Read registers while decoding the instruction (ID)
- In MIPS implementation reading and decoding occur simultaneously.
- Execute the operation or calculate an address(EX)
- Access an operand in data memory(MEM)
- Write back the result into a register(WB)

PIPELINE HAZARDS

- Any condition that causes the pipeline to stall is called a hazard. It prevents the next instruction in the instruction stream from being executing during its designated clock cycle. These events are called hazards

PIPELINE HAZARDS (Contd)



PIPELINE HAZARDS (Contd)

- Types:
- Data hazard
- Control/ Instruction hazard

DATA HAZARDS

- It occurs when the data are not available at the time expected in the pipeline. It is also called pipeline data hazard.

DATA HAZARDS(Contd)

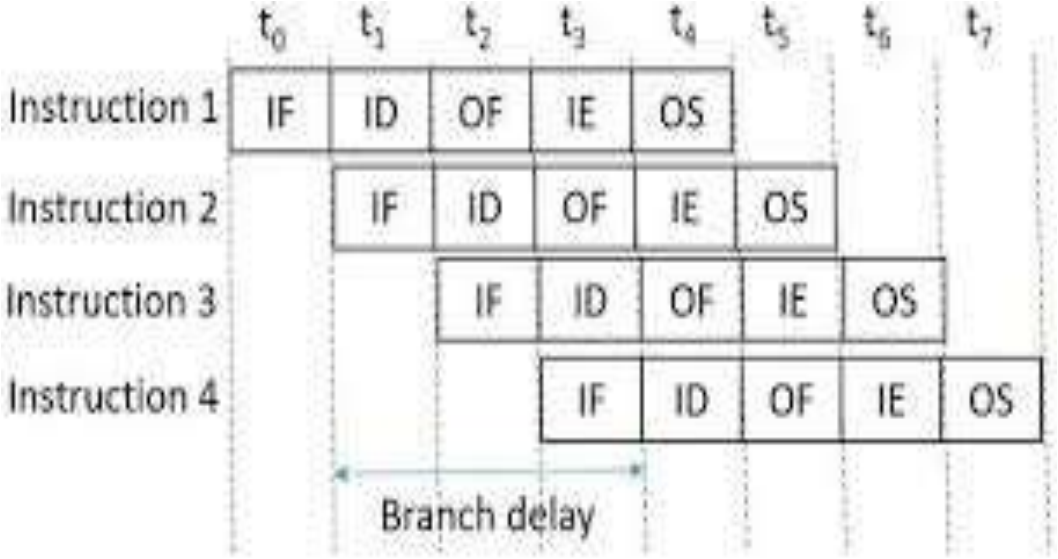
- It is an occurrence in which a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

DATA HAZARDS(Contd)

CONTROL HAZARDS

- It is also called branch hazard or instruction hazard.
- It occurs when the branching decisions are made before branch condition is evaluated.
- It is an occurrence in which the proper instruction cannot execute in the proper clock cycle because the instruction that was fetched is not the one that is needed.
- The flow of instruction addresses is not what the pipeline expected.

CONTROL HAZARDS(Contd)



Control Hazards

THANK YOU

Kongunadu College of Engineering And Technology

(Autonomous)

Namakkal – Trichy Main Road, Thottiam

Department of Computer Science and Engineering

24EC304-Digital Logic and Computer Organization

Presented By

Ms.SUGANYA S

UNIT V
MEMORY AND I/O

Memory Concepts and Hierarchy – Memory Management – Cache Memories: Mapping and Replacement Techniques – Virtual Memory – DMA – I/O – Accessing I/O: Parallel and Serial Interface – Interrupt I/O – Interconnection Standards: USB, SATA.

MEMORY CONCEPTS AND HIERARCHY

- Memory in computer systems is organized in a hierarchy to optimize speed, cost, and storage capacity.
- At the top are registers, the fastest and smallest memory units located within the CPU.
- Just below are cache memories, which store frequently accessed data and are faster than main memory but slower than registers.

MEMORY CONCEPTS AND HIERARCHY(Contd)

- Main memory (RAM) holds currently running programs and data and has a larger capacity than cache. At the bottom is secondary storage, such as hard drives and SSDs, which provide large, permanent storage but operate much more slowly. This hierarchical arrangement ensures efficient and cost-effective data access.

MEMORY CONCEPTS AND HIERARCHY(Contd)

Types of Memory Hierarchy

- External Memory or Secondary Memory:
- Comprising of Magnetic Disk, Optical Disk, and Magnetic Tape i.e. peripheral storage devices which are accessible by the processor via an I/O Module.
- Internal Memory or Primary Memory:
- Comprising of Main Memory, Cache Memory & CPU registers. This is directly accessible by the processor.

Memory Hierarchy Levels

- Registers
- Registers are small, high-speed memory units located in the CPU. They are used to store the most frequently used data and instructions.
- Cache Memory
- Cache memory is a small, fast memory unit located close to the CPU. It stores frequently used data and instructions that have been recently accessed from the main memory.

Memory Hierarchy Levels(Contd)

- Main Memory
- Main memory also known as RAM (Random Access Memory), is the primary memory of a computer system. It has a larger storage capacity than cache memory, but it is slower. Main memory is used to store data and instructions that are currently in use by the CPU.

Characteristics of Memory Hierarchy

- Access Time:
  Registers have the fastest access time, while secondary storage has the slowest.
- Capacity:
  Secondary storage has the largest capacity, while registers have the smallest.
- Cost:
  Registers are the most expensive per bit, while secondary storage is the least expensive.
- Locality of Reference:
  The principle that programs tend to access the same data and instructions repeatedly, allowing caches to store frequently used information.

Advantages of Memory Hierarchy

- Performance
- Cost Efficiency
- Optimized Resource Utilization
- Efficient Data Management

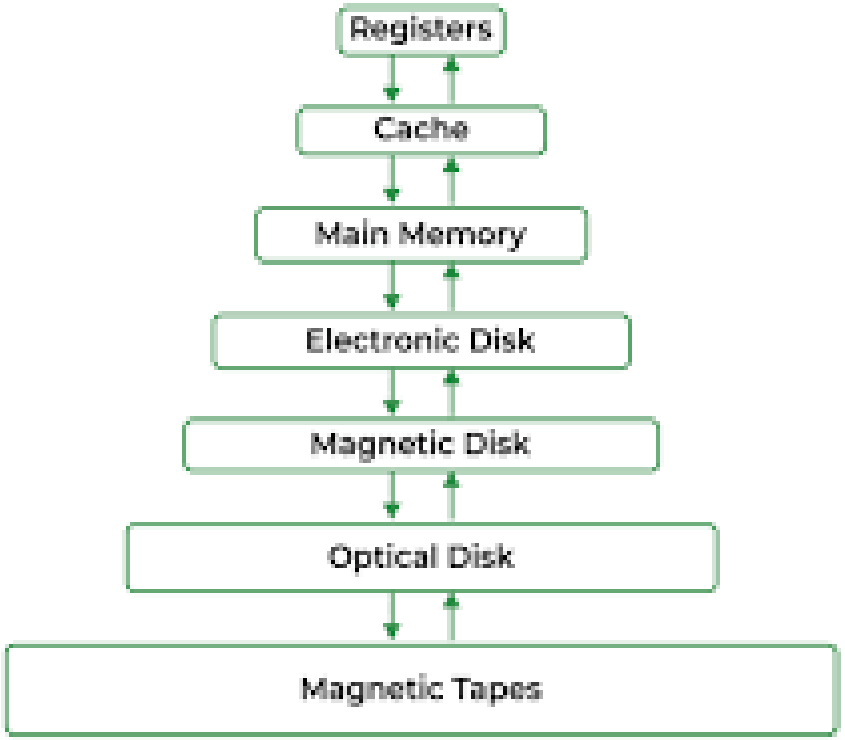
Disadvantages of Memory Hierarchy

- Complex Design
- Cost
- Latency
- Maintenance Overhead

MEMORY MANAGEMENT

- Memory is a crucial part of a computer used to store data.
- Since main memory is limited and multiple processes compete for it, efficient memory management is essential especially in multiprogramming systems.
- To enhance performance, several processes must reside in memory simultaneously.
- Poor memory allocation can leave the CPU idle while processes wait for I/O.
- Hence, memory must be managed efficiently to maximize CPU utilization and system throughput.

MEMORY MANAGEMENT(Contd)



MEMORY MANAGEMENT(Contd)

- Memory management mostly involves management of main memory. In a multiprogramming computer, the Operating System resides in a part of the main memory, and the rest is used by multiple processes.
- The task of subdividing the memory among different processes is called Memory Management.
- The main aim of memory management is to achieve efficient utilization of memory.

