

NUMBER SYSTEMS & BOOLEAN ALGEBRA

- Introduction about digital system
- Philosophy of number systems
- Complement representation of negative numbers
- Binary arithmetic
- Binary codes
- Error detecting & error correcting codes
- Hamming codes

INTRODUCTION ABOUT DIGITAL SYSTEM

A Digital system is an interconnection of digital modules and it is a system that manipulates discrete elements of information that is represented internally in the binary form.

Now a day's digital systems are used in wide variety of industrial and consumer products such as automated industrial machinery, pocket calculators, microprocessors, digital computers, digital watches, TV games and signal processing and so on.

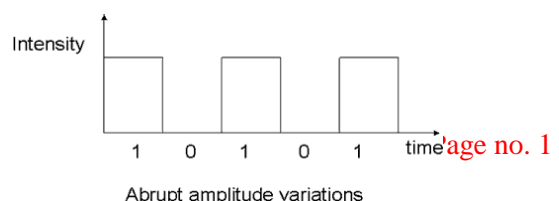
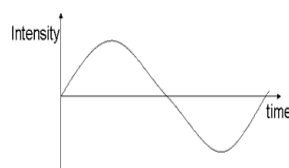
Characteristics of Digital systems

- Digital systems manipulate discrete elements of information.
- Discrete elements are nothing but the digits such as 10 decimal digits or 26 letters of alphabets and so on.
- Digital systems use physical quantities called signals to represent discrete elements.
- In digital systems, the signals have two discrete values and are therefore said to be binary.
- A signal in digital system represents one binary digit called a bit. The bit has a value either 0 or 1.

Analog systems vs Digital systems

Analog system process information that varies continuously i.e; they process time varying signals that can take on any values across a continuous range of voltage, current or any physical parameter.

Digital systems use digital circuits that can process digital signals which can take either 0 or 1 for binary system.



Advantages of Digital system over Analog system

1. Ease of programmability

The digital systems can be used for different applications by simply changing the program without additional changes in hardware.

2. Reduction in cost of hardware

The cost of hardware gets reduced by use of digital components and this has been possible due to advances in IC technology. With ICs the number of components that can be placed in a given area of Silicon are increased which helps in cost reduction.

3. High speed

Digital processing of data ensures high speed of operation which is possible due to advances in Digital Signal Processing.

4. High Reliability

Digital systems are highly reliable one of the reasons for that is use of error correction codes.

5. Design is easy

The design of digital systems which require use of Boolean algebra and other digital techniques is easier compared to analog designing.

6. Result can be reproduced easily

Since the output of digital systems unlike analog systems is independent of temperature, noise, humidity and other characteristics of components the reproducibility of results is higher in digital systems than in analog systems.

Disadvantages of Digital Systems

- Use more energy than analog circuits to accomplish the same tasks, thus producing more heat as well.
- Digital circuits are often fragile, in that if a single piece of digital data is lost or misinterpreted the meaning of large blocks of related data can completely change.
- Digital computer manipulates discrete elements of information by means of a binary code.
- Quantization error during analog signal sampling.

NUMBER SYSTEM

Number system is a basis for counting various items. Modern computers communicate and operate with binary numbers which use only the digits 0 & 1. Basic number system used by humans is Decimal number system.

For Ex: Let us consider decimal number 18. This number is represented in binary as 10010.

We observe that binary number system takes more digits to represent the decimal number. For large numbers we have to deal with very large binary strings. So this fact gave rise to three new number systems.

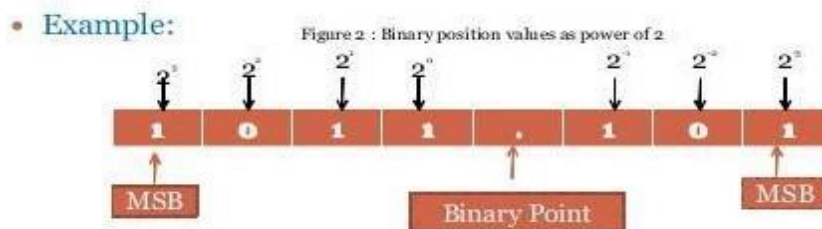
- i) Octal number systems
- ii) Hexa Decimal number system
- iii) Binary Coded Decimal number (BCD) system

To define any number system we have to specify

- Base of the number system such as 2, 8, 10 or 16.
- The base decides the total number of digits available in that number system.
- First digit in the number system is always zero and last digit in the number system is always base-1.

Binary number system:

The binary number has a radix of 2. As $r = 2$, only two digits are needed, and these are 0 and 1. In binary system weight is expressed as power of 2.



The left most bit, which has the greatest weight is called the Most Significant Bit (MSB). And the right most bit which has the least weight is called Least Significant Bit (LSB).

For Ex: $1001.01_2 = [(1) \times 2^3] + [(0) \times 2^2] + [(0) \times 2^1] + [(1) \times 2^0] + [(0) \times 2^{-1}] + [(1) \times 2^{-2}]$

$$1001.01_2 = [1 \times 8] + [0 \times 4] + [0 \times 2] + [1 \times 1] + [0 \times 0.5] + [1 \times 0.25]$$

$$1001.01_2 = 9.25_{10}$$

Decimal Number system

The decimal system has ten symbols: 0,1,2,3,4,5,6,7,8,9. In other words, it has a base of 10.

Octal Number System

Digital systems operate only on binary numbers. Since binary numbers are often very long, two shorthand notations, octal and hexadecimal, are used for representing large binary numbers. Octal systems use a base or radix of 8. It uses first eight digits of decimal number system. Thus it has digits from 0 to 7.

Hexa Decimal Number System

The hexadecimal numbering system has a base of 16. There are 16 symbols. The decimal digits 0 to 9 are used as the first ten digits as in the decimal system, followed by the letters A, B, C, D, E and F, which represent the values 10, 11, 12, 13, 14 and 15 respectively.

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Number Base conversions

The human beings use decimal number system while computer uses binary number system. Therefore it is necessary to convert decimal number system into its equivalent binary.

- i) Binary to octal number conversion
- ii) Binary to hexa decimal number conversion

The binary number: 001 010 011 000 100 101 110 111
 └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘
The octal number: 1 2 3 0 4 5 6 7

The binary number: 0001 0010 0100 1000 1001 1010 1101 1111
 └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘
The hexadecimal number: 1 2 5 8 9 A D F

- iii) Octal to binary Conversion

Each octal number converts to 3 binary digits

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

To convert 653_8 to binary, just substitute code:

6 5 3
↓ ↓ ↓
110 101 011

- iv) Hexa to binary conversion

0100 1111 1101 0111

- v) Octal to Decimal conversion

Ex: convert 4057.06_8 to octal

$$=4 \times 8^3 + 0 \times 8^2 + 5 \times 8^1 + 7 \times 8^0 + 0 \times 8^{-1} + 6 \times 8^{-2}$$

$$=2048 + 0 + 40 + 7 + 0 + 0.0937$$

$$=2095.0937_{10}$$

vi) Decimal to Octal Conversion

Ex: convert 378.93_{10} to octal

378_{10} to octal: Successive division:

$$\begin{array}{r} 8 \overline{) 378} \\ \underline{8 \mid 47} \quad \text{---} \quad 2 \\ \underline{8 \mid 5} \quad \text{---} \quad 7 \quad \uparrow \\ \underline{0} \quad \text{---} \quad 5 \end{array}$$

$$=572_8$$

0.93_{10} to octal :

$$0.93 \times 8 = 7.44$$

$$0.44 \times 8 = 3.52 \quad \downarrow$$

$$0.53 \times 8 = 4.16$$

$$0.16 \times 8 = 1.28$$

$$=0.7341_8$$

$$378.93_{10} = 572.7341_8$$

vii) Hexadecimal to Decimal Conversion

Ex: $5C7_{16}$ to decimal

$$=(5 \times 16^2) + (C \times 16^1) + (7 \times 16^0)$$

$$=1280 + 192 + 7$$

$$=147_{10}$$

viii) Decimal to Hexadecimal Conversion

Ex: 2598.6751_{10}

$$\begin{array}{r} 16 \overline{) 2598} \\ \underline{16 \mid 162} \quad -6 \\ 10 \quad \quad -2 \end{array}$$

$$= A26_{(16)}$$

$$0.675_{10} = 0.675 \times 16 \rightarrow 10.8$$

$$= 0.800 \times 16 \rightarrow 12.8 \quad \downarrow$$

$$= 0.800 \times 16 \rightarrow 12.8$$

$$= 0.800 \times 16 \rightarrow 12.8$$

$$= 0.ACCC_{16}$$

$$2598.675_{10} = A26.ACCC_{16}$$

ix) Octal to hexadecimal conversion:

The simplest way is to first convert the given octal no. to binary & then the binary no. to hexadecimal.

Ex: 756.603_8

7	5	6	.	6	0	3
111	101	110	.	110	000	011
0001	1110	1110	.	1100	0001	1000
1	E	E	.	C	1	8

x) Hexadecimal to octal conversion:

First convert the given hexadecimal no. to binary & then the binary no. to octal.

Ex: $B9F.AE_{16}$

B	9	F	.	A	E		
1011	1001	1111	.	1010	1110		
101	110	011	111	.	101	011	100
5	6	3	7	.	5	3	4

$$= 5637.534$$

Complements:

In digital computers to simplify the subtraction operation & for logical manipulation complements are used. There are two types of complements used in each radix system.

- The radix complement or r 's complement
- The diminished radix complement or $(r-1)$'s complement

Representation of signed no.s binary arithmetic in computers:

- Two ways of rep signed no.s
 1. Sign Magnitude form
 2. Complement form
- Two complimented forms
 1. 1's compliment form
 2. 2's compliment form

Advantage of performing subtraction by the compliment method is reduction in the hardware.(instead of addition & subtraction only adding ckt's are needed.)
i.e, subtraction is also performed by adders only.

Instead of subtracting one no. from other the compliment of the subtrahend is added to minuend. In sign magnitude form, an additional bit called the sign bit is placed in front of the no. If the sign bit is 0, the no. is +ve, If it is a 1, the no is _ve.

Ex:

0	1	0	1	0	0	1
---	---	---	---	---	---	---

Sign bit = +41 magnitude

1	1	0	1	0	0	1
---	---	---	---	---	---	---

$= -41$

Note: manipulation is necessary to add a +ve no to a -ve no

Representation of signed no.s using 2's or 1's complement method:

If the no. is +ve, the magnitude is rep in its true binary form & a sign bit 0 is placed in front of the MSB. If the no is _ve , the magnitude is rep in its 2's or 1's compliment form & a sign bit 1 is placed in front of the MSB.

Ex:

Given no.	Sign mag form	2's comp form	1's comp form
01101	+13	+13	+13
010111	+23	+23	+23
10111	-7	-7	-8
1101010	-42	-22	-21

Special case in 2's comp representation:

Whenever a signed no. has a 1 in the sign bit & all 0's for the magnitude bits, the decimal equivalent is -2^n , where n is the no of bits in the magnitude .

Ex: 1000= -8 & 10000=-16

Characteristics of 2's compliment no.s:

Properties:

1. There is one unique zero
2. 2's comp of 0 is 0
3. The leftmost bit can't be used to express a quantity . it is a 0 no. is +ve.
4. For an n-bit word which includes the sign bit there are $(2^{n-1}-1)$ +ve integers, 2^{n-1} -ve integers & one 0 , for a total of 2^n unique states.
5. Significant information is contained in the 1's of the +ve no.s & 0's of the -ve no.s
6. A -ve no. may be converted into a +ve no. by finding its 2's comp.

Signed binary numbers:

Decimal	Sign 2's comp form	Sign 1's comp form	Sign mag form
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0011	0011	0011
+0	0000	0000	0000

-0	--	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
8	1000	--	--

Methods of obtaining 2's comp of a no:

- In 3 ways
 1. By obtaining the 1's comp of the given no. (by changing all 0's to 1's & 1's to 0's) & then adding 1.
 2. By subtracting the given n bit no N from 2^n
 3. Starting at the LSB , copying down each bit upto & including the first 1 bit encountered , and complimenting the remaining bits.
- Ex: Express -45 in 8 bit 2's comp form

+45 in 8 bit form is 00101101

I method:

1's comp of 00101101 & the add 1

00101101

11010010

+1

— — — — — — — — — —

11010011

is 2's comp form

II method:

Subtract the given no. N from 2^n

$2^n = 100000000$

Subtract 45 = -00101101

+1

— — — —

11010011

is 2's comp

III method:

Original no: 00101101

Copy up to First 1 bit 1

Compliment remaining : 1101001

—————

bits

11010011

Ex:

-73.75 in 12 bit 2's comp form

I method

$$\begin{array}{r} 01001001.1100 \\ 10110110.0011 \\ +1 \\ \hline \end{array}$$

10110110.0100 is 2's

II method:

$$2^8 = 100000000.0000$$

Sub 73.75 = -01001001.1100

10110110.0100 is 2's comp

III method :

Original no : 01001001.1100

Copy up to 1'st bit 100

Comp the remaining bits: 10110110.0

10110110.0100

2's compliment Arithmetic:

- The 2's comp system is used to rep -ve no.s using modulus arithmetic . The word length of a computer is fixed. i.e, if a 4 bit no. is added to another 4 bit no . the result will be only of 4 bits. Carry if any , from the fourth bit will overflow called the Modulus arithmetic.

Ex: 1100 + 1111 = 1011

- In the 2's compl subtraction, add the 2's comp of the subtrahend to the minuend . If there is a carry out , ignore it , look at the sign bit I.e, MSB of the sum term .If the MSB is a 0, the result is positive.& it is in true binary form. If the MSB is a 1 (carry in or no carry at all) the result is negative.& is in its 2's comp form. Take its 2's comp to find its magnitude in binary.

Ex: Subtract 14 from 46 using 8 bit 2's comp arithmetic:

$$\begin{array}{rcl} +14 & = & 00001110 \\ -14 & = & 11110010 \quad \text{2's comp} \\ +46 & = & 00101110 \\ -14 & = & 11110010 \quad \text{2's comp form of -14} \\ \hline \end{array}$$

$$\begin{array}{r} \text{---} \\ -32 \quad (1)00100000 \end{array} \quad \text{ignore carry}$$

Ignore carry, The MSB is 0. so the result is +ve. & is in normal binary form. So the result is +00100000=+32.

EX: Add -75 to +26 using 8 bit 2's comp arithmetic

$$\begin{array}{r} +75 = 01001011 \\ -75 = 10110101 \quad \text{2's comp} \\ \hline +26 = 00011010 \\ -75 = +10110101 \quad \text{2's comp form of -75} \\ \hline -49 \quad 11001111 \quad \text{No carry} \end{array}$$

No carry, MSB is a 1, result is -ve & is in 2's comp. The magnitude is 2's comp of 11001111. i.e, 00110001 = 49. so result is -49

Ex: add -45.75 to +87.5 using 12 bit arithmetic

$$\begin{array}{r} +87.5 = 01010111.1000 \\ -45.75 = +11010010.0100 \end{array}$$

$$\begin{array}{r} \text{---} \\ -41.75 \quad (1)00101001.1100 \text{ ignore carry} \\ \text{MSB is 0, result is +ve. } = +41.75 \end{array}$$

1's compliment of n number:

- It is obtained by simply complimenting each bit of the no., & also, 1's comp of a no, is subtracting each bit of the no. from 1. This complemented value rep the -ve of the original no. One of the difficulties of using 1's comp is its rep of zero. Both 00000000 & its 1's comp 11111111 rep zero.
- The 00000000 called +ve zero & 11111111 called -ve zero.

Ex: -99 & -77.25 in 8 bit 1's comp

$$\begin{array}{r} +99 = 01100011 \\ -99 = 10011100 \end{array}$$

$$\begin{array}{r} +77.25 = 01001101.0100 \\ -77.25 = 10110010.1011 \end{array}$$

1's compliment arithmetic:

In 1's comp subtraction, add the 1's comp of the subtrahend to the minuend. If there is a carryout, bring the carry around & add it to the LSB called the **end around carry**. Look at the sign bit (MSB). If this is a 0, the result is +ve & is in true binary. If the MSB is a 1 (carry or no carry), the result is -ve & is in its 1's comp form. Take its 1's comp to get the magnitude in binary.

Ex: Subtract 14 from 25 using 8 bit 1's EX: ADD -25 to +14

$$\begin{array}{rcl}
 25 & = & 00011001 \\
 -45 & = & 11110001 \\
 \hline
 +11 & & (1)00001010 \\
 +1 & & \hline
 & & 00001011
 \end{array}
 \qquad
 \begin{array}{rcl}
 +14 & = & 00001110 \\
 -25 & = & +11100110 \\
 \hline
 -11 & & 11110100 \\
 \hline
 & & \text{No carry MSB}=1 \\
 & & \text{result}=-\text{ve}=-11_{10}
 \end{array}$$

MSB is a 0 so result is +ve (binary)

$=+11_{10}$

Binary codes

Binary codes are codes which are represented in binary system with modification from the original ones.

- ☐ Weighted Binary codes
- ☐ Non Weighted Codes

Weighted binary codes are those which obey the positional weighting principles, each position of the number represents a specific weight. The binary counting sequence is an example.

Decimal	BCD 8421	Excess-3	84-2-1	2421	5211	Bi-Quinary 5043210			5	0	4	3	2	1	0
0	0000	0011	0000	0000	0000	0100001		0		X					X
1	0001	0100	0111	0001	0001	0100010		1		X				X	
2	0010	0101	0110	0010	0011	0100100		2		X			X		
3	0011	0110	0101	0011	0101	0101000		3		X		X			
4	0100	0111	0100	0100	0111	0110000		4		X	X				
5	0101	1000	1011	1011	1000	1000001		5	X						X
6	0110	1001	1010	1100	1010	1000010		6	X					X	
7	0111	1010	1001	1101	1100	1000100		7	X				X		
8	1000	1011	1000	1110	1110	1001000		8	X			X			
9	1001	1111	1111	1111	1111	1010000		9	X		X				

Reflective Code

A code is said to be reflective when code for 9 is complement for the code for 0, and

so is for 8 and 1 codes, 7 and 2, 6 and 3, 5 and 4. Codes 2421, 5211, and excess-3 are reflective, whereas the 8421 code is not.