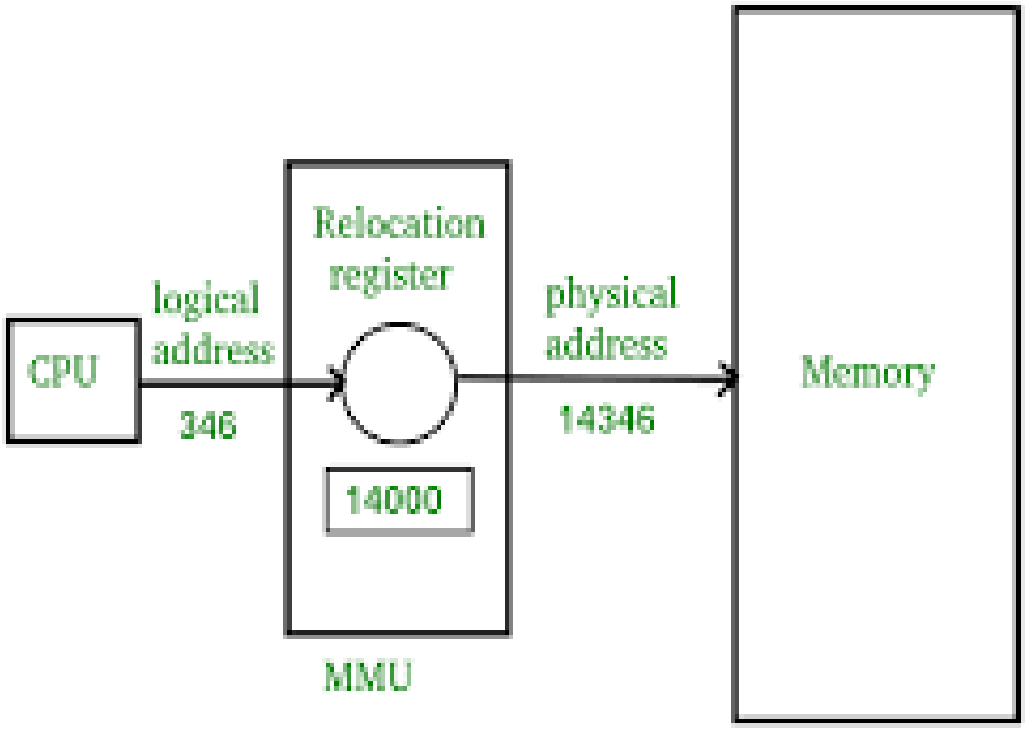
Why Memory Management is Required?

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process

Logical and Physical Address Space

- Logical Address Space: An address generated by the CPU is known as a “Logical Address”. It is also known as a Virtual address. Logical address space can be defined as the size of the process. A logical address can be changed.
- • Physical Address Space: An address seen by the memory unit (i.e. the one loaded into the memory address register of the memory) is commonly known as a “Physical Address”. A Physical address is also known as a Real address.

Logical and Physical Address Space



Static and Dynamic Loading

- Static Loading: Static Loading is basically loading the entire program into a fixed address. It requires more memory space.
- Dynamic Loading: The entire program and all data of a process must be in physical memory for the process to execute. So, the size of a process is limited to the size of physical memory.

Static and Dynamic Linking

- Static Linking: In static linking, the linker combines all necessary program modules into a single executable program. So there is no runtime dependency.

- Dynamic Linking: The basic concept of dynamic linking is similar to dynamic loading. In dynamic linking, “Stub” is included for each appropriate library routine reference.

Static and Dynamic Linking

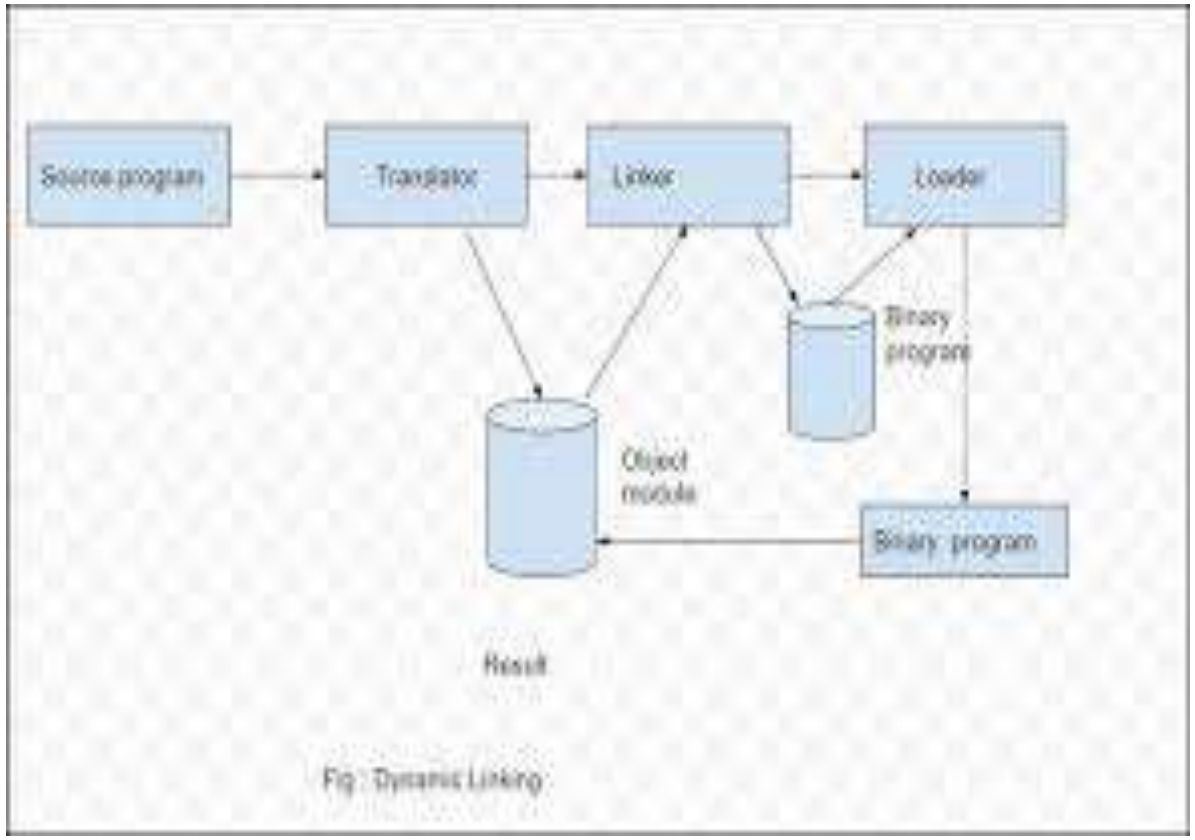
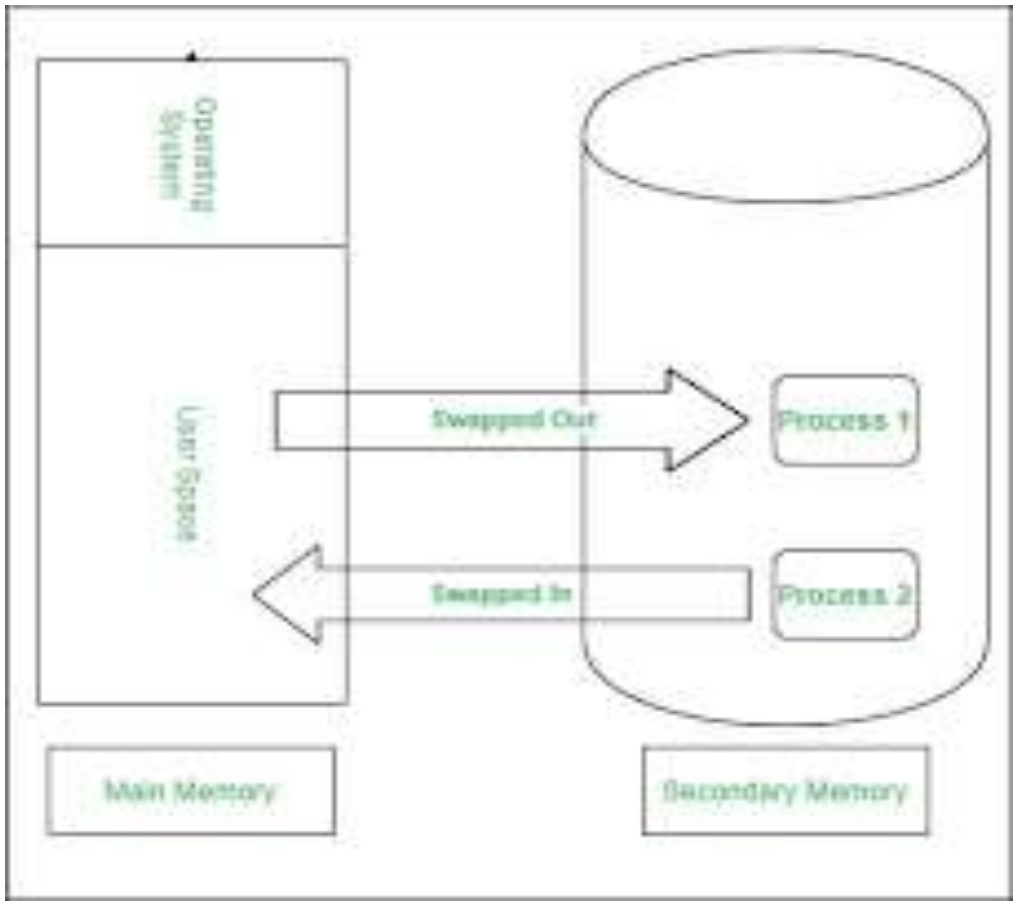


Fig : Dynamic Linking

Swapping

- When a process is executed it must have resided in memory.
- Swapping is a process of swapping a process temporarily into a secondary memory from the main memory, which is fast compared to secondary memory.
- A swapping allows more processes to be run and can be fit into memory at one time.
- The main part of swapping is transferred time and the total time is directly proportional to the amount of memory swapped.

Swapping(Contd)

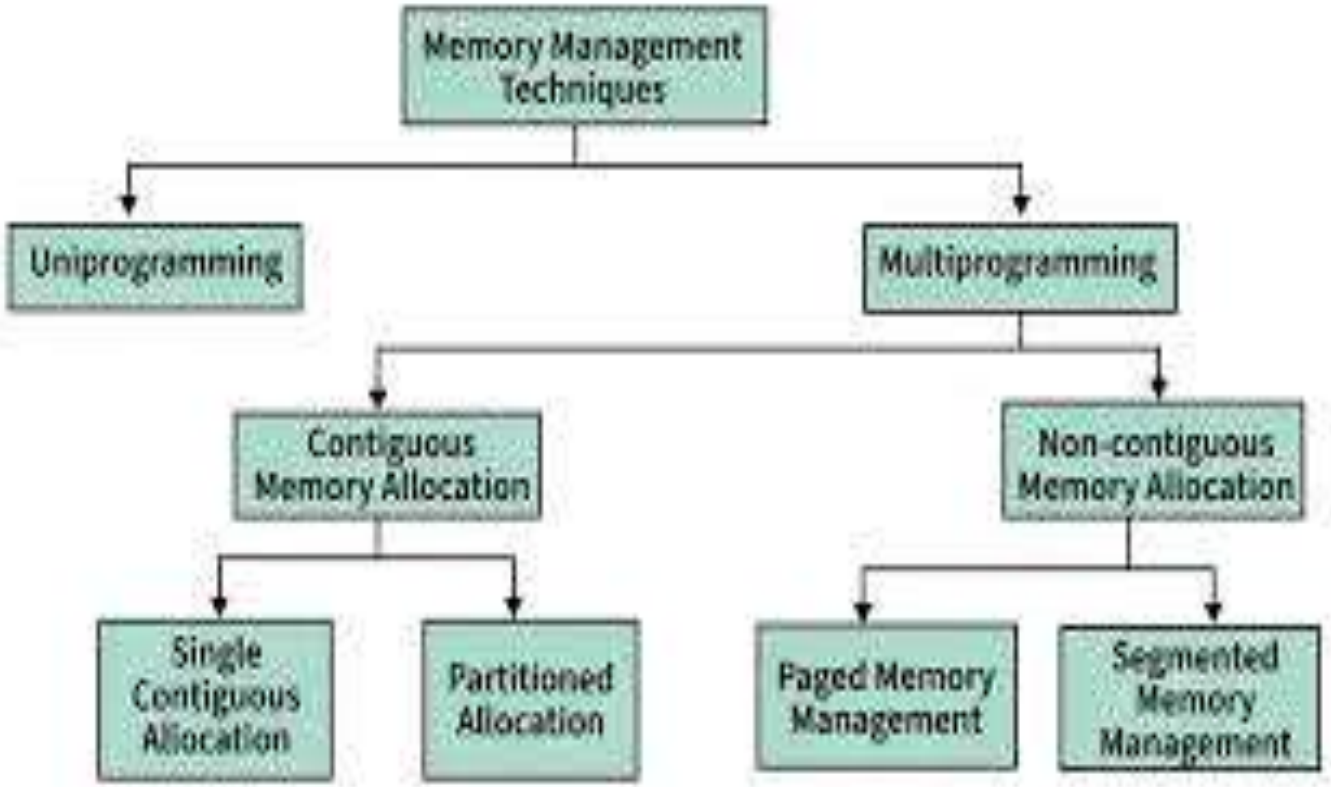


Memory Management Techniques

- Memory management techniques are methods used by an operating system to efficiently allocate, utilize, and manage memory resources for processes.

- These techniques ensure smooth execution of programs and optimal use of system memory

Memory Management Techniques (Contd)



Memory Management Techniques(Contd)

- Memory Management with Monoprogramming (Without Swapping)
- This is the simplest memory management approach the memory is divided into two sections:
- One part of the operating system
- The second part of the user program

Memory Management Techniques(Contd)

- Multiprogramming with Fixed Partitions (Without Swapping)
- • A memory partition scheme with a fixed number of partitions was introduced to support multiprogramming. this scheme is based on contiguous allocation
- • Each partition is a block of contiguous memory
- • Memory is partitioned into a fixed number of partitions.
- • Each partition is of fixed size

Memory Management Techniques(Contd)

- Partition Table

  Once partitions are defined operating system keeps track of the status of memory partitions it is done through a data structure called a partition table.

Memory Management Techniques(Contd)

- Logical vs Physical Address
- An address generated by the CPU is commonly referred to as a logical address. the address seen by the memory unit is known as the physical address. The logical address can be mapped to a physical address by hardware with the help of a base register this is known as dynamic relocation of memory references.

Memory Management Techniques(Contd)

- Contiguous Memory Allocation
- Contiguous memory allocation is a memory management method where each process is given a single, continuous block of memory. This means all the data for a process is stored in adjacent memory locations.

Memory Management Techniques(Contd)

- Partition Allocation Methods
- To gain proper memory utilization, memory allocation must be allocated efficient manner. One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions and each partition contains exactly one process.