Sequential Codes

A code is said to be sequential when two subsequent codes, seen as numbers in binary representation, differ by one. This greatly aids mathematical manipulation of data. The 8421 and Excess-3 codes are sequential, whereas the 2421 and 5211 codes are not.

Non weighted codes

Non weighted codes are codes that are not positionally weighted. That is, each position within the binary number is not assigned a fixed value. Ex: Excess-3 code

Excess-3 Code

Excess-3 is a non weighted code used to express decimal numbers. The code derives its name from the fact that each binary code is the corresponding 8421 code plus 0011(3).

Gray Code

The gray code belongs to a class of codes called minimum change codes, in which only one bit in the code changes when moving from one code to the next. The Gray code is non-weighted code, as the position of bit does not contain any weight. The gray code is a reflective digital code which has the special property that any two subsequent numbers codes differ by only one bit. This is also called a unit- distance code. In digital Gray code has got a special place.

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

Binary to Gray Conversion

- ☐ Gray Code MSB is binary code MSB.
- ☐ Gray Code MSB-1 is the XOR of binary code MSB and MSB-1.
- ☐ MSB-2 bit of gray code is XOR of MSB-1 and MSB-2 bit of binary code.
- ☐ MSB-N bit of gray code is XOR of MSB-N-1 and MSB-N bit of binary code.

8421 BCD code (Natural BCD code):

Each decimal digit 0 through 9 is coded by a 4 bit binary no. called natural binary codes. Because of the 8,4,2,1 weights attached to it. It is a weighted code & also sequential . it is useful for mathematical operations. The advantage of this code is its ease of conversion to & from decimal. It is less efficient than the pure binary, it requires more bits.

Ex: 14→1110 in binary

But as 0001 0100 in 8421 code.

The disadvantage of the BCD code is that , arithmetic operations are more complex than they are in pure binary . There are 6 illegal combinations 1010,1011,1100,1101,1110,1111 in these codes, they are not part of the 8421 BCD code system . The disadvantage of 8421 code is, the rules of binary addition 8421 no, but only to the individual 4 bit groups.

BCD Addition:

It is individually adding the corresponding digits of the decimal no,s expressed in 4 bit binary groups starting from the LSD . If there is no carry & the sum term is not an illegal code , no correction is needed .If there is a carry out of one group to the next group or if the sum term is an illegal code then $6_{10}(0100)$ is added to the sum term of that group & the resulting carry is added to the next group.

Ex: Perform decimal additions in 8421 code

(a)25+13

In BCD 25= 0010 0101

In BCD +13 =+0001 0011

38 0011 1000

No carry , no illegal code .This is the corrected sum

(b). 679.6 + 536.8

679.6	=	0110	0111	1001	.0110 in BCD	
+536.8	=	+0101	0011	0010	.1000 in BCD	
<hr style="border: none; border-top: 1px dashed black;"/>						
1216.4		1011	1010	0110	.1110	illegal codes
		+0110	+ 0011	+0110	. + 0110	add 0110 to each
<hr style="border: none; border-top: 1px dashed black;"/>						
		(1)0001	(1)0000	(1)0101	. (1)0100	propagate carry
		/	/	/	/	
		+1	+1	+1	+1	
		0001	0010	0001	0110	. 0100
		1	2	1	6	. 4

BCD Subtraction:

Performed by subtracting the digits of each 4 bit group of the subtrahend the digits from the corresponding 4- bit group of the minuend in binary starting from the LSD . if there is no borrow from the next group , then $6_{10}(0110)$ is subtracted from the difference term of this group.

(a)38-15

In BCD	38=	0011	1000
In BCD	-15 =	-0001	0101

23 0010 0011

No borrow, so correct difference.

(b) 206.7-147.8

206.7	=	0010	0000	0110	.	0111	in BCD
-147.8	=	-0001	0100	0111	.	0110	in BCD
<hr style="border: none; border-top: 1px dashed black;"/>							
58.9		0000	1011	1110	.	1111	borrows are present
		-0110	-0110	.		-0110	subtract 0110
<hr style="border: none; border-top: 1px solid black;"/>							
		0101	1000	.		1001	

BCD Subtraction using 9's & 10's compliment methods:

Form the 9's & 10's compliment of the decimal subtrahend & encode that no. in the 8421 code . the resulting BCD no.s are then added.

EX: 305.5 – 168.8

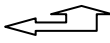
$$\begin{array}{r} 305.5 = 305.5 \\ -168.8 = +83.1 \quad \text{9's comp of -168.8} \\ \hline (1)136.6 \\ \quad +1 \quad \text{end around carry} \\ \quad \mathbf{136.7} \quad \text{corrected difference} \\ 305.5_{10} = 0011 \ 0000 \ 0101 \ . \ 0101 \\ +831.1_{10} = +1000 \ 0011 \ 0001 \ . \ 0001 \quad \text{9's comp of 168.8 in BCD} \\ \hline +1011 \ 0011 \ 0110 \ . \ 0110 \quad \text{1011 is illegal code} \\ +0110 \quad \text{add 0110} \\ \hline (1)0001 \ 0011 \ 0110 \ . \ 0110 \\ \quad +1 \quad \text{End around carry} \\ \hline 0001 \ 0011 \ 0110 \ . \ 0111 \\ = 136.7 \end{array}$$

Excess three(xs-3)code:

It is a non-weighted BCD code .Each binary codeword is the corresponding 8421 codeword plus 0011(3).It is a sequential code & therefore , can be used for arithmetic operations..It is a self-complementing code.s o the subtraction by the method of compliment addition is more direct in xs-3 code than that in 8421 code. The xs-3 code has six invalid states 0000,0010,1101,1110,1111.. It has interesting properties when used in addition & subtraction.

Excess-3 Addition:

Add the xs-3 no.s by adding the 4 bit groups in each column starting from the LSD. If there is no carry starting from the addition of any of the 4-bit groups , subtract 0011 from the sum term of those groups (because when 2 decimal digits are added in xs-3 & there is no carry , result in xs-6). If there is a carry out, add 0011 to the sum term of those groups(because when there is a carry, the invalid states are skipped and the result is normal binary).

EX:	37	0110	1010	
	+28	+0101	1011	
	<hr/>			
	65	1011	(1)0101	carry generated
		+1		propagate carry
		<hr/>		
		1100	0101	add 0011 to correct 0101 &
		-0011	+0011	subtract 0011 to correct 1100
		<hr/>		
		1001	1000	=65 ₁₀

Excess -3 (XS-3) Subtraction:

Subtract the xs-3 no.s by subtracting each 4 bit group of the subtrahend from the corresponding 4 bit group of the minuend starting from the LSD .if there is no borrow from the next 4-bit group add 0011 to the difference term of such groups (because when decimal digits are subtracted in xs-3 & there is no borrow , result is normal binary). If there is a borrow , subtract 0011 from the difference term (because taking a borrow is equivalent to adding six invalid states , result is in xs-6)

Ex: 267-175

267 =	0101	1001	1010	
-175 =	-0100	1010	1000	
	<hr/>			
	0000	1111	0010	
	+0011	-0011	+0011	
	<hr/>			
	0011	1100	+0011	=92 ₁₀

Ex:687-348 The subtrahend (348) xs -3 code & its compliment are:

Xs=3 code of 348 in xs=3 = 1001 1000 0100

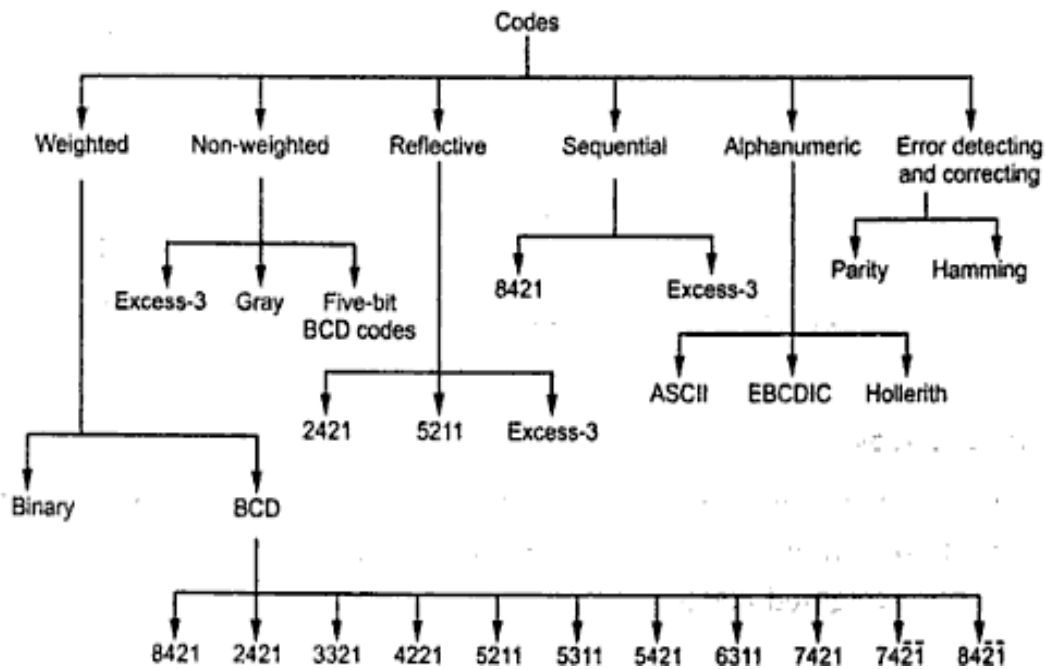
0110 0110 1100 corrected diff in xs-3 = 330₁₀

The Gray code (reflective –code):

Gray code is a non-weighted code & is not suitable for arithmetic operations. It is not a BCD code . It is a cyclic code because successive code words in this code differ in one bit position only i.e, it is a unit distance code. Popular of the unit distance code. It is also a reflective code i.e, both reflective & unit distance. The n least significant bits for 2^n through $2^{n+1}-1$ are the mirror images of those for 0 through 2^n-1 . An N bit gray code can be obtained by reflecting an $N-1$ bit code about an axis at the end of the code, & putting the MSB of 0 above the axis & the MSB of 1 below the axis.

Reflection of gray codes:

Gray Code				Decimal	4 bit binary
1 bit	2 bit	3 bit	4 bit		
0	00	000	0000	0	0000
1	01	001	0001	1	0001
	11	011	0011	2	0010
	10	010	0010	3	0011
		110	0110	4	0100
		111	0111	5	0101
		101	0101	6	0110
		110	0100	7	0111
			1100	8	1000
			1101	9	1001
			1111	10	1010
			1110	11	1011
			1010	12	1100
			1011	13	1101
			1001	14	1110
			1000	15	1111



Binary codes block diagram

Error – Detecting codes: When binary data is transmitted & processed, it is susceptible to noise that can alter or distort its contents. The 1's may get changed to 0's & 1's. Because digital systems must be accurate to the digit, error can pose a problem. Several schemes have been devised to detect the occurrence of a single bit error in a binary word, so that whenever such an error occurs the concerned binary word can be corrected & retransmitted.

Parity: The simplest technique for detecting errors is that of adding an extra bit known as parity bit to each word being transmitted. Two types of parity: Odd parity, even parity. For odd parity, the parity bit is set to a 0 or a 1 at the transmitter such that the total no. of 1 bit in the word including the parity bit is an odd no. For even parity, the parity bit is set to a 0 or a 1 at the transmitter such that the parity bit is an even no.

Decimal	8421 code	Odd parity	Even parity
0	0000	1	0
1	0001	0	1
2	0010	0	1
3	0011	1	0
4	0100	0	1
5	0100	1	0
6	0110	1	0
7	0111	0	1
8	1000	0	1
9	1001	1	0

When the digit data is received . a parity checking circuit generates an error signal if the total no of 1's is even in an odd parity system or odd in an even parity system. This parity check can always detect a single bit error but cannot detect 2 or more errors with in the same word. Odd parity is used more often than even parity does not detect the situation. Where all 0's are created by a short ckt or some other fault condition.

Ex: Even parity scheme

(a) 10101010 (b) 11110110 (c) 10111001

Ans:

- (a) No. of 1's in the word is even is 4 so there is no error
- (b) No. of 1's in the word is even is 6 so there is no error
- (c) No. of 1's in the word is odd is 5 so there is error

Ex: odd parity

(a) 10110111 (b) 10011010 (c) 11101010

Ans:

- (a) No. of 1's in the word is even is 6 so word has error
- (b) No. of 1's in the word is even is 4 so word has error
- (c) No. of 1's in the word is odd is 5 so there is no error

Checksums:

Simple parity can't detect two errors within the same word. To overcome this, use a sort of 2 dimensional parity. As each word is transmitted, it is added to the sum of the previously transmitted words, and the sum retained at the transmitter end. At the end of transmission, the sum called the check sum. Up to that time sent to the receiver. The receiver can check its sum with the transmitted sum. If the two sums are the same, then no errors were detected at the receiver end. If there is an error, the receiving location can ask for retransmission of the entire data, used in teleprocessing systems.

Block parity:

Block of data shown is create the row & column parity bits for the data using odd parity. The parity bit 0 or 1 is added column wise & row wise such that the total no. of 1's in each column & row including the data bits & parity bit is odd as

Data	Parity bit	data
10110	0	10110
10001	1	10001
10101	0	10101
00010	0	00010
11000	1	11000
00000	1	00000
11010	0	11010

Error –Correcting Codes:

A code is said to be an error –correcting code, if the code word can always be deduced from an erroneous word. For a code to be a single bit error correcting code, the minimum distance of that code must be three. The minimum distance of that code is the smallest no. of bits by which any two code words must differ. A code with minimum distance of 3 can't only correct single bit errors but also detect (can't correct) two bit errors, The key to error correction is that it must be possible to detect & locate erroneous that it must be possible to detect & locate erroneous digits. If the location of an error has been determined. Then by complementing the erroneous digit, the message can be corrected , error correcting , code is the Hamming code , In this , to each group of m information or message or data bits, K parity checking bits denoted by P₁,P₂,-----p_k located at positions 2^{k-1} from left are added to form an (m+k) bit code word. To correct the error, k parity checks are performed on selected digits of each code word, & the position of the error bit is located by forming an error word, & the error bit is then complemented. The k bit error word is generated by putting a 0 or a 1 in the 2^{k-1} th position depending upon whether the check for parity involving the parity bit P_k is satisfied or not. Error positions & their corresponding values :

Error Position	For 15 bit code C ₄ C ₃ C ₂ C ₁	For 12 bit code C ₄ C ₃ C ₂ C ₁	For 7 bit code C ₃ C ₂ C ₁
0	0 0 0 0	0 0 0 0	0 0 0
1	0 0 0 1	0 0 0 1	0 0 1
2	0 0 1 0	0 0 1 0	0 1 0
3	0 0 1 1	0 0 1 1	0 1 1
4	0 1 0 0	0 1 0 0	1 0 0
5	0 1 0 1	0 1 0 1	1 0 1
6	0 1 1 0	0 1 1 0	1 1 0
7	0 1 1 1	0 1 1 1	1 1 1
8	1 0 0 0	1 0 0 0	
9	1 0 0 1	1 0 0 1	
10	1 0 1 0	1 0 1 0	
11	1 0 1 1	1 0 1 1	
12	1 1 0 0	1 1 0 0	
13	1 1 0 1		
14	1 1 1 0		
15	1 1 1 1		

7-bit Hamming code:

To transmit four data bits, 3 parity bits located at positions 2^0 , 2^1 & 2^2 from left are added to make a 7 bit codeword which is then transmitted.

The word format

P ₁	P ₂	D ₃	P ₄	D ₅	D ₆	D ₇
----------------	----------------	----------------	----------------	----------------	----------------	----------------

D—Data bits P-

Parity bits

Decimal Digit	For BCD P ₁ P ₂ D ₃ P ₄ D ₅ D ₆ D ₇	For Excess-3 P ₁ P ₂ D ₃ P ₄ D ₅ D ₆ D ₇
0	0 0 0 0 0 0 0	1 0 0 0 0 1 1
1	1 1 0 1 0 0 1	1 0 0 1 1 0 0
2	0 1 0 1 0 1 1	0 1 0 0 1 0 1
3	1 0 0 0 0 1 1	1 1 0 0 1 1 0
4	1 0 0 1 1 0 0	0 0 0 1 1 1 1
5	0 1 0 0 1 0 1	1 1 1 0 0 0 0
6	1 1 0 0 1 1 0	0 0 1 1 0 0 1
7	0 0 0 1 1 1 1	1 0 1 1 0 1 0
8	1 1 1 0 0 0 0	0 1 1 0 0 1 1
9	0 0 1 1 0 0 1	0 1 1 1 1 0 0

Ex: Encode the data bits 1101 into the 7 bit even parity Hamming Code

The bit pattern is

$P_1 P_2 D_3 P_4 D_5 D_6 D_7$

1 1 0 1

Bits 1,3,5,7 (P_1 111) must have even parity, so $P_1=1$

Bits 2, 3, 6, 7(P_2 101) must have even parity, so $P_2=0$

Bits 4,5,6,7 (P_4 101) must have even parity, so $P_4=0$

The final code is 1010101

EX: Code word is 1001001

Bits 1,3,5,7 (C_1 1001) →no error →put a 0 in the 1's position→ $C_1=0$

Bits 2, 3, 6, 7(C_2 0001)) → error →put a 1 in the 2's position→ $C_2=1$

Bits 4,5,6,7 (C_4 1001)) →no error →put a 0 in the 4's position→ $C_3=0$

15-bit Hamming Code: It transmit 11 data bits, 4 parity bits located $2^0 2^1 2^2 2^3$

Word format is

P_1	P_2	D_3	P_4	D_5	D_6	D_7	P_8	D_9	D_{10}	D_{11}	D_{12}	D_{13}	D_{14}	D_{15}
-------	-------	-------	-------	-------	-------	-------	-------	-------	----------	----------	----------	----------	----------	----------

12-Bit Hamming Code:It transmit 8 data bits, 4 parity bits located at position $2^0 2^1 2^2 2^3$

Word format is









P_1	P_2	D_3	P_4	D_5	D_6	D_7	P_8	D_9	D_{10}	D_{11}	D_{12}
-------	-------	-------	-------	-------	-------	-------	-------	-------	----------	----------	----------

Alphanumeric Codes:

These codes are used to encode the characteristics of alphabet in addition to the decimal digits. It is used for transmitting data between computers & its I/O device such as printers, keyboards & video display terminals. Popular modern alphanumeric codes are ASCII code & EBCDIC code.

Digital Logic Gates

Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these type of gates.

Name	Graphic symbol	Algebraic function	Truth table															
AND		$F = x \cdot y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = x + y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	1
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$F = x'$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	x	F	0	1	1	0									
x	F																	
0	1																	
1	0																	
Buffer		$F = x$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	x	F	0	0	1	1									
x	F																	
0	0																	
1	1																	
NAND		$F = (xy)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = (x + y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	0
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

Properties of XOR Gates

- XOR (also \oplus) : the “not-equal” function
- $\text{XOR}(X,Y) = X \oplus Y = X'Y + XY'$
- Identities:
 - $X \oplus 0 = X$
 - $X \oplus 1 = X'$
 - $X \oplus X = 0$
 - $X \oplus X' = 1$
- Properties:
 - $X \oplus Y = Y \oplus X$
 - $(X \oplus Y) \oplus W = X \oplus (Y \oplus W)$

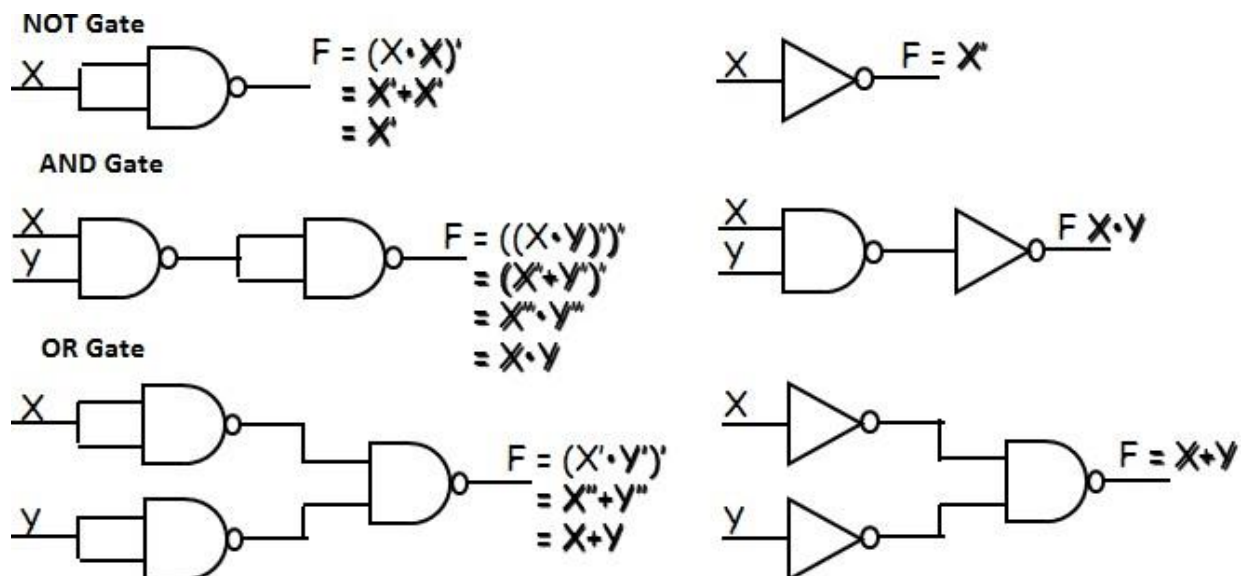
Universal Logic Gates

NAND and NOR gates are called Universal gates. All fundamental gates (NOT, AND, OR) can be realized by using either only NAND or only NOR gate. A universal gate provides flexibility and offers enormous advantage to logic designers.

NAND as a Universal Gate

NAND Known as a “universal” gate because ANY digital circuit can be implemented with NAND gates alone.

To prove the above, it suffices to show that AND, OR, and NOT can be implemented using NAND gates only.



Boolean Algebra: In 1854, George Boole developed an algebraic system now called Boolean algebra. In 1938, Claude E. Shannon introduced a two-valued Boolean algebra called switching algebra that represented the properties of bistable electrical switching circuits. For the formal definition of Boolean algebra, we shall employ the postulates formulated by E. V. Huntington in 1904.

Boolean algebra is a system of mathematical logic. It is an algebraic system consisting of the set of elements (0, 1), two binary operators called OR, AND, and one unary operator NOT. It is the basic mathematical tool in the analysis and synthesis of switching circuits. It is a way to express logic functions algebraically.

Boolean algebra, like any other deductive mathematical system, may be defined with a set of elements, a set of operators, and a number of unproved axioms or postulates. A *set* of elements is any collection of objects having a common property. If S is a set and x and y are certain objects, then $x \in S$ denotes that x is a member of the set S , and $y \notin S$ denotes that y is not an element of S . A set with a denumerable number of elements is specified by braces: $A = \{1, 2, 3, 4\}$, i.e. the elements of set A are the numbers 1, 2, 3, and 4. A *binary operator* defined on a set S of elements is a rule that assigns to each pair of elements from S a unique element from S . Example: In $a * b = c$, we say that $*$ is a binary operator if it specifies a rule for finding c from the pair (a, b) and also if $a, b, c \in S$.

Axioms and laws of Boolean algebra

Axioms or Postulates of Boolean algebra are a set of logical expressions that we accept without proof and upon which we can build a set of useful theorems.

	AND Operation	OR Operation	NOT Operation
Axiom1 :	$0.0=0$	$0+0=0$	$\overline{0}=1$
Axiom2:	$0.1=0$	$0+1=1$	$\overline{1}=0$
Axiom3:	$1.0=0$	$1+0=1$	
Axiom4:	$1.1=1$	$1+1=1$	

AND Law

Law1: $A.0=0$ (Null law)
 Law2: $A.1=A$ (Identity law)
 Law3: $A.A=A$ (Idempotence law)

OR Law

Law1: $A+0=A$
 Law2: $A+1=1$
 Law3: $A+A=A$ (Idempotence law)

CLOSURE: The Boolean system is *closed* with respect to a binary operator if for every pair of Boolean values, it produces a Boolean result. For example, logical AND is closed in the Boolean system because it accepts only Boolean operands and produces only Boolean results.

_ A set S is closed with respect to a binary operator if, for every pair of elements of S , the binary operator specifies a rule for obtaining a unique element of S .

_ For example, the set of natural numbers $N = \{1, 2, 3, 4, \dots, 9\}$ is closed with respect to the binary operator plus (+) by the rule of arithmetic addition, since for any $a, b \in N$ we obtain a unique $c \in N$ by the operation $a + b = c$.

ASSOCIATIVE LAW:

A binary operator $*$ on a set S is said to be associative whenever $(x * y) * z = x * (y * z)$ for all $x, y, z \in S$, for all Boolean values x, y and z .

COMMUTATIVE LAW:

A binary operator $*$ on a set S is said to be commutative whenever $x * y = y * x$ for all $x, y, z \in S$

IDENTITY ELEMENT:

A set S is said to have an identity element with respect to a binary operation $*$ on S if there exists an element $e \in S$ with the property $e * x = x * e = x$ for every $x \in S$

BASIC IDENTITIES OF BOOLEAN ALGEBRA

- *Postulate 1(Definition):* A Boolean algebra is a closed algebraic system containing a set K of two or more elements and the two operators \cdot and $+$ which refer to logical AND and logical OR • $x + 0 = x$
- $x \cdot 0 = 0$
- $x + 1 = 1$
- $x \cdot 1 = x$
- $x + x = x$
- $x \cdot x = x$
- $x + x' = 1$
- $x \cdot x' = 0$
- $x + y = y + x$
- $xy = yx$
- $x + (y + z) = (x + y) + z$
- $x(yz) = (xy)z$
- $x(y + z) = xy + xz$
- $x + yz = (x + y)(x + z)$
- $(x + y)' = x'y'$
- $(xy)' = x' + y'$

- $(x')' = x$

DeMorgan's Theorem

(a) $(a + b)' = a'b'$

(b) $(ab)' = a' + b'$

Generalized DeMorgan's Theorem

(a) $(a + b + \dots z)' = a'b' \dots z'$

(b) $(a.b \dots z)' = a' + b' + \dots z'$

Basic Theorems and Properties of Boolean algebra Commutative law

Law1: $A+B=B+A$

Law2: $A.B=B.A$

Associative law

Law1: $A + (B + C) = (A + B) + C$

Law2: $A(B.C) = (A.B)C$

Distributive law

Law1: $A.(B + C) = AB + AC$

Law2: $A + BC = (A + B).(A + C)$

Absorption law

Law1: $A + AB = A$

Law2: $A(A + B) = A$

Solution: $\frac{A(1+B)}{A}$

—

Solution: $\frac{A.A+A.B}{A+A.B}$
 $\frac{A(1+B)}{A}$

Consensus Theorem

Theorem1. $AB + A'C + BC = AB + A'C$ Theorem2. $(A+B).(A'+C).(B+C) = (A+B).(A'+C)$

The BC term is called the consensus term and is redundant. The consensus term is formed from a PAIR OF TERMS in which a variable (A) and its complement (A') are present; the consensus term is formed by multiplying the two terms and leaving out the selected variable and its complement

Consensus Theorem1 Proof:

$$\begin{aligned} AB + A'C + BC &= AB + A'C + (A + A')BC \\ &= AB + A'C + ABC + A'BC \end{aligned}$$

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

$$= AB+ A'C$$

Principle of Duality

Each postulate consists of two expressions statement one expression is transformed into the other by interchanging the operations (+) and (·) as well as the identity elements 0 and 1.

Such expressions are known as duals of each other.

If some equivalence is proved, then its dual is also immediately true.

If we prove: $(x.x)+(x'+x')=1$, then we have by duality: $(x+x)·(x'·x')=0$

The Huntington postulates were listed in pairs and designated by part (a) and part (b) in below table.

Table for Postulates and Theorems of Boolean algebra

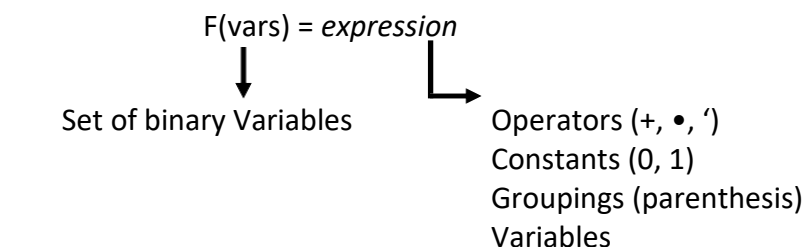
Part-A	Part-B
$A+0=A$	$A·0=0$
$A+1=1$	$A·1=A$
$A+A=A$ (Idempotence law)	$A·A=A$ (Idempotence law)
$A+\bar{A}1$	$A·\bar{A}0$
$\bar{\bar{A}}A$ (double inversion law)	--
Commutative law: $A+B=B+A$	$A·B=B·A$
Associative law: $A + (B + C) = (A + B) + C$	$A(B·C) = (A·B)C$
Distributive law: $A·(B + C) = AB+ AC$	$A + BC = (A + B)·(A + C)$
Absorption law: $A +AB =A$	$A(A +B) = A$
DeMorgan Theorem: $\overline{(A+B)} = \bar{A}·\bar{B}$	$\overline{(A·B)} = \bar{A} + \bar{B}$
Redundant Literal Rule: $A + \bar{A}B=A+B$	$A·(\bar{A}+B)=AB$
Consensus Theorem: $AB+ A'C + BC = AB + A'C$	$(A+B)·(A'+C)·(B+C)=(A+B)·(A'+C)$

Boolean Function

Boolean algebra is an algebra that deals with binary variables and logic operations.

A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols.

For a given value of the binary variables, the function can be equal to either 1 or 0.



Consider an example for the Boolean function

$$F1 = x + y'z$$

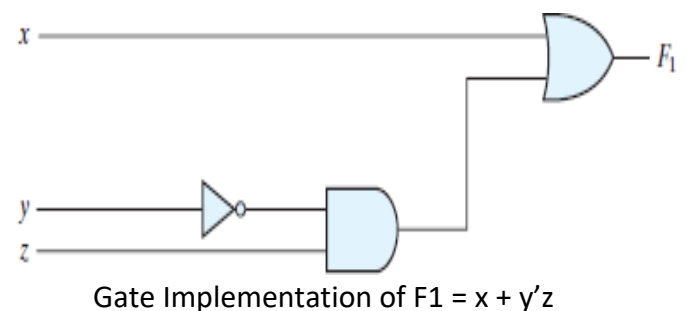
The function F_1 is equal to 1 if x is equal to 1 or if both y' and z are equal to 1. F_1 is equal to 0 otherwise. The complement operation dictates that when $y' = 1$, $y = 0$. Therefore, $F_1 = 1$ if $x = 1$ or if $y = 0$ and $z = 1$.

A Boolean function expresses the logical relationship between binary variables and is evaluated by determining the binary value of the expression for all possible values of the variables.

A Boolean function can be represented in a truth table. The number of rows in the truth table is 2^n , where n is the number of variables in the function. The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through $2^n - 1$.

Truth Table for F_1

x	y	z	F_1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



Gate Implementation of $F_1 = x + y'z$

Note:

Q: Let a function $F()$ depend on n variables. How many rows are there in the truth table of $F()$?

A: 2^n rows, since there are 2^n possible binary patterns/combinations for the n variables.

Truth Tables

- Enumerates all possible combinations of variable values and the corresponding function value
- Truth tables for some arbitrary functions
 $F_1(x,y,z)$, $F_2(x,y,z)$, and $F_3(x,y,z)$ are shown to the below.

x	y	z	F_1	F_2	F_3
0	0	0	0	1	1
0	0	1	0	0	1

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

- Truth table: a unique representation of a Boolean function
- If two functions have identical truth tables, the functions are equivalent (and vice-versa).
- Truth tables can be used to prove equality theorems.
- However, the size of a truth table grows exponentially with the number of variables involved, hence unwieldy. This motivates the use of Boolean Algebra.

Boolean expressions-NOT unique

Unlike truth tables, expressions representing a Boolean function are NOT unique.

- Example:
 - $F(x,y,z) = x' \cdot y' \cdot z' + x' \cdot y \cdot z' + x \cdot y \cdot z'$
 - $G(x,y,z) = x' \cdot y' \cdot z' + y \cdot z'$
- The corresponding truth tables for F() and G() are to the right. They are identical.
- Thus, $F() = G()$

x	y	z	F	G
0	0	0	1	1
0	0	1	0	0
0	1	0	1	1
0	1	1	0	0
1	0	0	0	0
1	0	1	0	0
1	1	0	1	1
1	1	1	0	0

Algebraic Manipulation (Minimization of Boolean function)

- Boolean algebra is a useful tool for simplifying digital circuits.
- Why do it? Simpler can mean cheaper, smaller, faster.
- Example: Simplify $F = x'yz + x'yz' + xz$.

$$\begin{aligned}
 F &= x'yz + x'yz' + xz \\
 &= x'y(z+z') + xz \\
 &= x'y \cdot 1 + xz
 \end{aligned}$$

$$= x'y + xz$$

- Example: Prove

$$x'y'z' + x'yz' + xyz' = x'z' + yz'$$

- **Proof:**

$$\begin{aligned} x'y'z' + x'yz' + xyz' \\ &= x'y'z' + x'yz' + x'yz' + xyz' \\ &= x'z'(y' + y) + yz'(x' + x) \\ &= x'z' \cdot 1 + yz' \cdot 1 \\ &= x'z' + yz' \end{aligned}$$

Complement of a Function

- The complement of a function is derived by interchanging (\bullet and $+$), and (1 and 0), and complementing each variable.
- Otherwise, interchange 1s to 0s in the truth table column showing F.
- The *complement* of a function IS NOT THE SAME as the *dual* of a function.

Example

- Find $G(x,y,z)$, the complement of $F(x,y,z) = xy'z' + x'yz$

$$\begin{aligned} \text{Ans: } G = F' &= (xy'z' + x'yz)' \\ &= (xy'z')' \bullet (x'yz)' \quad \text{DeMorgan} \\ &= (x' + y + z) \bullet (x + y' + z') \quad \text{DeMorgan again} \end{aligned}$$

Note: The complement of a function can also be derived by finding the function's *dual*, and then complementing all of the literals

Canonical and Standard Forms

We need to consider formal techniques for the simplification of Boolean functions.

Identical functions will have exactly the same canonical form.

- Minterms and Maxterms
- Sum-of-Minterms and Product-of- Maxterms
- Product and Sum terms
- Sum-of-Products (SOP) and Product-of-Sums (POS)

Definitions

Literal: A variable or its complement

Product term: literals connected by \bullet

Sum term: literals connected by $+$

Minterm: a product term in which all the variables appear exactly once, either complemented or uncomplemented.