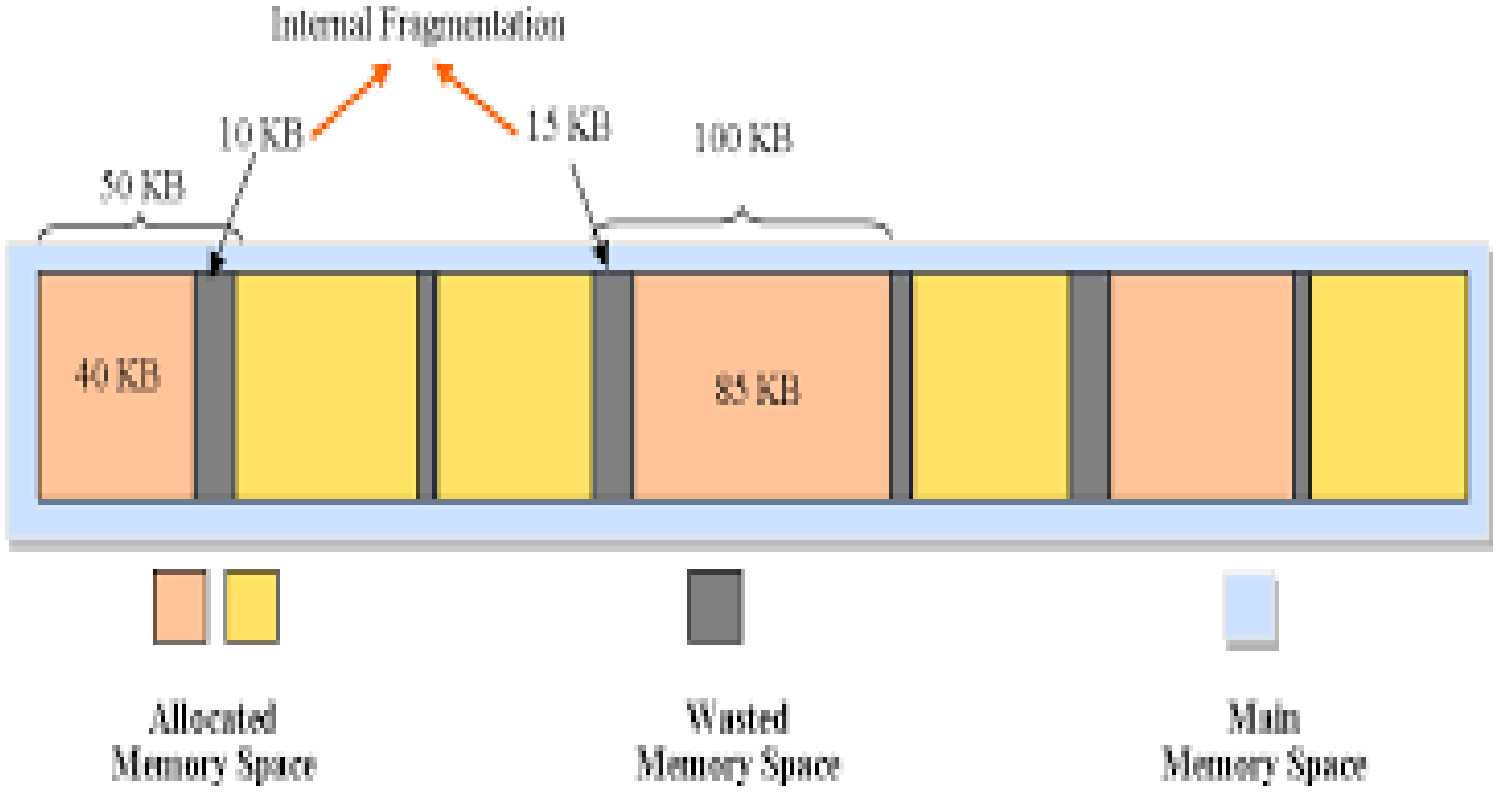
Memory Management Techniques(Contd)

- Non-Contiguous Memory Allocation
- Non-contiguous memory allocation is a memory management method where a process is divided into smaller parts, and these parts are stored in different, non-adjacent memory locations.
- Techniques of Non-Contiguous Memory Allocation are:
- Paging
- Segmentation

Memory Management Techniques(Contd)

- Fragmentation
- Fragmentation is defined as when the process is loaded and removed after execution from memory, it creates a small free hole. These holes cannot be assigned to new processes because holes are not combined or do not fulfil the memory requirement of the process.
-

Memory Management Techniques (Contd) – Fragmentation



Memory Management Techniques(Contd)

- Types of fragmentation are:
- Internal fragmentation: Internal fragmentation occurs when memory blocks are allocated to the process more than their requested size. Due to this some unused space is left over and creating an internal fragmentation problem.

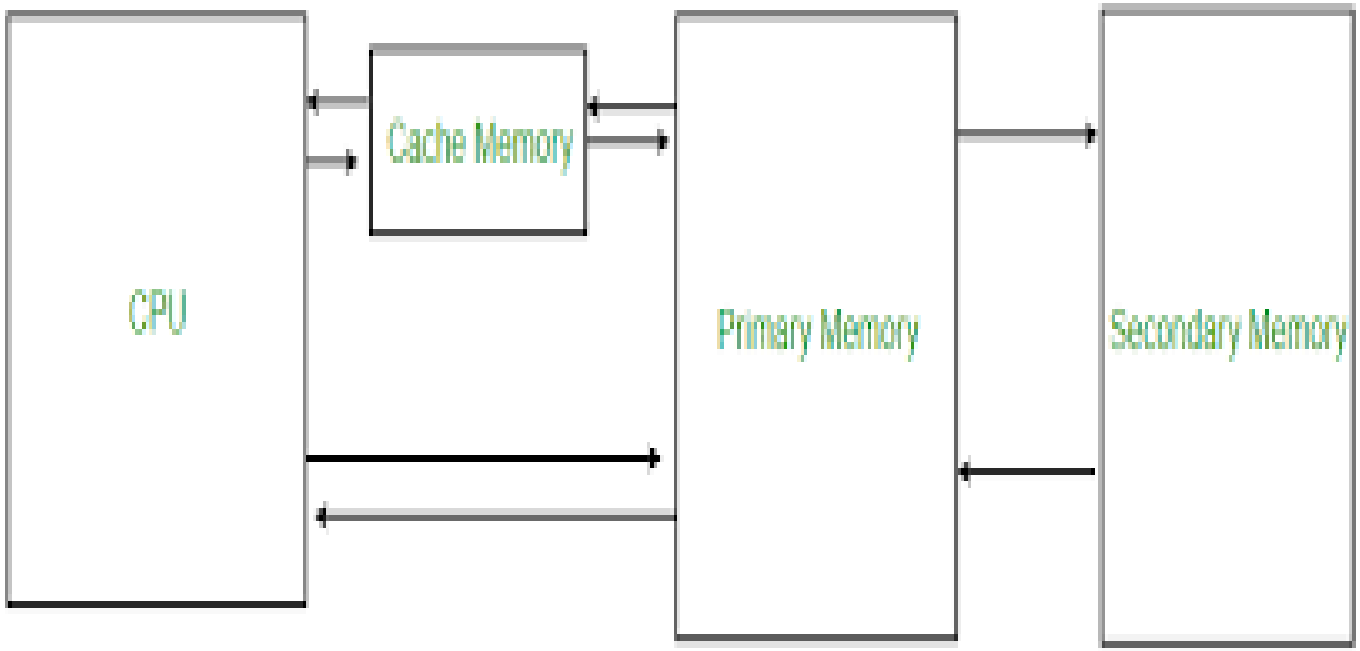
Memory Management Techniques(Contd)

- Types of fragmentation(Contd):
- External fragmentation: In External Fragmentation, we have a free memory block, but we cannot assign it to a process because blocks are not contiguous.

CACHE MEMORIES

- Cache memory is a small, high-speed storage area in a computer. The cache is a smaller and faster memory that stores copies of the data from frequently used main memory locations.
- There are various independent caches in a CPU, which store instructions and data.

CACHE MEMORIES (contd)



CACHE MEMORIES(Contd)

Characteristics of Cache Memory

- Extremely fast memory type that acts as a buffer between RAM and the CPU.
- Holds frequently requested data and instructions, ensuring that they are immediately available to the CPU when needed.
- Costlier than main memory or disk memory but more economical than CPU registers.
- Used to speed up processing and synchronize with the high-speed CPU.

CACHE MEMORIES(Contd)

- Key Features of Cache Memory
- Speed: Faster than the main memory (RAM), which helps the CPU retrieve data more quickly.
- Proximity: Located very close to the CPU, often on the CPU chip itself, reducing data access time.
- Function: Temporarily holds data and instructions that the CPU is likely to use again soon, minimizing the need to access the slower main memory.

CACHE MEMORIES(Contd)

- The role of cache memory is explained below,
- • Cache memory plays a crucial role in computer systems.
- • It provides faster access.
- • It acts buffer between CPU and main memory(RAM).
- • Primary role of it is to reduce average time taken to access data, thereby improving. overall system performance.