Maxterm: a sum term in which all the variables appear exactly once, either complemented or uncomplemented.

Canonical form: Boolean functions expressed as a sum of Minterms or product of Maxterms are said to be in canonical form.

Minterm

- Represents exactly one combination in the truth table.
- Denoted by m_j , where j is the decimal equivalent of the minterm's corresponding binary combination (b_j).
- A variable in m_j is complemented if its value in b_j is 0, otherwise is uncomplemented.

Example: Assume 3 variables (A, B, C), and $j=3$. Then, $b_j = 011$ and its corresponding minterm is denoted by $m_j = A'BC$

Maxterm

- Represents exactly one combination in the truth table.
- Denoted by M_j , where j is the decimal equivalent of the maxterm's corresponding binary combination (b_j).
- A variable in M_j is complemented if its value in b_j is 1, otherwise is uncomplemented.

Example: Assume 3 variables (A, B, C), and $j=3$. Then, $b_j = 011$ and its corresponding maxterm is denoted by $M_j = A+B'+C'$

Truth Table notation for Minterms and Maxterms

- Minterms and Maxterms are easy to denote using a truth table.

Example: Assume 3 variables x, y, z (order is fixed)

x	y	z	Minterm	Maxterm
0	0	0	$x'y'z' = m_0$	$x+y+z = M_0$
0	0	1	$x'y'z = m_1$	$x+y+z' = M_1$
0	1	0	$x'yz' = m_2$	$x+y'+z = M_2$
0	1	1	$x'yz = m_3$	$x+y'+z' = M_3$
1	0	0	$xy'z' = m_4$	$x'+y+z = M_4$
1	0	1	$xy'z = m_5$	$x'+y+z' = M_5$
1	1	0	$xyz' = m_6$	$x'+y'+z = M_6$
1	1	1	$xyz = m_7$	$x'+y'+z' = M_7$

Canonical Forms

- Every function $F()$ has two canonical forms:
 - Canonical Sum-Of-Products (sum of minterms)
 - Canonical Product-Of-Sums (product of maxterms)

Canonical Sum-Of-Products:

The minterms included are those m_j such that $F() = 1$ in row j of the truth table for $F()$.

Canonical Product-Of-Sums:

The maxterms included are those M_j such that $F() = 0$ in row j of the truth table for $F()$.

Example

Consider a Truth table for $f_1(a,b,c)$ at right

The canonical sum-of-products form for f_1 is

$$f_1(a,b,c) = m_1 + m_2 + m_4 + m_6$$

$$= a'b'c + a'bc' + ab'c' + abc'$$

The canonical product-of-sums form for f_1 is

$$f_1(a,b,c) = M_0 \cdot M_3 \cdot M_5 \cdot M_7$$

$$= (a+b+c) \cdot (a+b'+c') \cdot (a'+b+c') \cdot (a'+b'+c').$$

- Observe that: $m_j = M_j'$

a	b	c	f_1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Shorthand: Σ and Π

- $f_1(a,b,c) = \Sigma m(1,2,4,6)$, where Σ indicates that this is a sum-of-products form, and $m(1,2,4,6)$ indicates that the minterms to be included are m_1 , m_2 , m_4 , and m_6 .
- $f_1(a,b,c) = \Pi M(0,3,5,7)$, where Π indicates that this is a product-of-sums form, and $M(0,3,5,7)$ indicates that the maxterms to be included are M_0 , M_3 , M_5 , and M_7 .
- Since $m_j = M_j'$ for any j ,
 $\Sigma m(1,2,4,6) = \Pi M(0,3,5,7) = f_1(a,b,c)$
-

Conversion between Canonical Forms

- Replace Σ with Π (or *vice versa*) and replace those j 's that appeared in the original form with those that do not.

- Example:

$$\begin{aligned} f_1(a,b,c) &= a'b'c + a'bc' + ab'c' + abc' \\ &= m_1 + m_2 + m_4 + m_6 \\ &= \Sigma(1,2,4,6) \\ &= \Pi(0,3,5,7) \\ &= (a+b+c) \cdot (a+b'+c') \cdot (a'+b+c') \cdot (a'+b'+c') \end{aligned}$$

Standard Forms

Another way to express Boolean functions is in standard form. In this configuration, the terms that form the function may contain one, two, or any number of literals.

There are two types of standard forms: the sum of products and products of sums.

The sum of products is a Boolean expression containing AND terms, called product terms, with one or more literals each. The sum denotes the ORing of these terms. An example of a function expressed as a sum of products is

$$F1 = y' + xy + x'yz'$$

The expression has three product terms, with one, two, and three literals. Their sum is, in effect, an OR operation.

A product of sums is a Boolean expression containing OR terms, called sum terms. Each term may have any number of literals. The product denotes the ANDing of these terms. An example of a function expressed as a product of sums is

$$F2 = x(y' + z)(x' + y + z')$$

This expression has three sum terms, with one, two, and three literals. The product is an AND operation.

Conversion of SOP from standard to canonical form

Example-1.

Express the Boolean function $F = A + B'C$ as a sum of minterms.

Solution: The function has three variables: A, B, and C. The first term A is missing two variables; therefore,

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

This function is still missing one variable, so

$$\begin{aligned} A &= AB(C + C') + AB'(C + C') \\ &= ABC + ABC' + AB'C + AB'C' \end{aligned}$$

The second term $B'C$ is missing one variable; hence,

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

Combining all terms, we have

$$\begin{aligned} F &= A + B'C \\ &= ABC + ABC' + AB'C + AB'C' + A'B'C \end{aligned}$$

But $AB'C$ appears twice, and according to theorem $(x + x = x)$, it is possible to remove one of those occurrences. Rearranging the minterms in ascending order, we finally obtain

$$\begin{aligned} F &= A'B'C + AB'C + AB'C' + ABC' + ABC \\ &= m_1 + m_4 + m_5 + m_6 + m_7 \end{aligned}$$

When a Boolean function is in its sum-of-minterms form, it is sometimes convenient to express the function in the following brief notation:

$$F(A, B, C) = \sum m(1, 4, 5, 6, 7)$$

Example-2.

Express the Boolean function $F = xy + x'z$ as a product of maxterms.

Solution: First, convert the function into OR terms by using the distributive law:

$$\begin{aligned} F &= xy + x'z = (xy + x')(xy + z) \\ &= (x + x')(y + x')(x + z)(y + z) \\ &= (x' + y)(x + z)(y + z) \end{aligned}$$

The function has three variables: x, y, and z. Each OR term is missing one variable; therefore,

$$\begin{aligned} x' + y &= x' + y + zz' = (x' + y + z)(x' + y + z') \\ x + z &= x + z + yy' = (x + y + z)(x + y' + z) \\ y + z &= y + z + xx' = (x + y + z)(x' + y + z) \end{aligned}$$

Combining all the terms and removing those which appear more than once, we finally obtain

$$\begin{aligned} F &= (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z) \\ F &= M_0 M_2 M_4 M_5 \end{aligned}$$

A convenient way to express this function is as

$$\text{follows: } F(x, y, z) = \pi M(0, 2, 4, 5)$$

The product symbol, π , denotes the ANDing of maxterms; the numbers are the indices of the maxterms of the function.

Minimization Techniques

Two-variable k-map:

A two-variable k-map can have $2^2=4$ possible combinations of the input variables A and B. Each of these combinations, $\bar{A}\bar{B}$, $\bar{A}B$, $A\bar{B}$, AB (in the SOP form) is called a minterm. The minterm may be represented in terms of their decimal designations – m0 for $\bar{A}\bar{B}$, m1 for $\bar{A}B$, m2 for $A\bar{B}$ and m3 for AB , assuming that A represents the MSB. The letter m stands for minterm and the subscript represents the decimal designation of the minterm. The presence or absence of a minterm in the expression indicates that the output of the logic circuit assumes logic 1 or logic 0 level for that combination of input variables.

The expression $f = \bar{A}\bar{B} + A\bar{B} + AB$, it can be expressed using min

$$\text{term as } F = m_0 + m_2 + m_3 = \sum m(0, 2, 3)$$

Using Truth Table:

Minterm	Inputs		Output F
	A	B	
0	0	0	1
1	0	1	0
2	1	0	1
3	1	1	1

A 1 in the output contains that particular minterm in its sum and a 0 in that column indicates that the particular minterm does not appear in the expression for output. This information can also be indicated by a two-variable k-map.

Mapping of SOP Expressions:

A two-variable k-map has $2^2=4$ squares. These squares are called cells. Each square on the k-map represents a unique minterm. The minterm designation of the squares are placed in any square, indicates that the corresponding minterm does output expressions. And a 0 or no entry in any square indicates that the corresponding minterm does not appear in the expression for output.

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

The minterms of a two-variable k-map

The mapping of the expressions $=\sum m(0,2,3)$ is

		B	
		0	1
A	0	1 ⁰	0 ¹
	1	1 ²	1 ³

k-map of $\sum m(0,2,3)$

EX: Map the expressions $f = B + A$

$F = m_1 + m_2 = \sum m(1,2)$ The k-map is

		B	
		0	1
A	0	0 ⁰	1 ¹
	1	1 ²	0 ³

Minimizations of SOP expressions:

To minimize Boolean expressions given in the SOP form by using the k-map, look for adjacent adjacent squares having 1's minterms adjacent to each other, and combine them to form larger squares to eliminate some variables. Two squares are said to be adjacent to each other, if their minterms differ in only one variable. (i.e., B & A differ only in one variable. so they may be combined to form a 2-square to eliminate the variable B. similarly all other.

The necessary condition for adjacency of minterms is that their decimal designations must differ by a power of 2. A minterm can be combined with any number of minterms adjacent to it to form larger squares. Two minterms which are adjacent to each other can be combined to form a bigger square called a 2-square or a pair. This eliminates one variable – the variable that is not common to both the minterms. For EX:

m_0 and m_1 can be combined to yield,

$$f_1 = m_0 + m_1 = B = (B +$$

$) = m_0$ and m_2 can be combined to yield,

$$f_2 = m_0 + m_2 = A = (A +$$

m_1 and m_3 can be combined to yield,

$$f_3 = m_1 + m_3 = B + AB = B(+) = B$$

m_2 and m_3 can be combined to yield,

$$f_4 = m_2 + m_3 = A + AB = A(B +) = A$$

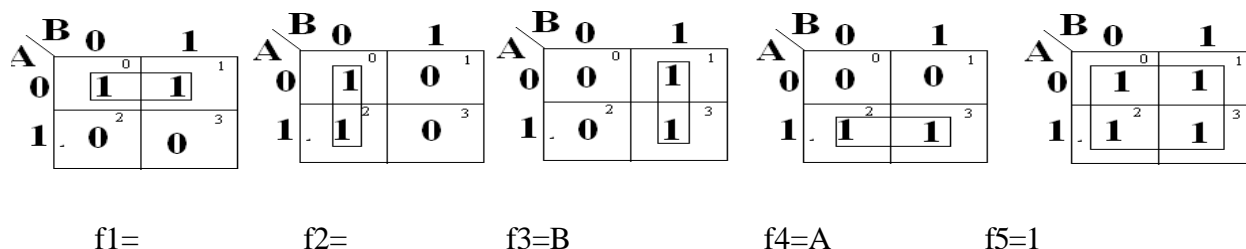
m_0, m_1, m_2 and m_3 can be combined to yield,

$$= + + A + AB$$

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

$$= + A$$

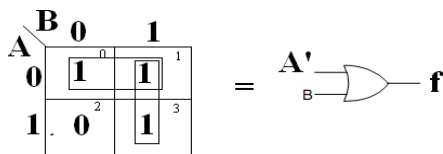
$$= 1$$



The possible minterm groupings in a two-variable k-map.

Two 2-squares adjacent to each other can be combined to form a 4-square. A 4-square eliminates 2 variables. A 4-square is called a quad. To read the squares on the map after minimization, consider only those variables which remain constant through the square, and ignore the variables which are varying. Write the non complemented variable if the variable is remaining constant as a 1, and the complemented variable if the variable is remaining constant as a 0, and write the variables as a product term. In the above figure f_1 read as , because, along the square , A remains constant as a 0, that is , as , where as B is changing from 0 to 1.

EX: Reduce the minterm $f = +A +AB$ using mapping Expressed in terms of minterms, the given expression is $F = m_0 + m_1 + m_2 + m_3 = m\sum(0,1,3)$ & the figure shows the k-map for f and its reduction . In one 2-square, A is constant as a 0 but B varies from a 0 to a 1, and in the other 2-square, B is constant as a 1 but A varies from a 0 to a 1. So, the reduced expressions is $+B$.



It requires two gate inputs for realization as

$f = +B$ (k-map in SOP form, and logic diagram.)

The main criterion in the design of a digital circuit is that its cost should be as low as possible. For that the expression used to realize that circuit must be minimal. Since the cost is proportional to number of gate inputs in the circuit, an expression is considered minimal only if it corresponds to the least possible number of gate inputs. & there is no guarantee for that k-map in SOP is the real minimal. To obtain real minimal expression, obtain the minimal expression both in SOP & POS form by using k-maps and take the minimal of these two minimals.

The 1's on the k-map indicate the presence of minterms in the output expressions, where as the 0s indicate the absence of minterms. Since the absence of a minterm in the SOP expression means the presence of the corresponding maxterm in the POS expression of the same. When a SOP expression is plotted on the k-map, 0s or no entries on the k-map represent the maxterms. To obtain the minimal expression in the POS form, consider the 0s on the k-map and follow the procedure used for combining 1s. Also, since the absence of a maxterm in the POS expression means the presence of the corresponding minterm in the SOP expression of the same, when a POS expression is plotted on the k-map, 1s or no entries on the k-map represent the minterms.

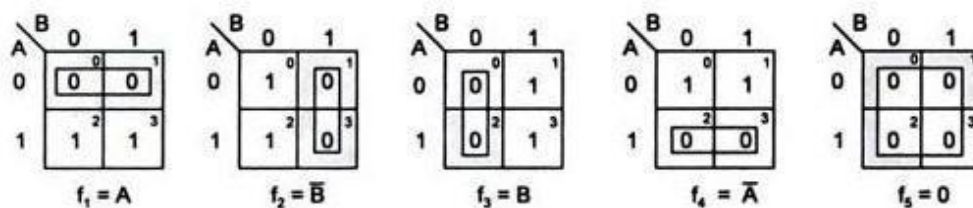
Mapping of POS expressions:

Each sum term in the standard POS expression is called a maxterm. A function in two variables (A, B) has four possible maxterms, $A+B$, $A+\bar{B}$, $\bar{A}+B$, $\bar{A}+\bar{B}$.

. They are represented as M_0 , M_1 , M_2 , and M_3 respectively. The uppercase letter M stands for maxterm and its subscript denotes the decimal designation of that maxterm obtained by treating the non-complemented variable as a 0 and the complemented variable as a 1 and putting them side by side for reading the decimal equivalent of the binary number so formed.

For mapping a POS expression on to the k-map, 0s are placed in the squares corresponding to the maxterms which are presented in the expression and 1s are placed in the squares corresponding to the maxterm which are not present in the expression. The decimal designation of the squares of the squares for maxterms is the same as that for the minterms. A two-variable k-map & the associated maxterms are as the maxterms of a two-variable k-map

The possible maxterm groupings in a two-variable k-map



Minimization of POS Expressions:

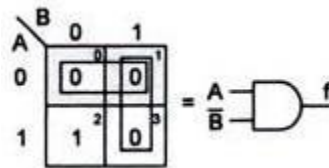
To obtain the minimal expression in POS form, map the given POS expression on to the K-map and combine the adjacent 0s into as large squares as possible. Read the squares putting the complemented variable if its value remains constant as a 1 and the non-complemented variable if its value remains constant as a 0 along the entire square (ignoring the variables which do not remain constant throughout the square) and then write them as a sum term.

Various maxterm combinations and the corresponding reduced expressions are shown in figure. In this f_1 read as A because A remains constant as a 0 throughout the square and B changes from a 0 to a 1. f_2 is read as B' because B remains constant along the square as a 1 and A changes from a 0 to a 1. f_3

Is read as a 0 because both the variables are changing along the square.

Ex: Reduce the expression $f = (A+B)(A+B')(A'+B')$ using mapping.

The given expression in terms of maxterms is $f = \pi M(0,1,3)$. It requires two gates inputs for realization of the reduced expression as



$$F = AB'$$

K-map in POS form and logic diagram

In this given expression ,the maxterm M_2 is absent. This is indicated by a 1 on the k-map. The corresponding SOP expression is $\sum m_2$ or AB' . This realization is the same as that for the POS form.

Three-variable K-map:

A function in three variables (A, B, C) expressed in the standard SOP form can have eight possible combinations: $A B C$, $A B C'$, $A B C$, $A B C'$, $A B C$, $A B C'$, $A B C$, and $A B C$. Each one of these combinations designate d by $m_0, m_1, m_2, m_3, m_4, m_5, m_6$, and m_7 , respectively, is called a minterm. A is the MSB of the minterm designator and C is the LSB.

In the standard POS form, the eight possible combinations are: $A+B+C$, $A+B+C'$, $A+B+C$, $A+B+C'$, $A+B+C$, $A+B+C'$, $A+B+C$, $A+B+C'$. Each one of these combinations designated by $M_0, M_1, M_2, M_3, M_4, M_5, M_6$, and M_7 respectively is called a maxterm. A is the MSB of the maxterm designator and C is the LSB.

A three-variable k-map has, therefore, $8(=2^3)$ squares or cells, and each square on the map represents a minterm or maxterm as shown in figure. The small number on the top right corner of each cell indicates the minterm or maxterm designation.

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

(a) Minterms

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

(b) Maxterms

The three-variable k-map.