CACHE MEMORIES(Contd)

- In order to understand the working of cache we must understand few points:
- Cache memory is faster, they can be accessed very fast
- Cache memory is smaller, a large amount of data cannot be stored

CACHE MEMORIES(Contd)

- Application of Cache Memory
- • Primary Cache
- • Secondary Cache
- • Spatial Locality of Reference
- • Temporal Locality of Reference

CACHE MEMORIES(Contd)

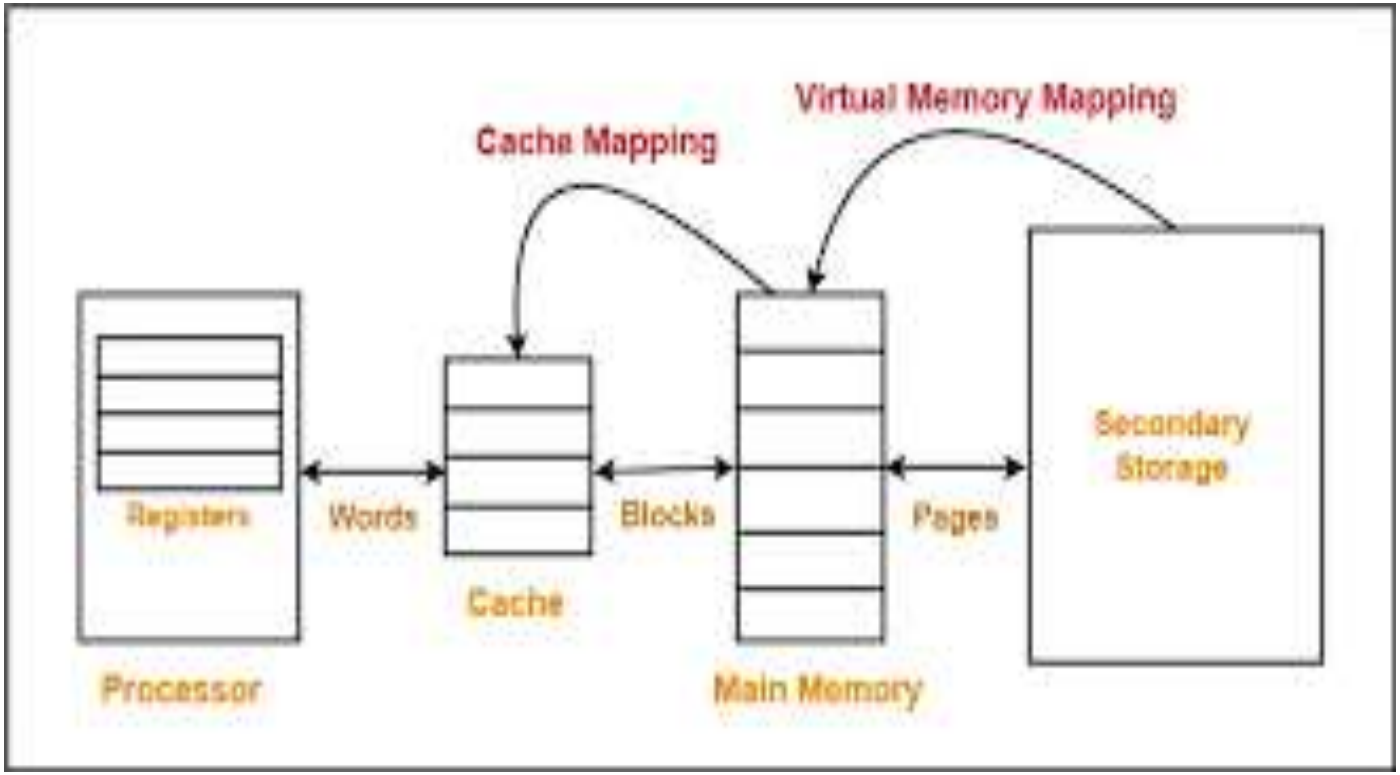
- Advantages
- Cache Memory is faster in comparison to main memory and secondary memory.
- Programs stored by Cache Memory can be executed in less time.
- The data access time of Cache Memory is less than that of the main memory.
- Cache Memory stored data and instructions that are regularly used by the CPU, therefore it increases the performance of the CPU.

MAPPING AND REPLACEMENT TECHNIQUES

- Cache Mapping
- Cache mapping refers to the method used to store data from main memory into the cache. It determines how data from memory is mapped to specific locations in the cache.

- Need of Replacement Algorithm:
- Set associative mapping is a combination of direct mapping and fully associative mapping.
- It uses fully associative mapping within each set.
- Thus, set associative mapping requires a replacement algorithm.

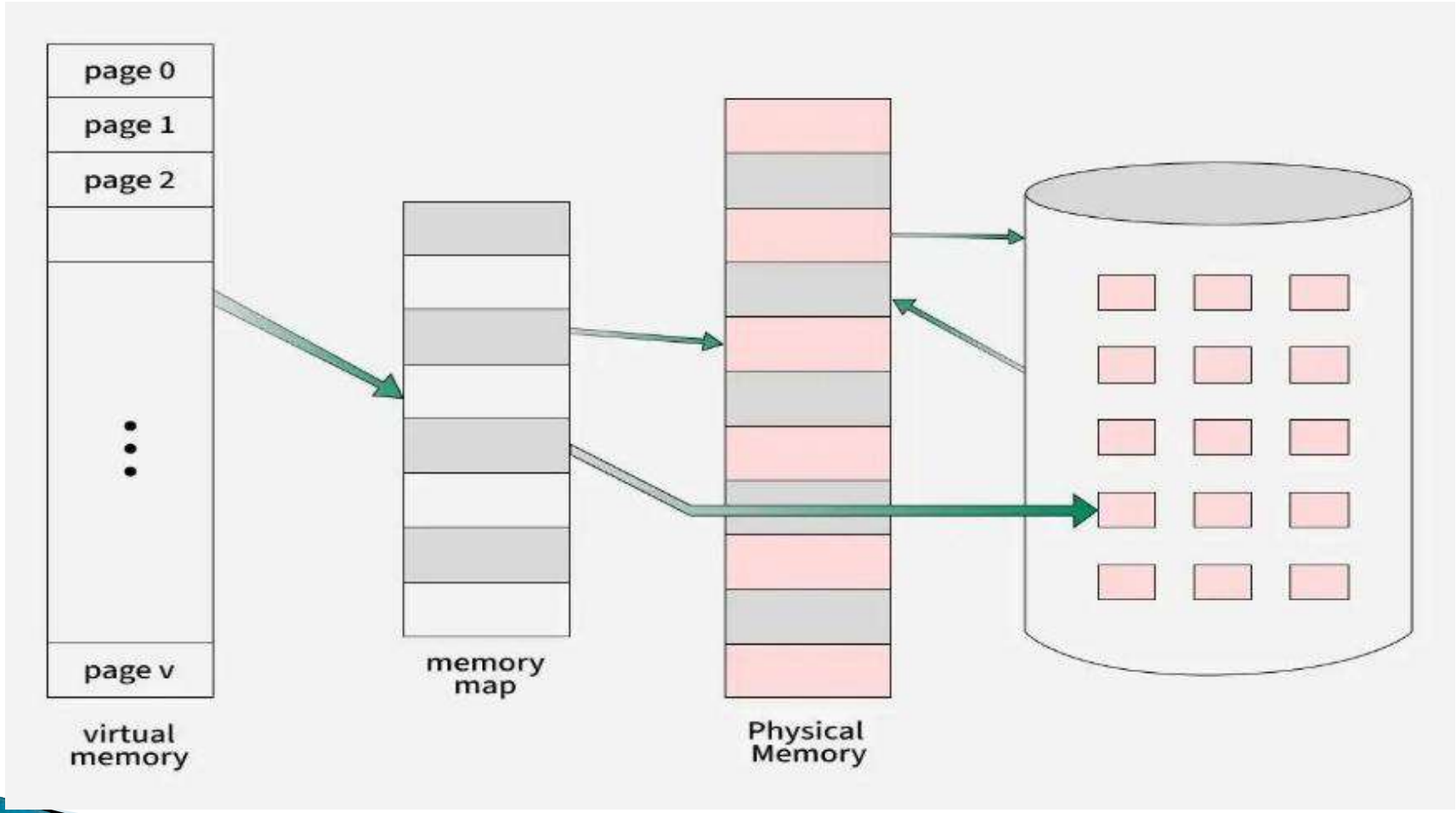
MAPPING AND REPLACEMENT TECHNIQUES (Contd)



VIRTUAL MEMORY

- Virtual memory is a memory management technique used by operating systems to give the appearance of a large, continuous block of memory to applications, even if the physical memory (RAM) is limited. It allows larger applications to run on systems with less RAM.

VIRTUAL MEMORY(Contd)



VIRTUAL MEMORY(Contd)

- How Virtual Memory Works?
- All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. 126
- This means that a process can be swapped in and out of the main memory such that it occupies different places in the main memory at different times during the course of execution.
- A process may be broken into a number of pieces and these pieces need not be continuously located in the main memory during execution.
- The combination of dynamic run-time address translation and the use of a page or segment table permit this.

VIRTUAL MEMORY(Contd)

- Types of Virtual Memory
- In a computer, virtual memory is managed by the Memory Management Unit (MMU), which is often built into the CPU. The CPU generates virtual addresses that the MMU translates into physical addresses.
- There are two main types of virtual memory:
- Paging
- Segmentation

VIRTUAL MEMORY(Contd)

- Paging divides memory into small fixed-size blocks called pages. When the computer runs out of RAM, pages that aren't currently in use are moved to the hard drive, into an area called a swap file.
- The swap file acts as an extension of RAM. When a page is needed again, it is swapped back into RAM, a process known as page swapping.
- This ensures that the operating system (OS) and applications have enough memory to run.
- Demand Paging: The process of loading the page into memory on demand (whenever a page fault occurs) is known as demand paging.

VIRTUAL MEMORY(Contd)

- Segmentation
- Segmentation divides virtual memory into segments of different sizes. Segments that aren't currently needed can be moved to the hard drive.
- The system uses a segment table to keep track of each segment's status, including whether it's in memory, if it's been modified, and its physical address.
- Segments are mapped into a process's address space only when needed.

VIRTUAL MEMORY(Contd)

- Swapping is a process out means removing all of its pages from memory, or marking them so that they will be removed by the normal page replacement process.
- Suspending a process ensures that it is not run able while it is swapped out.
- At some later time, the system swaps back the process from the secondary storage to the main memory.
- When a process is busy swapping pages in and out then this situation is called thrashing.

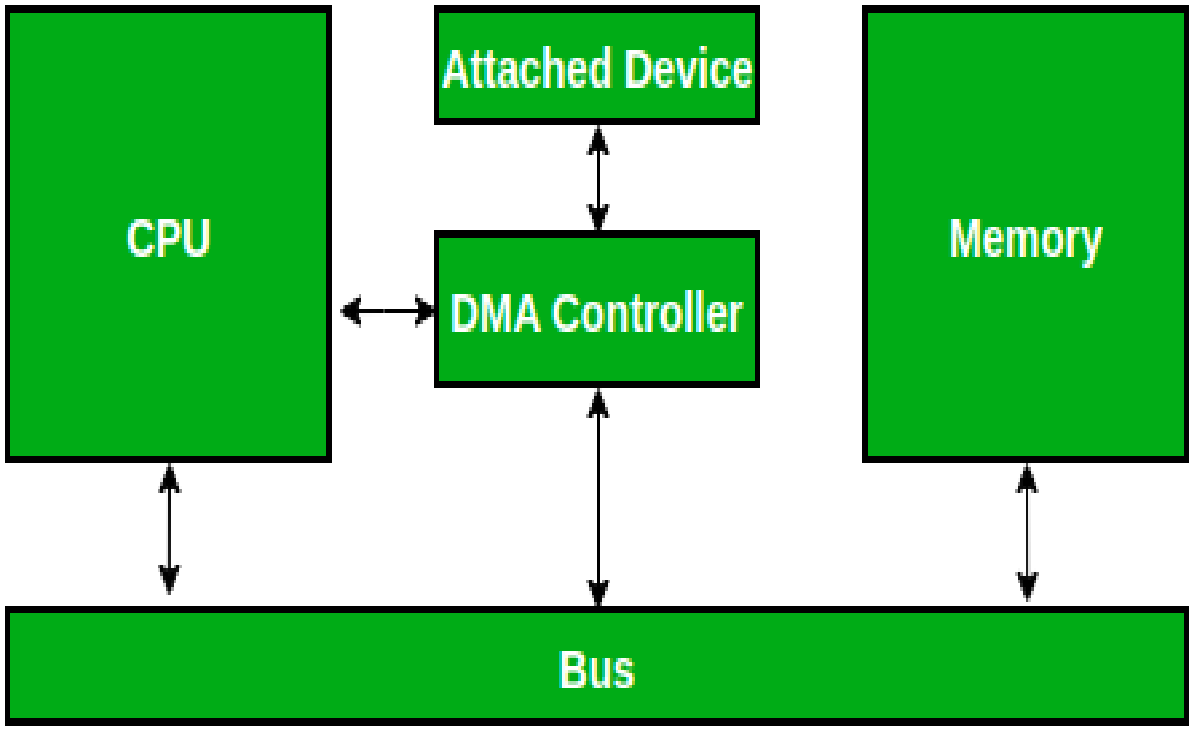
VIRTUAL MEMORY(Contd)

- Thrashing:
- At any given time, only a few pages of any process are in the main memory, and therefore more processes can be maintained in memory. Furthermore, time is saved because unused pages are not swapped in and out of memory.

DMA- Direct Memory Access

- In modern computer systems, transferring data between input/output devices and memory can be a slow process if the CPU is required to manage every step. To address this, a Direct Memory Access (DMA) Controller is utilized.

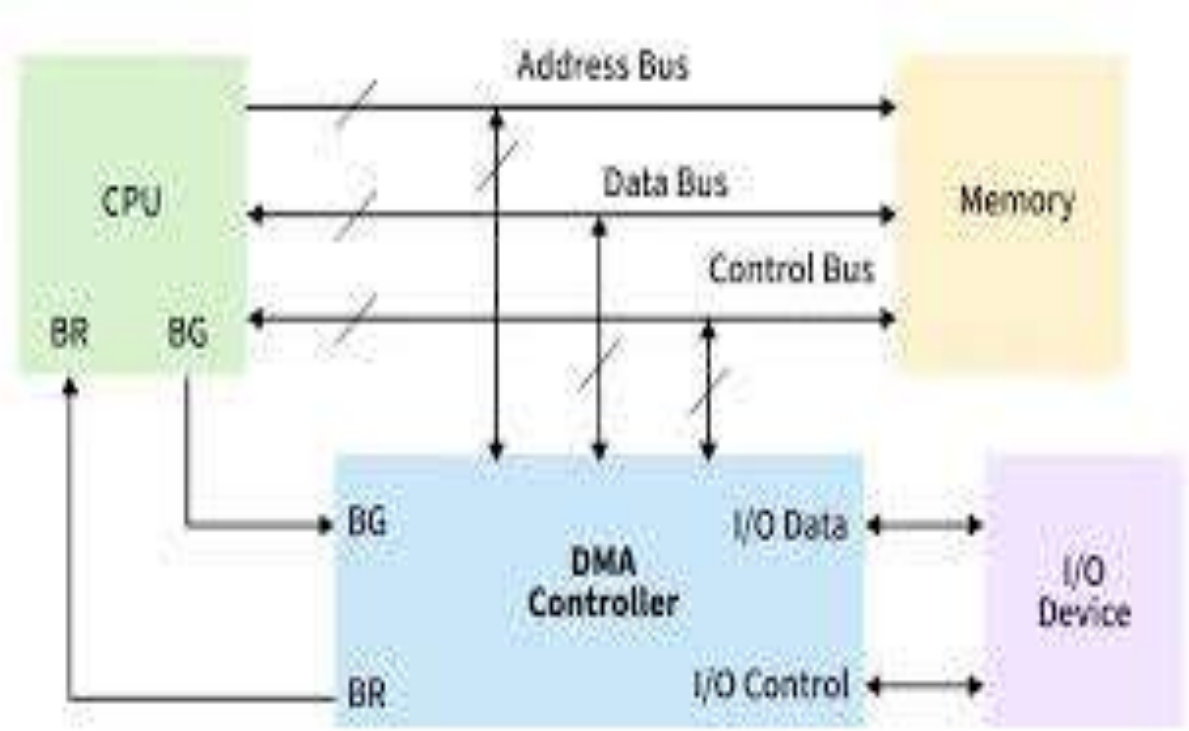
DMA(Contd)



DMA(Contd)

- DMA Controller
- Direct Memory Access (DMA) uses hardware for accessing the memory.
- This hardware is called a DMA Controller. It has the work of transferring the data between input, output devices and main memory with very less interaction with the processor.
- The Direct Memory Access Controller is a control unit, which has the work of transferring data.

DMA Controller



DMA(Contd)

- Types of DMA
- Single-Ended DMA: In this type, the DMA controller is connected only to one device (usually either the memory or the I/O device), and it directly controls data transfer.
- Dual-Ended DMA: The DMA controller is connected to both the source and the destination, typically memory and an I/O device.

DMA(Contd)

- Types of DMA
- Arbitrated-Ended DMA: In systems with multiple DMA devices or masters, arbitration is needed to decide which device gets control of the bus. It is more advanced than Dual-Ended DMA.
- Interleaved DMA: Interleaved DMA are those DMA that read from one memory address and write from another memory address.

DMA(Contd)

- Working of DMA Controller
- The DMA controller registers have three registers as follows.
- Address register: It contains the address to specify the desired location in memory.
- Word count register: It contains the number of words to be transferred.
- Control register: It specifies the transfer mode.

I/O – ACCESSING I/O: PARALLEL AND SERIAL INTERFACE

- The method that is used to transfer information between internal storage and external I/O devices is known as I/O interface.
- The CPU is interfaced using special communication links by the peripherals connected to any computer system.
- These communication links are used to resolve the differences between CPU and peripheral.
- There exists special hardware components between CPU and peripherals to supervise and synchronize all the input and output transfers that are called interface units.