The binary numbers along the top of the map indicate the condition of B and C for each column. The binary number along the left side of the map against each row indicates the condition of A for that row. For example, the binary number 01 on top of the second column in fig indicates that the variable B appears in complemented form and the variable C in non-complemented form in all the minterms in that column. The binary number 0 on the left of the first row indicates that the variable A appears in complemented form in all the minterms in that row, the binary numbers along the top of the k-map are not in normal binary order. They are, infact, in the Gray code. This is to ensure that twophysically adjacent squares are really adjacent, i.e., their minterms or maxterms differ by only one variable.

Ex: Map the expression $f = C + \dots + \dots + ABC$

In the given expression , the minterms are : $C=001=m_1$; $=101=m_5$;
 $=010=m_2$;

$=110=m_6$; $ABC=111=m_7$.

So the expression is $f = \sum m(1,5,2,6,7) = \sum m(1,2,5,6,7)$. The corresponding k-map is

A \ BC	00	01	11	10
	0^0	1^1	0^3	1^2
0	0	1	0	1
1	0^4	1^5	1^7	1^6

K-map in SOP form

Ex: Map the expression $f = (A+B+C)(\dots)(\dots)(A+\dots)(\dots)$

In the given expression the maxterms are
 $:A+B+C=000=M_0$; $\dots = 101=M_5$; $\dots = 111=M_7$; $A+\dots = 011=M_3$; \dots
 $=110=M_6$.

So the expression is $f = \pi M(0,5,7,3,6) = \pi M(0,3,5,6,7)$. The mapping of the expression is

A \ BC	00	01	11	10
	0	1	0	1
0	0 ⁰	1 ¹	0 ³	1 ²
1	1 ⁴	0 ⁵	0 ⁷	0 ⁶

K-map in POS form.

Minimization of SOP and POS expressions:

For reducing the Boolean expressions in SOP (POS) form plotted on the k-map, look at the 1s (0s) present on the map. These represent the minterms (maxterms). Look for the minterms (maxterms) adjacent to each other, in order to combine them into larger squares. Combining of adjacent squares in a k-map containing 1s (or 0s) for the purpose of simplification of a SOP (or POS) expression is called *looping*. Some of the minterms (maxterms) may have many adjacencies. Always start with the minterms (maxterm) with the least number of adjacencies and try to form as large as large a square as possible. The larger must form a geometric square or rectangle. They can be formed even by wrapping around, but cannot be formed by using diagonal configurations. Next consider the minterm (maxterm) with next to the least number of adjacencies and form as large a square as possible. Continue this till all the minterms (maxterms) are taken care of . A minterm (maxterm) can be part of any number of squares if it is helpful in reduction. Read the minimal expression from the k-map, corresponding to the squares formed. There can be more than one minimal expression.

Two squares are said to be adjacent to each other (since the binary designations along the top of the map and those along the left side of the map are in Gray code), if they are physically adjacent to each other, or can be made adjacent to each other by wrapping around. For squares to be combinable into bigger squares it is essential but not sufficient that their minterm designations must differ by a power of two.

General procedure to simplify the Boolean expressions:

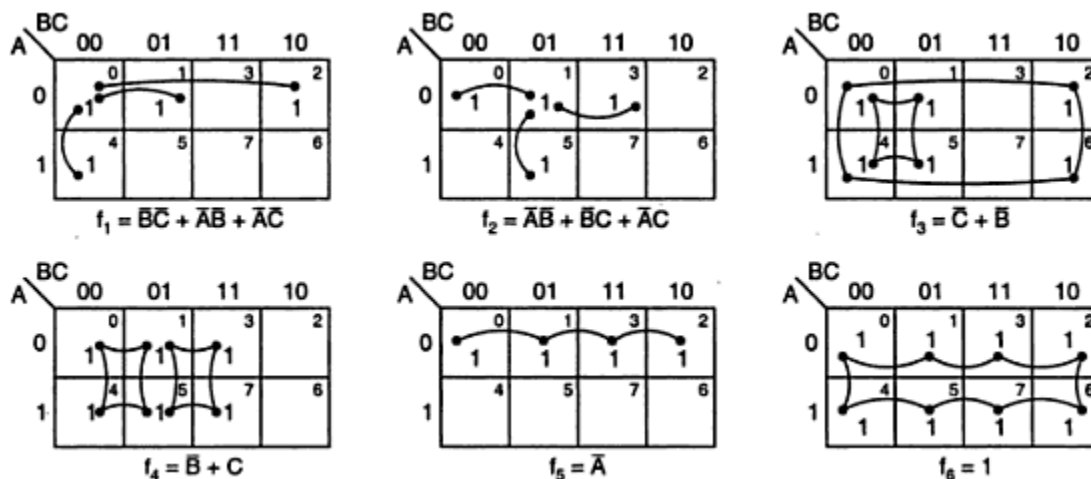
1. Plot the k-map and place 1s(0s) corresponding to the minterms (maxterms) of the SOP (POS) expression.
2. Check the k-map for 1s(0s) which are not adjacent to any other 1(0). They are isolated minterms(maxterms) . They are to be read as they are because they cannot be combined even into a 2-square.
3. Check for those 1s(0s) which are adjacent to only one other 1(0) and make them pairs (2 squares).
4. Check for quads (4 squares) and octets (8 squares) of adjacent 1s (0s) even if they contain some 1s(0s) which have already been combined. They must geometrically form a square or a rectangle.
5. Check for any 1s(0s) that have not been combined yet and combine them into bigger squares if possible.
6. Form the minimal expression by summing (multiplying) the product the product (sum) terms of all the groups.

Reading the K-maps:

While reading the reduced k-map in SOP (POS) form, the variable which remains constant as 0 along the square is written as the complemented (non-complemented) variable and the one which remains constant as 1 along the square is written as non-complemented (complemented) variable and the term as a product (sum) term. All the product (sum) terms are added (multiplied).

Some possible combinations of minterms and the corresponding minimal expressions read from the k-maps are shown in fig: Here f_6 is read as 1, because along the 8-square no variable remains constant. F_5 is read as \bar{A} , because, along the 4-square formed by m_0, m_1, m_2 and m_3 , the variables B and C are changing, and A remains constant as a 0. Algebraically,

$$\begin{aligned} f_5 &= m_0 + m_1 + m_2 + m_3 \\ &= \bar{A} + \bar{A}C + \bar{A}B + \bar{A}BC \\ &= \bar{A}(\bar{C} + B + C) \\ &= \bar{A}(B + 1) \\ &= \bar{A} \end{aligned}$$

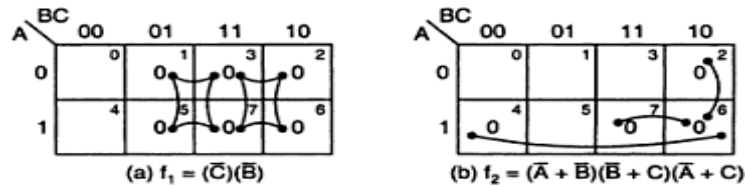


f_3 is read as $\bar{C} + B$, because in the 4-square formed by m_0, m_2, m_6 , and m_4 , the variable A and B are changing, whereas the variable C remains constant as a 0. So it is read as \bar{C} . In the 4-square formed by m_0, m_1, m_4, m_5 , A and C are changing but B remains constant as a 0. So it is read as B. So, the resultant expression for f_3 is the sum of these two, i.e., $\bar{C} + B$.

f_1 is read as $\bar{A}B + \bar{A}C$, because in the 2-square formed by m_0 and m_4 , A is changing from a 0 to a 1. Whereas B and C remain constant as a 0. So it is read as $\bar{A}B$. In the 2-square formed by m_0 and m_1 , C is changing from a 0 to a 1, whereas A and B remain constant as a 0. So it is read as $\bar{A}C$. In the 2-square formed by m_0 and m_2 , B is changing from a 0 to a 1 whereas A and C remain constant as a 0. So, it is read as $\bar{A}C$. Therefore, the resultant SOP expression is

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

Some possible maxterm groupings and the corresponding minimal POS expressions read from the k-map are



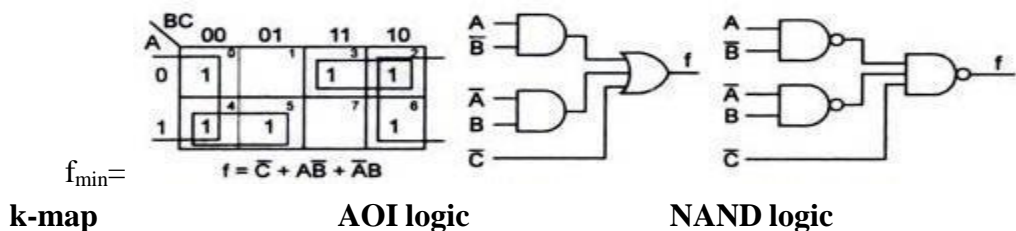
In this figure, along the 4-square formed by M_1, M_3, M_7, M_5 , A and B are changing from a 0 to a 1, where as C remains constant as a 1. SO it is read as AB . Along the 4-square formed by M_3, M_2, M_7 , and M_6 , variables A and C are changing from a 0 to a 1. But B remains constant as a 1. So it is read as BC . The minimal expression is the product of these two terms, i.e., $f_1 = (AB)(BC)$. also in this figure, along the 2-square formed by M_4 and M_6 , variable B is changing from a 0 to a 1, while variable A remains constant as a 1 and variable C remains constant as a 0. SO, read it as AC .

Similarly, the 2-square formed by M_7 and M_6 is read as AB , while the 2-square formed by M_2 and M_6 is read as BC . The minimal expression is the product of these sum terms, i.e., $f_2 = (AB + BC + AC)$.

Ex: Reduce the expression $f = \sum m(0, 2, 3, 4, 5, 6)$ using mapping and implement it in AOI logic as well as in NAND logic. The Sop k-map and its reduction, and the implementation of the minimal expression using AOI logic and the corresponding NAND logic are shown in figures below

In SOP k-map, the reduction is done as:

- m_5 has only one adjacency m_4 , so combine m_5 and m_4 into a square. Along this 2-square A remains constant as 1 and B remains constant as 0 but C varies from 0 to 1. So read it as AB .
- m_3 has only one adjacency m_2 , so combine m_3 and m_2 into a square. Along this 2-square A remains constant as 0 and B remains constant as 1 but C varies from 1 to 0. So read it as BC .
- m_6 can form a 2-square with m_2 and m_4 can form a 2-square with m_0 , but observe that by wrapping the map from left to right m_0, m_4, m_2, m_6 can form a 4-square. Out of these m_2 and m_4 have already been combined but they can be utilized again. So make it. Along this 4-square, A is changing from 0 to 1 and B is also changing from 0 to 1 but C is remaining constant as 0. so read it as \bar{C} .
- Write all the product terms in SOP form. So the minimal SOP expression is



Four variable k-maps:

Four variable k-map expressions can have $2^4=16$ possible combinations of input variables such as $\bar{A}\bar{B}\bar{C}\bar{D}$, $\bar{A}\bar{B}\bar{C}D$, $\bar{A}\bar{B}C\bar{D}$, $\bar{A}\bar{B}CD$ with minterm designations m_0, m_1, \dots, m_{15} respectively in SOP form & $A+B+C+D$, $A+B+C+\bar{D}$, $A+B+\bar{C}+D$, $A+B+\bar{C}+\bar{D}$ with maxterms M_0, M_1, \dots, M_{15} respectively in POS form. It has $2^4=16$ squares or cells. The binary number designations of rows & columns are in the gray code. Here follows 01 & 10 follows 11 called Adjacency ordering.

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

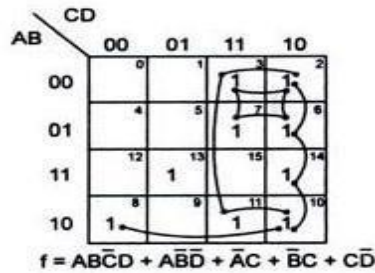
SOP form

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

POS form

EX: Reduce using mapping the expression $\Sigma m(2, 3, 6, 7, 8, 10, 11, 13, 14)$.

Start with the minterm with the least number of adjacencies. The minterm m_{13} has no adjacency. Keep it as it is. The m_8 has only one adjacency, m_{10} . Expand m_8 into a 2-square with m_{10} . The m_7 has two adjacencies, m_6 and m_3 . Hence m_7 can be expanded into a 4-square with m_6 , m_3 and m_2 . Observe that, m_7 , m_6 , m_2 , and m_3 form a geometric square. The m_{11} has 2 adjacencies, m_{10} and m_3 . Observe that, m_{11} , m_{10} , m_3 , and m_2 form a geometric square on wrapping the K-map. So expand m_{11} into a 4-square with m_{10} , m_3 and m_2 . Note that, m_2 and m_3 , have already become a part of the 4-square m_7 , m_6 , m_2 , and m_3 . But if m_{11} is expanded only into a 2-square with m_{10} , only one variable is eliminated. So m_2 and m_3 are used again to make another 4-square with m_{11} and m_{10} to eliminate two variables. Now only m_6 and m_{14} are left uncovered. They can form a 2-square that eliminates only one variable. Don't do that. See whether they can be expanded into a larger square. Observe that, m_2 , m_6 , m_{14} , and m_{10} form a rectangle. So m_6 and m_{14} can be expanded into a 4-square with m_2 and m_{10} . This eliminates two variables.



Five variable k-map:

Five variable k-map can have $2^5 = 32$ possible combinations of input variable as
 , E,-----ABCDE with minterms m_0, m_1, \dots, m_{31} respectively in SOP &
 $A+B+C+D+E, A+B+C+ \dots + \dots + \dots$ with maxterms M_0, M_1, \dots, M_{31} respectively in POS form. It has $2^5 = 32$ squares or cells of the k-map are divided into 2 blocks of

16 squares each. The left block represents minterms from m_0 to m_{15} in which A is a 0, and the right block represents minterms from m_{16} to m_{31} in which A is 1. The 5-variable k-map may contain 2-squares, 4-squares, 8-squares, 16-squares or 32-squares involving these two blocks. Squares are also considered adjacent in these two blocks, if when superimposing one block on top of another, the squares coincide with one another.

Some possible 2-squares in a five-variable map are $m_0, m_{16}; m_2, m_{18}; m_5, m_{21}; m_{15}, m_{31}; m_{11}, m_{27}$.

Some possible 4-squares are $m_0, m_2, m_{16}, m_{18}; m_0, m_1, m_{16}, m_{17}; m_0, m_4, m_{16}, m_{20}; m_{13}, m_{15}, m_{29}, m_{31}; m_5, m_{13}, m_{21}, m_{29}$.

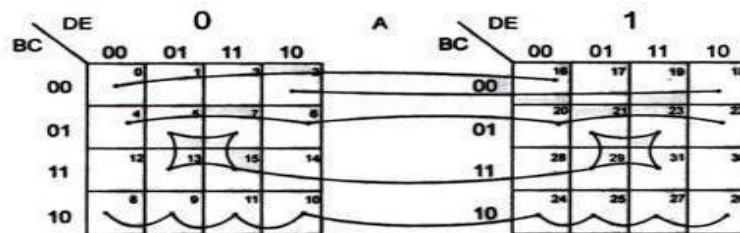
Some possible 8-squares are $m_0, m_1, m_3, m_2, m_{16}, m_{17}, m_{19}, m_{18}; m_0, m_4, m_{12}, m_8, m_{16}, m_{20}, m_{28}, m_{24}; m_5, m_7, m_{13}, m_{15}, m_{21}, m_{23}, m_{29}, m_{31}$.

The squares are read by dropping out the variables which change. Some possible

Grouping s is

- (a) $m_0, m_{16} = \overline{B}\overline{C}\overline{D}\overline{E}$
- (b) $m_2, m_{18} = \overline{B}\overline{C}D\overline{E}$
- (c) $m_4, m_6, m_{20}, m_{22} = \overline{B}C\overline{E}$
- (d) $m_5, m_7, m_{13}, m_{15}, m_{21}, m_{23}, m_{29}, m_{31} = CE$
- (e) $m_8, m_9, m_{10}, m_{11}, m_{24}, m_{25}, m_{26}, m_{27} = \overline{B}C$

- $M_0, M_{16} = B + C + D + E$
- $M_2, M_{18} = B + C + \overline{D} + E$
- $M_4, M_6, M_{20}, M_{22} = B + \overline{C} + E$
- $M_5, M_7, M_{13}, M_{15}, M_{21}, M_{23}, M_{29}, M_{31} = \overline{C} + \overline{E}$
- $M_8, M_9, M_{10}, M_{11}, M_{24}, M_{25}, M_{26}, M_{27} = \overline{B} + C$



Ex: $F = \sum m(0,1,4,5,6,13,14,15,22,24,25,28,29,30,31)$ is SOP

POS is $F = \pi M(2,3,7,8,9,10,11,12,16,17,18,19,20,21,23,26,27)$

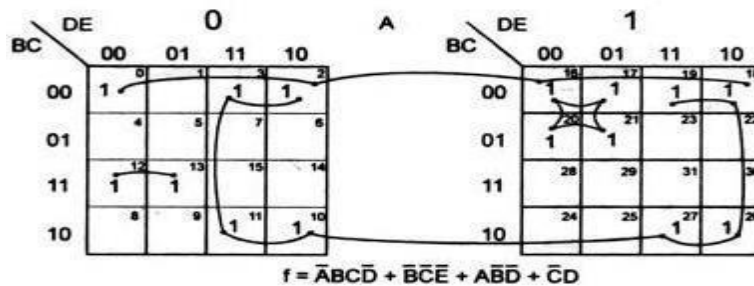
The real minimal expression is the minimal of the SOP and POS forms.