I/O – ACCESSING I/O: PARALLEL AND SERIAL INTERFACE

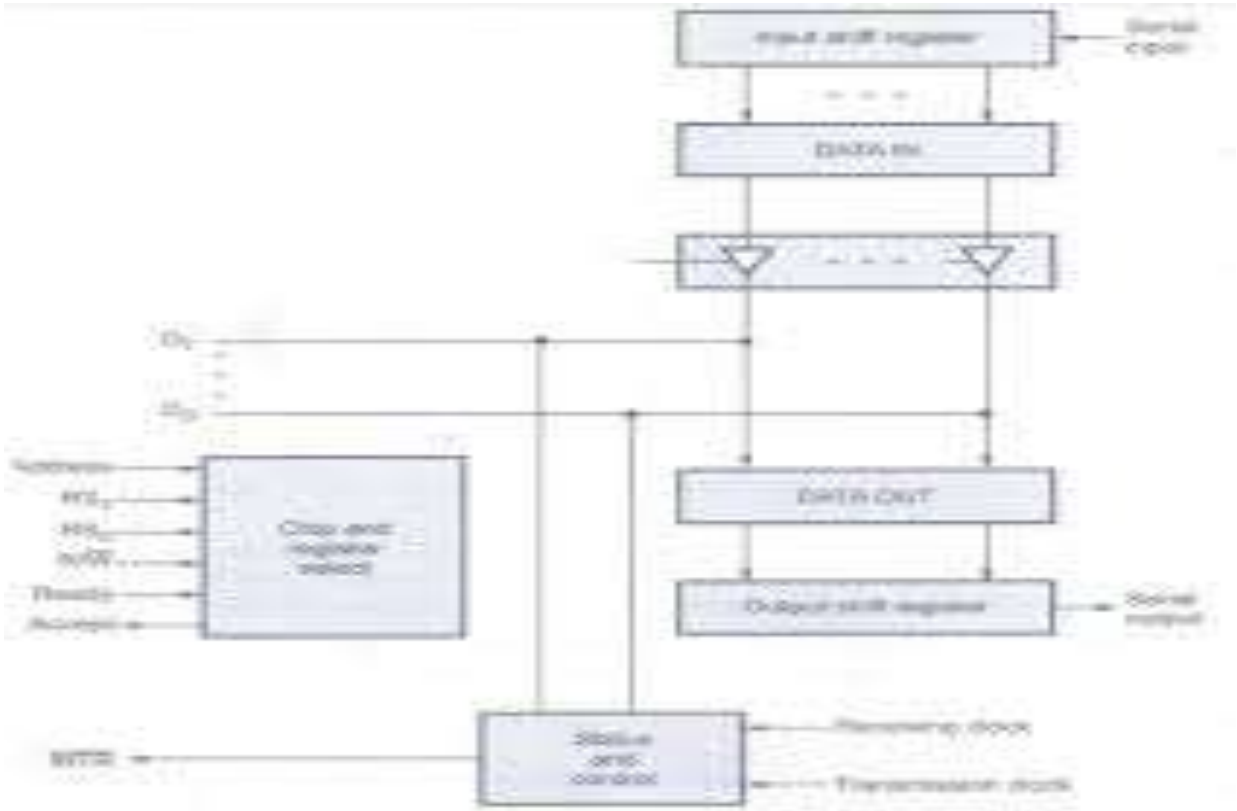
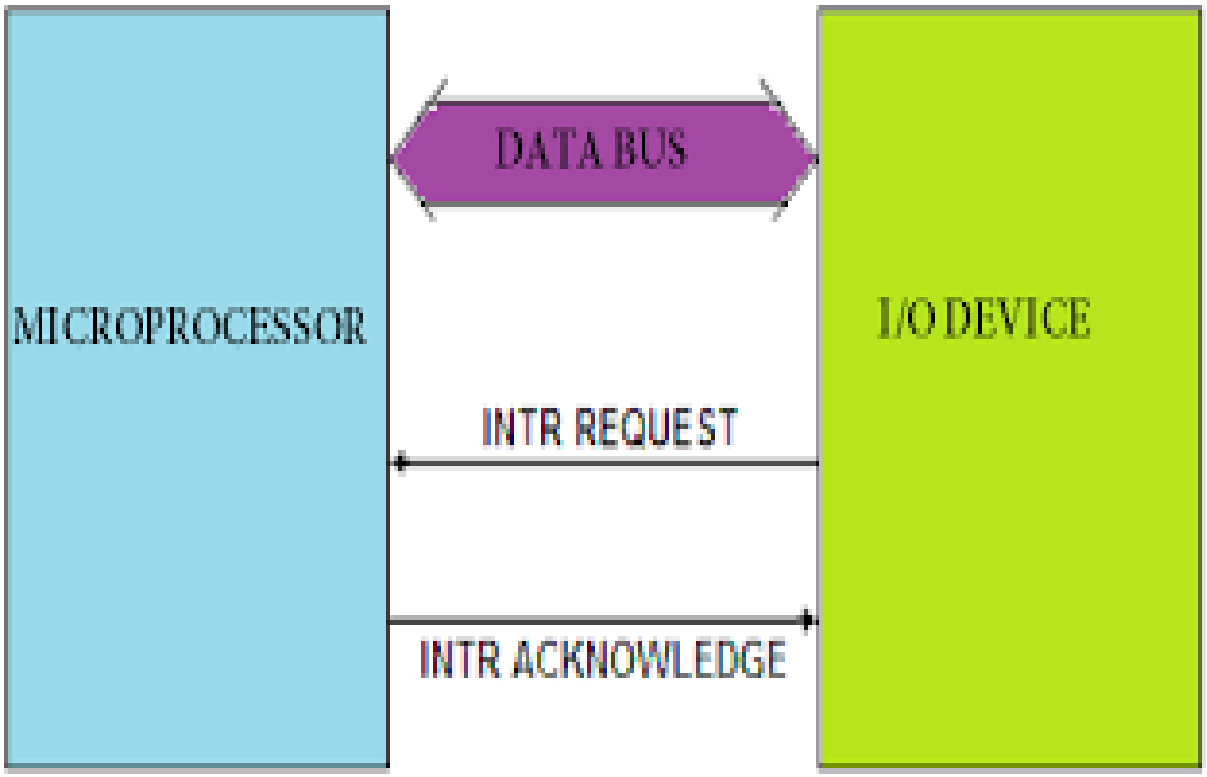


Fig. 8.6 A serial interface

INTERRUPT I/O

- An interrupt I/O is a process of data transfer in which an external device or a peripheral informs the CPU that it is ready for communication and requests the attention of the CPU.The terminals send and receive serial information.

INTERRUPT I/O (Contd)



INTERCONNECTION STANDARDS: USB

- USB was designed to standardize the connection of peripherals like pointing devices, keyboards, digital images and video cameras.
- But some devices such as printers, portable media players, disk drives, and network adaptors to personal computers used USB to communicate and to supply electric power.

USB(Contd)

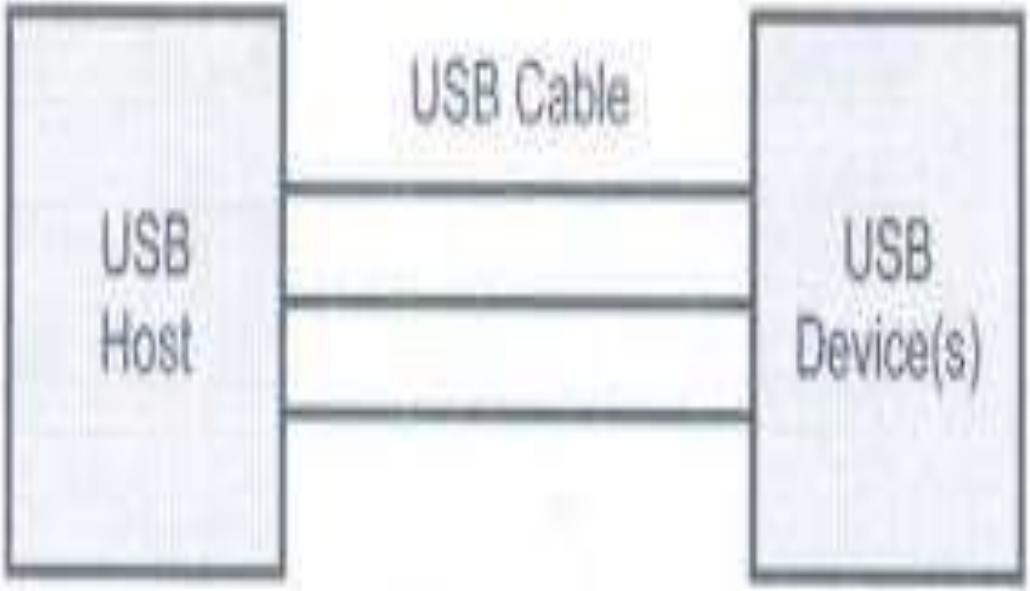


Fig. 8.11.1 USB system

USB(Contd)

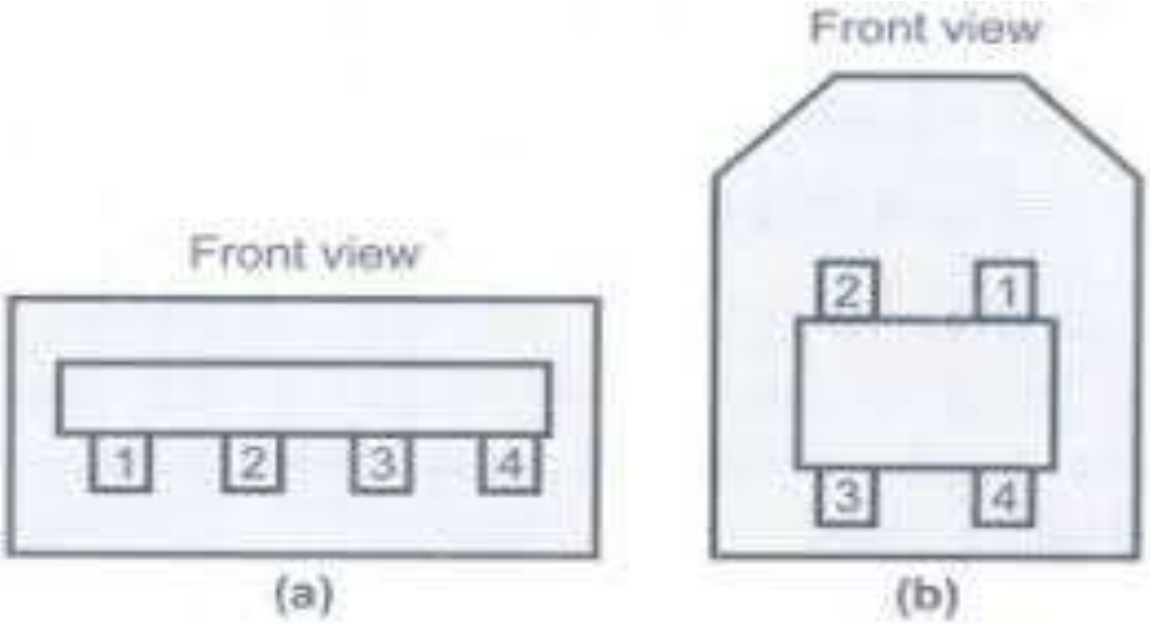


Fig. 8.11.2 USB connector

INTERCONNECTION STANDARDS: USB(Contd)

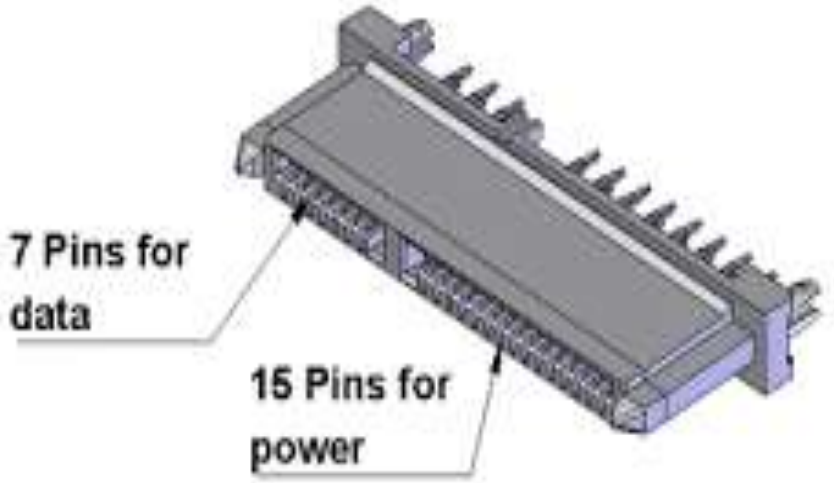
- Universal Serial Bus (USB) is an industry standard that establishes specifications for connectors, cables, and protocols for communication, connection, and power supply between personal computers and their peripheral devices.
- There have been 3 generations of USB specifications:
- USB 1.x
- USB 2.0
- USB 3.x

USB(Contd)

Serial Advanced Technology Attachment (SATA)

- SATA stands for Serial Advanced Technology Attachment or Serial ATA. SATA is an interface that connects various storage devices such as hard disks, optical drives SSD's, etc to the motherboard.

SATA (Contd)



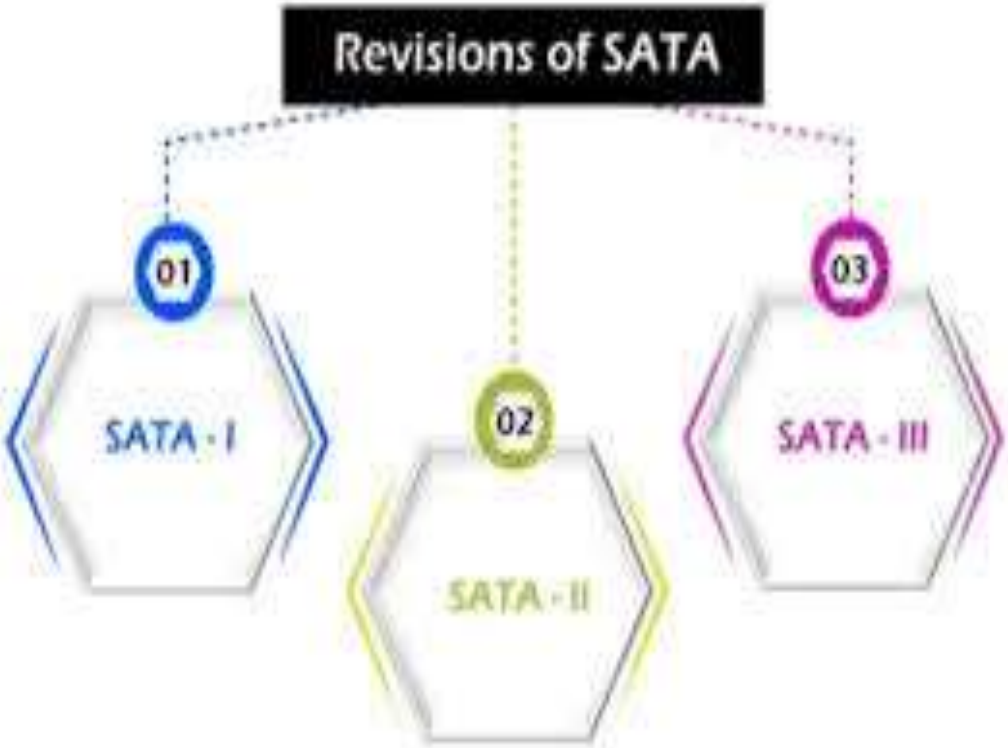
Serial Advanced Technology Attachment (SATA)(Contd)

- SATA operates on two modes:
- 1. IDE mode: IDE stands for Integrated Drive Electronics. This is a mode which is used to provide backward compatibility with older hardware, which runs on PATA, at the cost of low performance.
- 2. AHCI mode: AHCI is abbreviation for Advanced Host Controller Interface. AHCI is a high-performance mode that also provides support for hot-swapping.

SATA (Contd)

- Characteristics of SATA
- Low Voltage Requirement: SATA operates on 500mV (0.5V) peak-to-peak signaling.
- Differential Signaling: SATA uses differential signaling. Differential signaling is a technology which uses two adjacent wires to simultaneously the in-phase and out- of-phase signals.
- • High data transfer rate: SATA has a high data transfer rate of 150/300/600 MBs/second.

SATA (Contd)



SATA (Contd)

- Advantages of SATA
- Faster data transfer rate as compared to PATA.
- SATA cable can be of length upto 1 meter, whereas PATA cable can only have length of maximum 18 inches.
- SATA cables are smaller in size.
- Since, they are smaller in size; they take up less space inside the computer and increase the internal air flow.

THANK YOU