The reduction is done as

1. There is no isolated 1s
2. M_{12} can go only with m_{13} . Form a 2-square which is read as $A'B'CD'$
3. M_0 can go with m_2, m_{16} and m_{18} . so form a 4-square which is read as $B'C'E'$
4. M_{20}, m_{21}, m_{17} and m_{16} form a 4-square which is read as $AB'D'$
5. $M_2, m_3, m_{18}, m_{19}, m_{10}, m_{11}, m_{26}$ and m_{27} form an 8-square which is read as $C'D$
6. Write all the product terms in SOP form.

So the minimal expression is

$$F_{\min} = A'B'CD' + B'C'E' + AB'D' + C'D \text{ (16 inputs)}$$



In the POS k-map, the reduction is done as:

1. There are no isolated 0s

M_1 can go only with M_5 . So, make a 2-square, which is read as $(A + B + D + \bar{E})$.

3. M_4 can go with M_5, M_7 , and M_6 to form a 4-square, which is read as $(A + B + \bar{C})$.

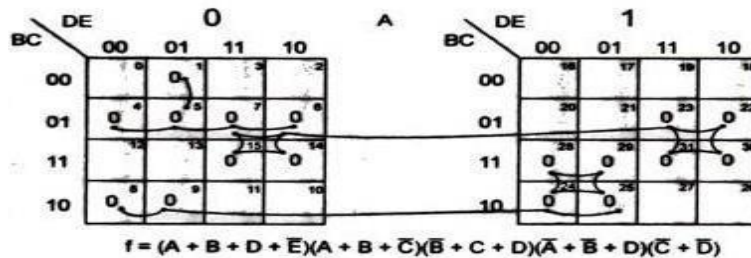
4. M_8

5. M_{28}

6. M_{30}

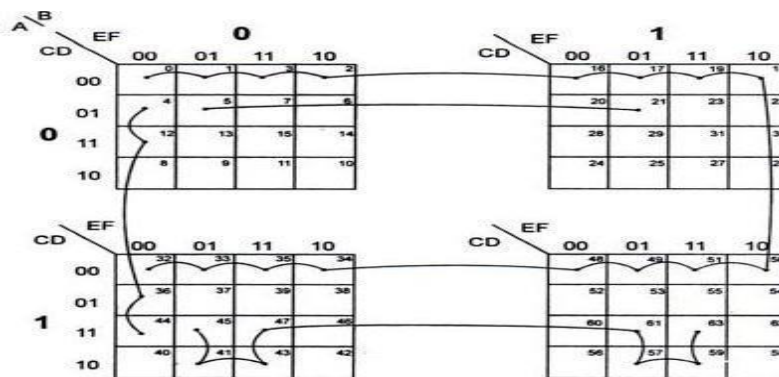
7. Sum terms in POS form. So the minimal expression in POS is

$$F_{\min} = A'BcD' + B'C'E' + AB'D' + C'D$$



Six variable k-map:

Six variable k-map can have $2^6 = 64$ combinations as $2^6 = 64$, -----
 ---ABCDEF with minterms m_0, m_1, \dots, m_{63} respectively in SOP & $(A+B+C+D+E+F)$, ----- (
 $+$ $+$ $+$ $+$ $+$ $+$) with maxterms M_0, M_1, \dots, M_{63} respectively in POS form. It has
 $2^6 = 64$ squares or cells of the k-map are divided into 4 blocks of 16 squares each.



Some possible groupings in a six variable k-map

Don't care combinations: For certain input combinations, the value of the output is unspecified either because the input combinations are invalid or because the precise value of the output is of no consequence. The combinations for which the value of experiments are not specified are called don't care combinations are invalid or because the precise value of the output is of no consequence. The combinations for which the value of expressions is not specified are called don't care combinations or Optional Combinations, such expressions stand incompletely specified. The output is a don't care for these invalid combinations.

Ex: In XS-3 code system, the binary states 0000, 0001, 0010, 1101, 1110, 1111 are unspecified. & never occur called don't cares.

A standard SOP expression with don't cares can be converted into a standard POS form by keeping the don't cares as they are & writing the missing minterms of the SOP form as the maxterms of the POS form viceversa.

Don't cares denoted by $_X'$ or $_\phi'$

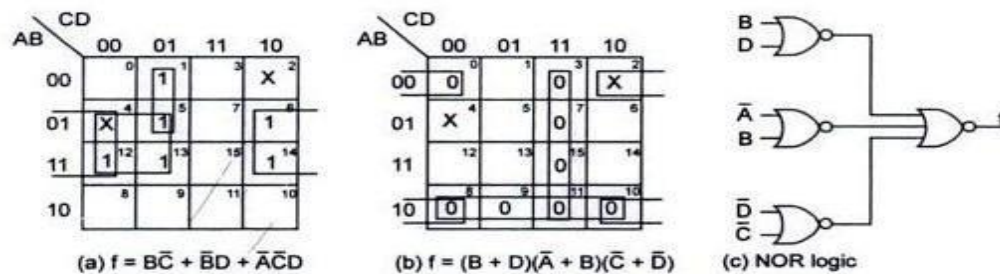
$$\text{Ex: } f = \sum m(1, 5, 6, 12, 13, 14) + d(2, 4)$$

$$\text{Or } f = \pi M(0, 3, 7, 9, 10, 11, 15) \cdot \pi d(2, 4)$$

$$\text{SOP minimal form } f_{\min} = \quad + B +$$

$$\text{POS minimal form } f_{\min} = (B + D)(\quad + B)(\quad + D)$$

$$= \quad + \quad + \quad + \quad + (\quad +$$



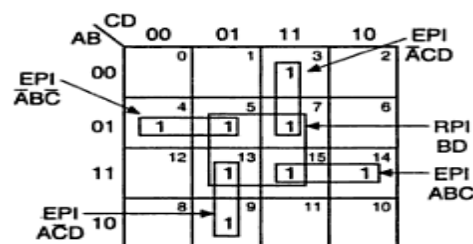
Prime implicants, Essential Prime implicants, Redundant prime implicants:

Each square or rectangle made up of the bunch of adjacent minterms is called a subcube. Each of these subcubes is called a Prime implicant (PI). The PI which contains at least one minterm which cannot be covered by any other prime implicants is called as Essential Prime implicant (EPI). The PI whose each 1 is covered at least by one EPI is called a Redundant Prime implicant (RPI). A PI which is neither an EPI nor a RPI is called a Selective Prime implicant (SPI).

The function has unique MSP comprising EPI is

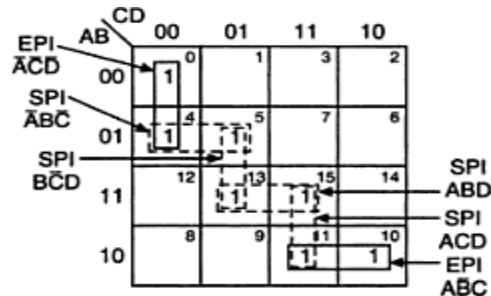
$$F(A, B, C, D) = CD + ABC + A\bar{D} + B$$

The RPI $\bar{B}D$ may be included without changing the function but the resulting expression would not be in minimal SOP(MSP) form.



Essential and Redundant Prime Implicants

$F(A,B,C,D)=\sum m(0,4,5,10,11,13,15)$ SPI are marked by dotted squares, shows MSP form of a function need not be unique.



Essential and Selective Prime Implicants

Here, the MSP form is obtained by including two EPI's & selecting a set of SPI's to cover remaining uncovered minterms 5,13,15. & these can be covered as

$$(A) (4,5) \& (13,15) \text{ ----- } B + ABD$$

$$(B) (5,13) \& (13,15) \text{ ----- } B D + ABD$$

$$(C) (5,13) \& (15,11) \text{ ----- } B D + ACD$$

$$F(A,B,C,D) = +A C \text{ ----- EPI's } + B + ABD$$

$$(OR) \quad F(A,B,C,D) = +A C \text{ ----- EPI's } + B D + ABD$$

$$(OR) \quad F(A,B,C,D) = +A C \text{ ----- EPI's } + B D + ACD$$

False PI's Essential False PI's, Redundant False PI's & Selective False PI's:

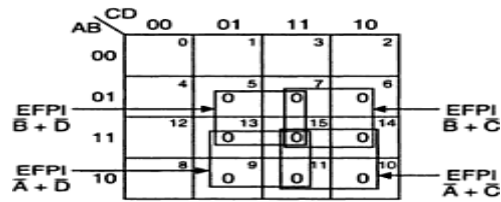
The maxterms are called false minterms. The PI's is obtained by using the maxterms are called False PI's (FPI). The FPI which contains at least one $_0'$ which can't be covered by only other FPI is called an Essential False Prime implicant (ESPI)

$$F(A,B,C,D) = \sum m(0,1,2,3,4,8,12)$$

$$= \pi M(5,6,7,9,10,11,13,14,15)$$

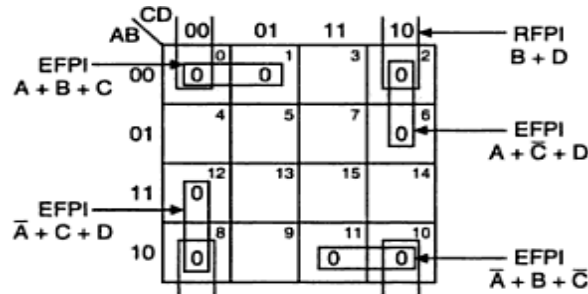
$$F_{min} = (+)(+)(+)(+)$$

All the FPI, EFPI's as each of them contain atleast one $_0'$ which can't be covered by any other FPI



Essential False Prime implicants

Consider Function $F(A,B,C,D) = \pi M(0,1,2,6,8,10,11,12)$



Essential and Redundant False Prime Implicants

Mapping when the function is not expressed in minterms (maxterms):

An expression in k-map must be available as a sum (product) of minterms (maxterms). However if not so expressed, it is not necessary to expand the expression algebraically into its minterms (maxterms). Instead, expansion into minterms (maxterms) can be accomplished in the process of entering the terms of the expression on the k-map.

Limitations of Karnaugh maps:

- Convenient as long as the number of variables does not exceed six.
- Manual technique, simplification process is heavily dependent on the human abilities.

Quine-Mccluskey Method:

It also known as *Tabular method*. It is more systematic method of minimizing expressions of even larger number of variables. It is suitable for hand computation as well as computation by machines i.e., programmable. . The procedure is based on repeated application of the combining theorem.

$PA+P=P$ (P is set of literals) on all adjacent pairs of terms, yields the set of all PI's from which a minimal sum may be selected.

Consider expression

$$\sum m(0,1,4,5) = +C+A +A C$$

First, second terms & third, fourth terms can be combined

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

Reduced to

$$(+ +) =$$

The same result can be obtained by combining m_0 & m_4 & m_1 & m_5 in first step & resulting terms in the second step.

Procedure:

- Decimal Representation
- Don't cares
- PI chart
- EPI
- Dominating Rows & Columns
- Determination of Minimal expressions in complex cases.

Branching Method:

EXAMPLE 3.29 Obtain the set of prime implicants for the Boolean expression
 $f = \sum m(0, 1, 6, 7, 8, 9, 13, 14, 15)$ using the tabular method.

Solution

Group the minterms in terms of the number of 1s present in them and write their binary designations. The procedure to obtain the prime implicants is shown in Table 3.3.

Table 3.3 Example 3.29

Column 1			Column 2			Column 3		
	Minterm	Binary designation		A	B	C	D	
Index 0	0	0000✓	0, 1 (1)	0	0	0	-	✓ 0, 1, 8, 9 (1, 8) - 00 - Q
Index 1	1	0001✓	0, 8 (8)	-	0	0	0	✓
	8	1000✓	1, 9 (8)	-	0	0	1	✓
Index 2	6	0110✓	8, 9 (1)	1	0	0	-	✓ 6, 7, 14, 15 (1, 8) - 11 - P
	9	1001✓	6, 7 (1)	0	1	1	-	✓
Index 3	7	0111✓	6, 14 (8)	-	1	1	0	✓
	13	1101✓	9, 13 (4)	1	-	0	1	S
	14	1110✓	7, 15 (8)	-	1	1	1	✓
Index 4	15	1111✓	13, 15 (2)	1	1	-	1	R
			14, 15 (1)	1	1	1	-	✓

Comparing the terms of index 0 with the terms of index 1 of column 1, $m_0(0000)$ is combined with $m_1(0001)$ to yield 0, 1 (1), i.e. 000–. This is recorded in column 2 and 0000 and 0001 are checked off in column 1. $m_0(0000)$ is combined with $m_8(1000)$ to yield 0, 8 (8), i.e. –000. This is recorded in column 2 and 1000 is checked off in column 1. Note that 0000 of column 1 has already been checked off. No more combinations of terms of index 0 and index 1 are possible. So, draw a line below the last combination of these groups, i.e. below 0, 8 (8), –000 in column 2. Now 0, 1 (1), i.e. 000– and 0, 8 (8), i.e. –000 are the terms in the first group of column 2.

Comparing the terms of index 1 with the terms of index 2 in column 1, $m_1(0001)$ is combined with $m_9(1001)$ to yield 1, 9 (8), i.e. –001. This is recorded in column 2 and 1001 is checked off in column 1 because 0001 has already been checked off. $m_8(1000)$ is combined with $m_9(1001)$ to yield 8, 9 (1), i.e. 100–. This is recorded in column 2. 1000 and 1001 of column 1 have already been checked off. So, no need to check them off again. No more combinations of terms of index 1 and index 2 are possible. So, draw a line below the last combination of these groups, i.e. 8, 9 (1),

–001 in column 2. Now 1, 9 (8), i.e. –001 and 8, 9 (1), i.e. 100– are the terms in the second group of column 2.

Similarly, comparing the terms of index 2 with the terms of index 3 in column 1,

$m_6(0110)$ and $m_7(0111)$ yield 6, 7 (1), i.e. 011–. Record it in column 2 and check off 6(0110) and 7(0111).

$m_6(0110)$ and $m_{14}(1110)$ yield 6, 14 (8), i.e. –110. Record it in column 2 and check off 6(0110) and 14(1110).

$m_9(1001)$ and $m_{13}(1101)$ yield 9, 13 (4), i.e. 1–01. Record it in column 2 and check off 9(1001) and 13(1101).

So, 6, 7 (1), i.e. 011–, and 6, 14 (8), i.e. –110 and 9, 13 (4), i.e. 1–01 are the terms in group 3 of column 2. Draw a line at the end of 9, 13 (4), i.e. 1–01.

Also, comparing the terms of index 3 with the terms of index 4 in column 1,

$m_7(0111)$ and $m_{15}(1111)$ yield 7, 15 (8), i.e. –111. Record it in column 2 and check off 7(0111) and 15(1111).

$m_{13}(1101)$ and $m_{15}(1111)$ yield 13, 15 (2), i.e. 11–1. Record it in column 2 and check off 13 and 15.

$m_{14}(1110)$ and $m_{15}(1111)$ yield 14, 15 (1), i.e. 111–. Record it in column 2 and check off 14 and 15.

So, 7, 15 (8), i.e. –111, and 13, 15 (2), i.e. 11–1 and 14, 15 (1), i.e. 111– are the terms in group 4 of column 2. Column 2 is completed now.

Comparing the terms of group 1 with the terms of group 2 in column 2, the terms 0, 1 (1), i.e. 000– and 8, 9 (1), i.e. 100– are combined to form 0, 1, 8, 9 (1, 8), i.e. –00–. Record it in group 1 of column 3 and check off 0, 1 (1), i.e. 000–, and 8, 9 (1), i.e. 100– of column 2. The terms 0, 8 (8), i.e. –000 and 1, 9 (8), i.e. –001 are combined to form 0, 1, 8, 9 (1, 8), i.e. –00–. This has already been recorded in column 3. So, no need to record again. Check off 0, 8 (8), i.e. –000 and 1, 9 (8), i.e. –001 of column 2. Draw a line below 0, 1, 8, 9 (1, 8), i.e. –00–. This is the only term in group 1 of column 3. No term of group 2 of column 2 can be combined with any term of group 3 of column 2. So, no entries are made in group 2 of column 2.

Comparing the terms of group 3 of column 2 with the terms of group 4 of column 2, the terms 6, 7 (1), i.e. 011–, and 14, 15 (1), i.e. 111– are combined to form 6, 7, 14, 15 (1, 8), i.e. –11–. Record it in group 3 of column 3 and check off 6, 7 (1), i.e. 011– and 14, 15 (1), i.e. 111– of column 2. The terms 6, 14 (8), i.e. –110 and 7, 15 (8), i.e. –111 are combined to form 6, 7, 14, 15 (1, 8), i.e. –11–. This has already been recorded in column 3; so, check off 6, 14 (8), i.e. –110 and 7, 15 (8), i.e. –111 of column 2.

Observe that the terms 9, 13 (4), i.e. 1–01 and 13, 15 (2), i.e. 11–1 cannot be combined with any other terms. Similarly in column 3, the terms 0, 1, 8, 9 (1, 8), i.e. –00– and 6, 7, 14, 15 (1, 8), i.e. –11– cannot also be combined with any other terms. So, these 4 terms are the prime implicants.

The terms, which cannot be combined further, are labelled as P, Q, R, and S. These form the set of prime implicants.

EX:

Obtain the minimal expression for $f = \sum m(1, 2, 3, 5, 6, 7, 8, 9, 12, 13, 15)$ using the tabular method.

Solution

The procedure to obtain the set of prime implicants is illustrated in Table 3.4.

Table 3.4 Example 3.30

Step 1		Step 2		Step 3	
Index 1	1 ✓	1, 3 (2) ✓	1, 3, 5, 7 (2, 4)	T	
	2 ✓	1, 5 (4) ✓	1, 5, 9, 13 (4, 8)	S	
	8 ✓	1, 9 (8) ✓	2, 3, 6, 7 (1, 4)	R	
Index 2	3 ✓	2, 3 (1) ✓	8, 9, 12, 13 (1, 4)	Q	
	5 ✓	2, 6 (4) ✓	5, 7, 13, 15 (2, 8)	P	
	6 ✓	8, 9 (1) ✓			
	9 ✓	8, 12 (4) ✓			
	12 ✓	3, 7 (4) ✓			
Index 3	7 ✓	5, 7 (2) ✓			
	13 ✓	5, 13 (8) ✓			
Index 4	15 ✓	6, 7 (1) ✓			
		9, 13 (4) ✓			
		12, 13 (1) ✓			
		7, 15 (8) ✓			
		13, 15 (2) ✓			

The non-combinable terms P, Q, R, S and T are recorded as prime implicants.

$$P \rightarrow 5, 7, 13, 15 (2, 8) = X 1 X 1 = BD$$

(Literals with weights 2 and 8, i.e. C and A are deleted. The lowest minterm is $m_5 (5 = 4 + 1)$. So, literals with weights 4 and 1, i.e. B and D are present in non-complemented form. So, read it as BD.)

$$Q \rightarrow 8, 9, 12, 13 (1, 4) = 1 X 0 X = A\bar{C}$$

(Literals with weights 1 and 4, i.e. D and B are deleted. The lowest minterm is m_8 . So, literal with weight 8 is present in non-complemented form and literal with weight 2 is present in complemented form. So, read it as $A\bar{C}$.)

$$R \rightarrow 2, 3, 6, 7 (1, 4) = 0 X 1 X = \bar{A}C$$

(Literals with weights 1 and 4, i.e. D and B are deleted. The lowest minterm is m_2 . So, literal with weight 2 is present in non-complemented form and literal with weight 8 is present in complemented form. So, read it as $\bar{A}C$.)

$$S \rightarrow 1, 5, 9, 13 (4, 8) = X X 0 1 = \bar{C}D$$

(Literals with weights 4 and 8, i.e. B and A are deleted. The lowest minterm is m_1 . So, literal with weight 1 is present in non-complemented form and literal with weight 2 is present in complemented form. So, read it as $\bar{C}D$.)

$$T \rightarrow 1, 3, 5, 7 (2, 4) = 0 X X 1 = \bar{A}D$$

(Literals with weights 2 and 4, i.e. C and B are deleted. The lowest minterm is 1. So, literal with weight 1 is present in non-complemented form and literal with weight 8 is present in complemented form. So, read it as $\bar{A}D$.)

The prime implicant chart of the expression

$$f = \sum m(1, 2, 3, 5, 6, 7, 8, 9, 12, 13, 15)$$

is as shown in Table 3.5. It consists of 11 columns corresponding to the number of minterms and 5 rows corresponding to the prime implicants P, Q, R, S, and T generated. Row R contains four \times s at the intersections with columns 2, 3, 6, and 7, because these minterms are covered by the prime implicant R. A row is said to cover the columns in which it has \times s. The problem now is to select a minimal subset of prime implicants, such that each column contains at least one \times in the rows corresponding to the selected subset and the total number of literals in the prime implicants selected is as small as possible. These requirements guarantee that the number of unions of the selected prime implicants is equal to the original number of minterms and that, no other expression containing fewer literals can be found.

Table 3.5 Example 3.30: Prime implicant chart

	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	1	2	3	5	6	7	8	9	12	13	15
*P $\rightarrow 5, 7, 13, 15 (2, 8)$				\times		\times				\times	\times
*Q $\rightarrow 8, 9, 12, 13 (1, 4)$							\times	\times	\times	\times	
*R $\rightarrow 2, 3, 6, 7 (1, 4)$		\times	\times		\times	\times					
S $\rightarrow 1, 5, 9, 13 (4, 8)$	\times			\times				\times		\times	
T $\rightarrow 1, 3, 5, 7 (2, 4)$	\times		\times	\times		\times					

In the prime implicant chart of Table 3.5, m_2 and m_6 are covered by R only. So, R is an essential prime implicant. So, check off all the minterms covered by it, i.e. m_2 , m_3 , m_6 , and m_7 . Q is also an essential prime implicant because only Q covers m_8 and m_{12} . Check off all the minterms covered by it, i.e. m_8 , m_9 , m_{12} , and m_{13} . P is also an essential prime implicant, because m_{15} is covered only by P. So check off m_{15} , m_5 , m_7 , and m_{13} covered by it. Thus, only minterm 1 is not covered. Either row S or row T can cover it and both have the same number of literals. Thus, two minimal expressions are possible.

$$P + Q + R + S = BD + A\bar{C} + \bar{A}C + \bar{C}D$$

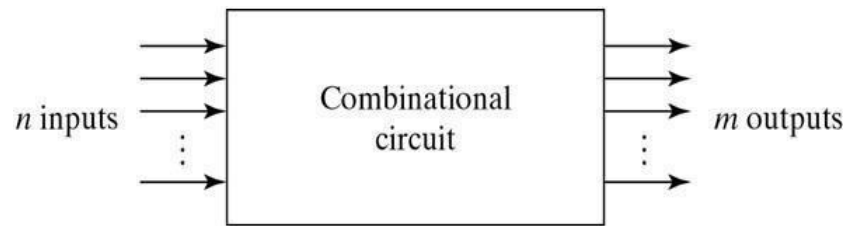
or

$$P + Q + R + T = BD + A\bar{C} + \bar{A}C + \bar{A}D$$

COMBINATIONAL CIRCUITS

Combinational Logic

- Logic circuits for digital systems may be combinational or sequential.
- A combinational circuit consists of input variables, logic gates, and output variables.



For n input variables, there are 2^n possible combinations of binary input variables. For each possible input combination, there is one and only one possible output combination. A combinational circuit can be described by m Boolean functions, one for each output variable. Usually the inputs come from flip-flops and outputs go to flip-flops.

Design Procedure:

1. The problem is stated
2. The number of available input variables and required output variables is determined.
3. The input and output variables are assigned letters/symbols.
4. The truth table that defines the required relationship between inputs and outputs is derived.
5. The simplified Boolean function for each output is obtained.

Adders:

Digital computers perform variety of information processing tasks, the one is arithmetic operations. And the most basic arithmetic operation is the addition of two binary digits. i.e., 4 basic possible operations are:

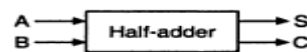
$$0+0=0, 0+1=1, 1+0=1, 1+1=10$$

The first three operations produce a sum whose length is one digit, but when augends and addend bits are equal to 1, the binary sum consists of two digits. The higher significant bit of this result is called a carry. A combinational circuit that performs the addition of two bits is called a half-adder. One that performs the addition of 3 bits (two significant bits & previous carry) is called a full adder. & 2 half adder can employ as a full-adder.

The Half Adder: A Half Adder is a combinational circuit with two binary inputs (augends and addend bits) and two binary outputs (sum and carry bits.) It adds the two inputs (A and B) and produces the sum (S) and the carry (C) bits. It is an arithmetic operation of addition of two single bit words.

Inputs		Outputs	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(a) Truth table



(b) Block diagram

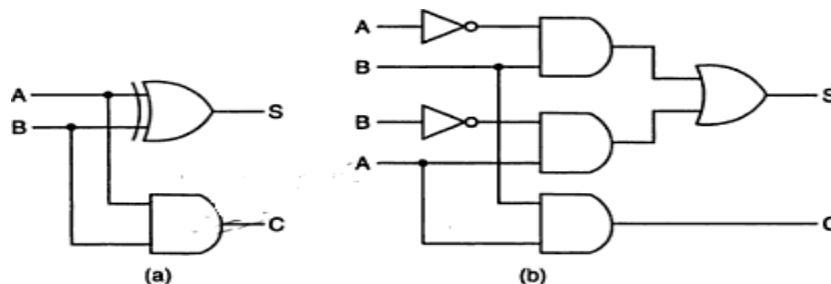
The Sum(S) bit and the carry (C) bit, according to the rules of binary addition, the sum (S) is the X-OR of A and B (It represents the LSB of the sum). Therefore,

$$S = A \oplus B$$

The carry (C) is the AND of A and B (it is 0 unless both the inputs are 1). Therefore,

$$C = AB$$

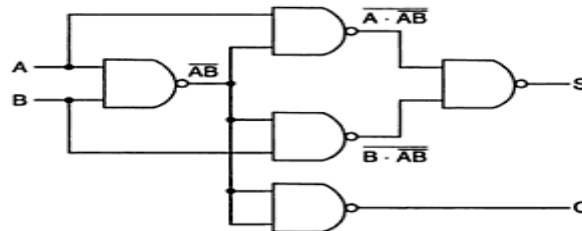
A half-adder can be realized by using one X-OR gate and one AND gate a



Logic diagrams of half-adder

NAND LOGIC:

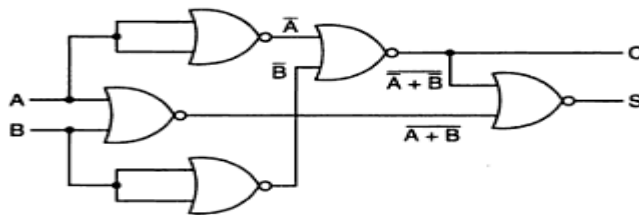
$$\begin{aligned}
 S &= A\bar{B} + \bar{A}B = A\bar{B} + A\bar{A} + \bar{A}B + B\bar{B} \\
 &= A(\bar{A} + \bar{B}) + B(\bar{A} + \bar{B}) \\
 &= A \cdot \bar{A}\bar{B} + B \cdot \bar{A}\bar{B} \\
 &= \overline{A \cdot \bar{A}\bar{B} \cdot B \cdot \bar{A}\bar{B}} \\
 C &= AB = \overline{\bar{A}\bar{B}}
 \end{aligned}$$



Logic diagram of a half-adder using only 2-input NAND gates.

NOR Logic:

$$\begin{aligned}
 S &= A\bar{B} + \bar{A}B = A\bar{B} + A\bar{A} + \bar{A}B + B\bar{B} \\
 &= A(\bar{A} + \bar{B}) + B(\bar{A} + \bar{B}) \\
 &= (A + B)(\bar{A} + \bar{B}) \\
 &= \overline{\overline{A + B} + \overline{\bar{A} + \bar{B}}} \\
 C &= AB = \overline{\bar{A}\bar{B}} = \overline{\bar{A} + \bar{B}}
 \end{aligned}$$



Logic diagram of a half-adder using only 2-input NOR gates.

The Full Adder:

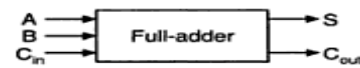
A Full-adder is a combinational circuit that adds two bits and a carry and outputs a sum bit and a carry bit. To add two binary numbers, each having two or more bits, the LSBs can be added by using a half-adder. The carry resulted from the addition of the LSBs is carried over to the next significant column and added to the two bits in that column. So, in the second and higher columns, the two data bits of that column and the carry bit generated from the addition in the previous column need to be added.

The full-adder adds the bits A and B and the carry from the previous column called the carry-in C_{in} and outputs the sum bit S and the carry bit called the carry-out C_{out} . The variable S gives the value of the least significant bit of the sum. The variable C_{out} gives the output carry. The

eight rows under the input variables designate all possible combinations of 1s and 0s that these variables may have. The 1s and 0s for the output variables are determined from the arithmetic sum of the input bits. When all the bits are 0s, the output is 0. The S output is equal to 1 when only 1 input is equal to 1 or when all the inputs are equal to 1. The C_{out} has a carry of 1 if two or three inputs are equal to 1.

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

(a) Truth table



(b) Block diagram

Full-adder.

From the truth table, a circuit that will produce the correct sum and carry bits in response to every possible combination of A,B and C_{in} is described by

$$S = \overline{A}\overline{B}C_{in} + \overline{A}B\overline{C_{in}} + A\overline{B}\overline{C_{in}} + \overline{A}BC_{in}$$

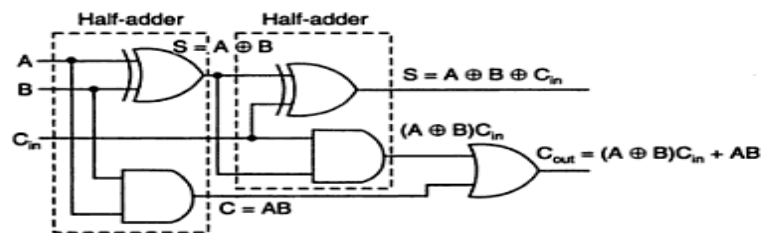
$$C_{out} = \overline{A}BC_{in} + A\overline{B}C_{in} + ABC_{in} + \overline{A}BC_{in}$$

and

$$S = A \oplus B \oplus C_{in}$$

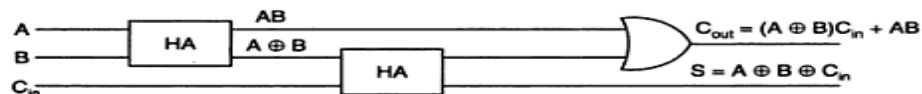
$$C_{out} = AC_{in} + BC_{in} + AB$$

The sum term of the full-adder is the X-OR of A,B, and C_{in} , i.e, the sum bit the modulo sum of the data bits in that column and the carry from the previous column. The logic diagram of the full-adder using two X-OR gates and two AND gates (i.e, Two half adders) and one OR gate is



Logic diagram of a full-adder using two half-adders.

The block diagram of a full-adder using two half-adders is :



Block diagram of a full-adder using two half-adders.

Even though a full-adder can be constructed using two half-adders, the disadvantage is that the bits must propagate through several gates in accession, which makes the total propagation delay greater than that of the full-adder circuit using AOI logic.

The Full-adder neither can also be realized using universal logic, i.e., either only NAND gates or only NOR gates as

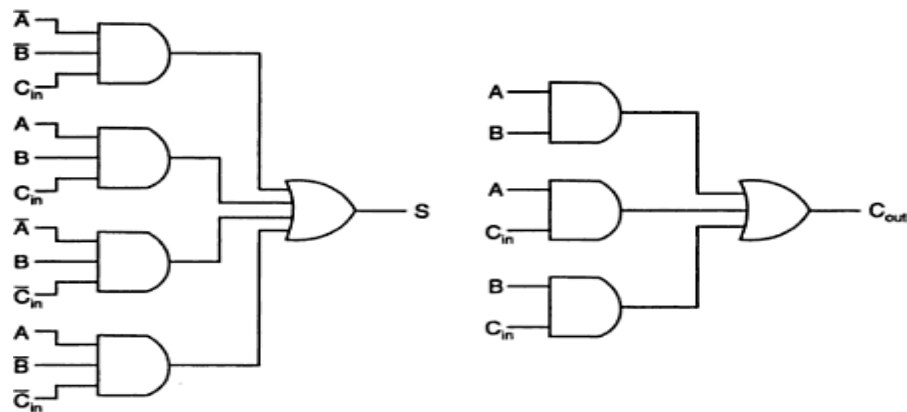
$$A \oplus B = \overline{\overline{A \cdot AB} \cdot \overline{B \cdot AB}}$$

Then

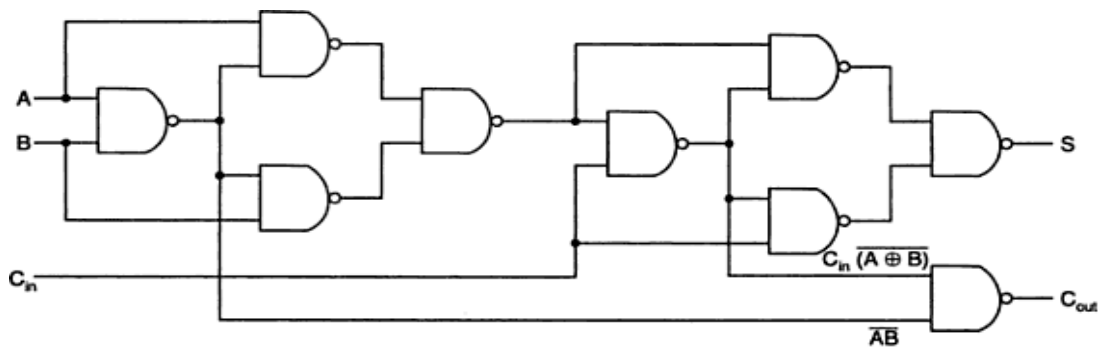
$$S = A \oplus B \oplus C_{in} = \overline{\overline{(A \oplus B) \cdot (A \oplus B)C_{in}} \cdot \overline{C_{in} \cdot (A \oplus B)C_{in}}}$$

NAND Logic:

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

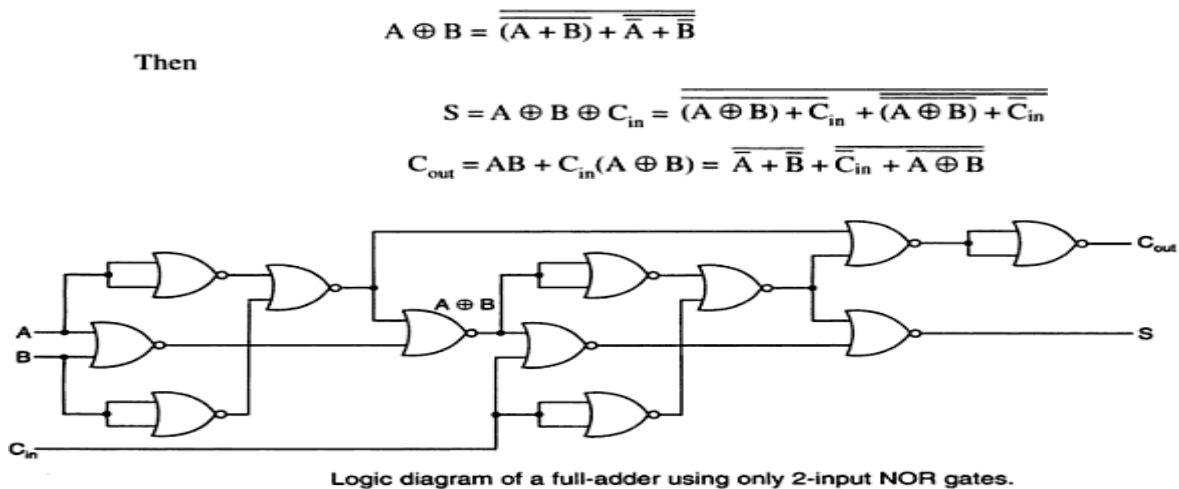


Sum and carry bits of a full-adder using AOI logic.



Logic diagram of a full-adder using only 2-input NAND gates.

NOR Logic:



Subtractors:

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend. By this, the subtraction operation becomes an addition operation and instead of having a separate circuit for subtraction, the adder itself can be used to perform subtraction. This results in reduction of hardware. In subtraction, each subtrahend bit of the number is subtracted from its corresponding significant minuend bit to form a difference bit. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position., that has been borrowed must be conveyed to the next higher pair of bits by means of a signal coming out (output) of a given stage and going into (input) the next higher stage.

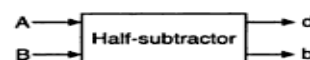
The Half-Subtractor:

A Half-subtractor is a combinational circuit that subtracts one bit from the other and produces the difference. It also has an output to specify if a 1 has been borrowed. . It is used to subtract the LSB of the subtrahend from the LSB of the minuend when one binary number is subtracted from the other.

A Half-subtractor is a combinational circuit with two inputs A and B and two outputs d and b. d indicates the difference and b is the output signal generated that informs the next stage that a 1 has been borrowed. When a bit B is subtracted from another bit A, a difference bit (d) and a borrow bit (b) result according to the rules given as

Inputs		Outputs	
A	B	d	b
0	0	0	0
1	0	1	0
1	1	0	0
0	1	1	1

(a) Truth table



(b) Block diagram

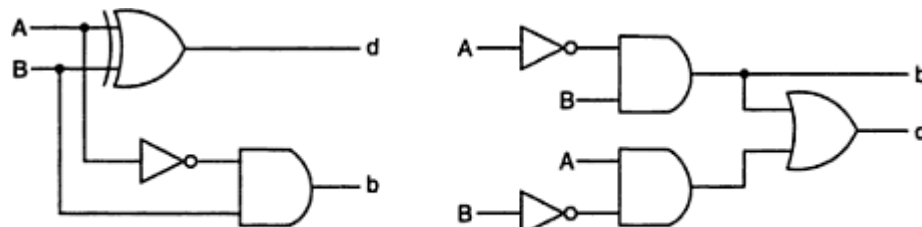
Half-subtractor.

The output borrow b is a 0 as long as $A \geq B$. It is a 1 for $A=0$ and $B=1$. The d output is the result of the arithmetic operation $2b + A - B$.

A circuit that produces the correct difference and borrow bits in response to every possible combination of the two 1-bit numbers is, therefore,

$$d = A \oplus B \text{ and } b = \bar{A}B$$

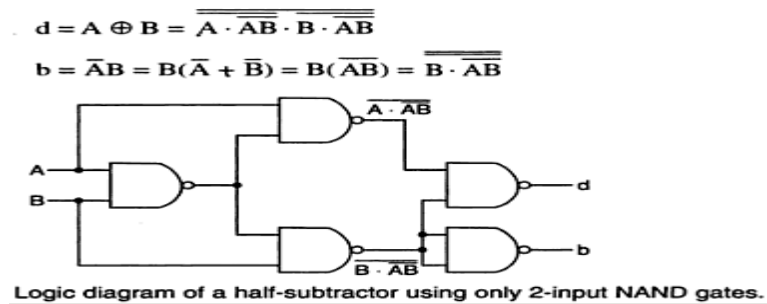
That is, the difference bit is obtained by X-OR ing the two inputs, and the borrow bit is obtained by ANDing the complement of the minuend with the subtrahend. Note that logic for this exactly the same as the logic for output S in the half-adder.



Logic diagrams of a half-subtractor.

A half-subtractor can also be realized using universal logic either using only NAND gates or using NOR gates as:

NAND Logic:

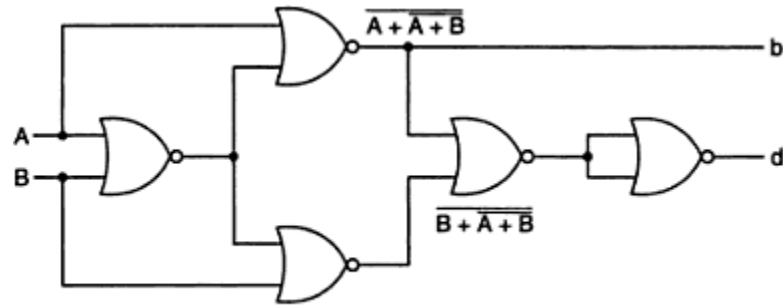


NOR Logic:

$$d = A \oplus B = A\bar{B} + \bar{A}B = A\bar{B} + B\bar{B} + \bar{A}B + A\bar{A}$$

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

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



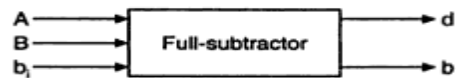
Logic diagram of a half-subtractor using only 2-input NOR gates.

The Full-Subtractor:

The half-subtractor can be only for LSB subtraction. IF there is a borrow during the subtraction of the LSBs, it affects the subtraction in the next higher column; the subtrahend bit is subtracted from the minuend bit, considering the borrow from that column used for the subtraction in the preceding column. Such a subtraction is performed by a full-subtractor. It subtracts one bit (B) from another bit (A), when already there is a borrow b_i from this column for the subtraction in the preceding column, and outputs the difference bit (d) and the borrow bit(b) required from the next d and b. The two outputs present the difference and output borrow. The 1s and 0s for the output variables are determined from the subtraction of $A-B-b_i$.

Inputs			Difference	Borrow
A	B	b_i	d	b
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

(a) Truth table



(b) Block diagram

Full-subtractor.

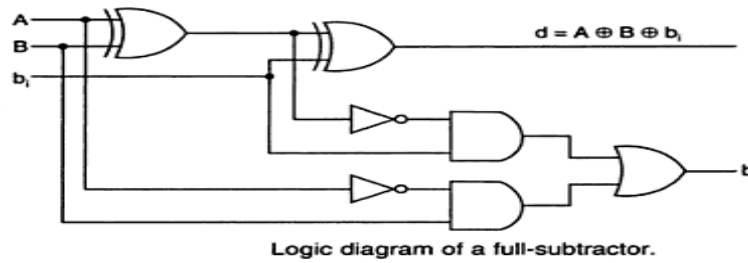
From the truth table, a circuit that will produce the correct difference and borrow bits in response to every possible combinations of A,B and b_i is

$$\begin{aligned}
 d &= \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + A\bar{B}\bar{b}_i + ABb_i \\
 &= b_i(AB + \bar{A}\bar{B}) + \bar{b}_i(A\bar{B} + \bar{A}B) \\
 &= b_i(\bar{A} \oplus B) + \bar{b}_i(A \oplus B) = A \oplus B \oplus b_i
 \end{aligned}$$

and

$$\begin{aligned}
 b &= \bar{A}\bar{B}b_i + \bar{A}B\bar{b}_i + \bar{A}Bb_i + ABb_i = \bar{A}B(b_i + \bar{b}_i) + (AB + \bar{A}\bar{B})b_i \\
 &= \bar{A}B + (A \oplus B)b_i
 \end{aligned}$$

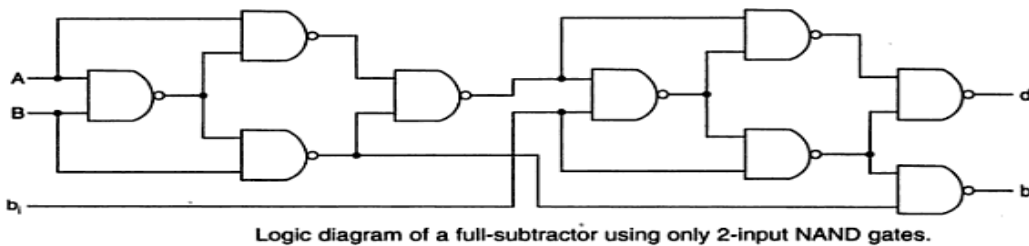
A full-subtractor can be realized using X-OR gates and AOI gates as



The full subtractor can also be realized using universal logic either using only NAND gates or using NOR gates as:

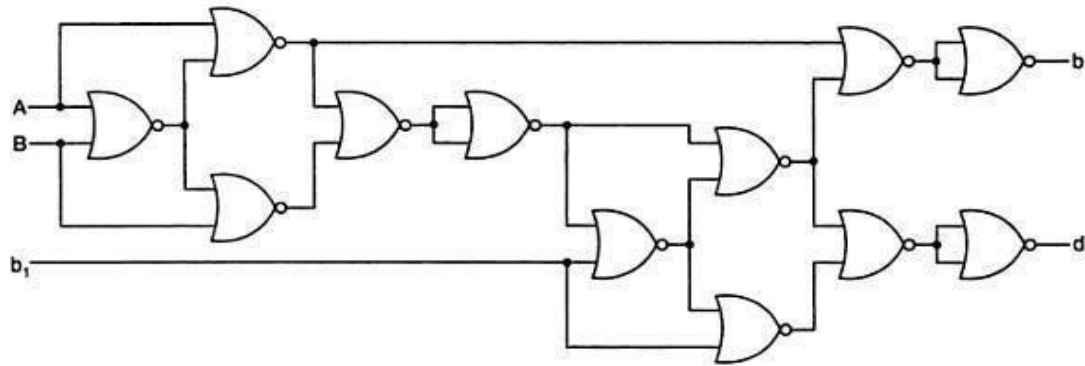
NAND Logic:

$$\begin{aligned}
 d &= A \oplus B \oplus b_i = \overline{\overline{(A \oplus B)} \oplus b_i} = \overline{\overline{(A \oplus B)}(\overline{A \oplus B})b_i \cdot b_i(\overline{A \oplus B})\overline{b_i}} \\
 b &= \overline{AB} + b_i(\overline{A \oplus B}) = \overline{\overline{AB} + b_i(\overline{A \oplus B})} \\
 &= \overline{\overline{AB} \cdot b_i(\overline{A \oplus B})} = \overline{B(\overline{A} + \overline{B}) \cdot b_i(b_i + (A \oplus B))} \\
 &= \overline{B \cdot AB \cdot b_i[b_i \cdot (A \oplus B)]}
 \end{aligned}$$



NOR Logic:

$$\begin{aligned}
 d &= A \oplus B \oplus b_i = \overline{\overline{(A \oplus B)} \oplus b_i} \\
 &= \overline{(A \oplus B)b_i + (\overline{A \oplus B})\overline{b_i}} \\
 &= \overline{[(A \oplus B) + (\overline{A \oplus B})\overline{b_i}][b_i + (\overline{A \oplus B})\overline{b_i}]} \\
 &= \overline{(A \oplus B) + (\overline{A \oplus B}) + b_i + b_i + (\overline{A \oplus B}) + b_i} \\
 &= \overline{(A \oplus B) + (\overline{A \oplus B}) + b_i + b_i + (\overline{A \oplus B}) + b_i} \\
 b &= \overline{AB} + b_i(\overline{A \oplus B}) \\
 &= \overline{\overline{A}(A + B) + (\overline{A \oplus B})[(A \oplus B) + b_i]} \\
 &= \overline{A + (A + B) + (\overline{A \oplus B}) + (\overline{A \oplus B}) + b_i}
 \end{aligned}$$

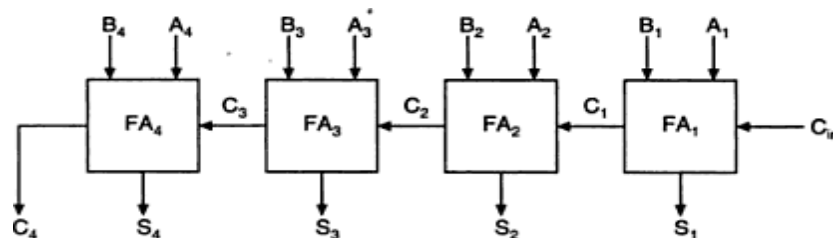


Logic diagram of a full subtractor using only 2-input NOR gates.

Binary Parallel Adder:

A binary parallel adder is a digital circuit that adds two binary numbers in parallel form and produces the arithmetic sum of those numbers in parallel form. It consists of full adders connected in a chain, with the output carry from each full-adder connected to the input carry of the next full-adder in the chain.

The interconnection of four full-adder (FA) circuits to provide a 4-bit parallel adder. The augends bits of A and addend bits of B are designated by subscript numbers from right to left, with subscript 1 denoting the lower-order bit. The carries are connected in a chain through the full-adders. The input carry to the adder is C_{in} and the output carry is C_4 . The S output generates the required sum bits. When the 4-bit full-adder circuit is enclosed within an IC package, it has four terminals for the augends bits, four terminals for the addend bits, four terminals for the sum bits, and two terminals for the input and output carries. An n-bit parallel adder requires n full adders. It can be constructed from 4-bit, 2-bit and 1-bit full adder ICs by cascading several packages. The output carry from one package must be connected to the input carry of the one with the next higher-order bits. The 4-bit full adder is a typical example of an MSI function.



Logic diagram of a 4-bit binary parallel adder.

Ripple carry adder:

In the parallel adder, the carry-out of each stage is connected to the carry-in of the next stage. The sum and carry-out bits of any stage cannot be produced, until sometime after the carry-in of that stage occurs. This is due to the propagation delays in the logic circuitry,

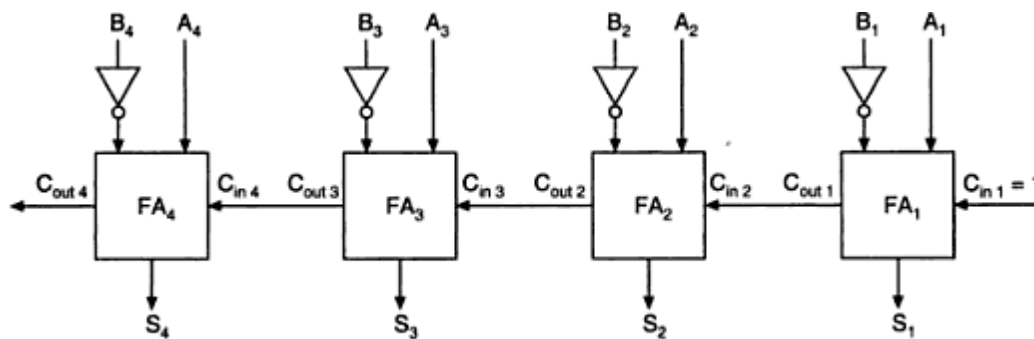
which lead to a time delay in the addition process. The carry propagation delay for each full-adder is the time between the application of the carry-in and the occurrence of the carry-out.

The 4-bit parallel adder, the sum (S_1) and carry-out (C_1) bits given by FA_1 are not valid, until after the propagation delay of FA_1 . Similarly, the sum S_2 and carry-out (C_2) bits given by FA_2 are not valid until after the cumulative propagation delay of two full adders (FA_1 and FA_2), and so on. At each stage, the sum bit is not valid until after the carry bits in all the preceding stages are valid. Carry bits must propagate or ripple through all stages before the most significant sum bit is valid. Thus, the total sum (the parallel output) is not valid until after the cumulative delay of all the adders.

The parallel adder in which the carry-out of each full-adder is the carry-in to the next most significant adder is called a ripple carry adder.. The greater the number of bits that a ripple carry adder must add, the greater the time required for it to perform a valid addition. If two numbers are added such that no carries occur between stages, then the add time is simply the propagation time through a single full-adder.

4- Bit ParallelSubtractor:

The subtraction of binary numbers can be carried out 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 1 to the least significant pair of bits. The 1's complement can be implemented with inverters as

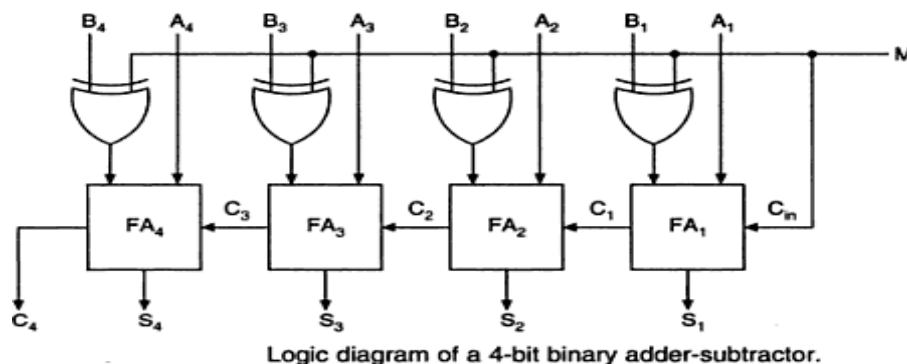


Logic diagram of a 4-bit parallel subtractor.

Binary-Adder Subtractor:

A 4-bit adder-subtractor, the addition and subtraction operations are combined into one circuit with one common binary adder. This is done by including an X-OR gate with each full-adder. The mode input M controls the operation. When $M=0$, the circuit is an adder, and when $M=1$, the circuit becomes a subtractor. Each X-OR gate receives input M and one of the inputs of B . When $M=0$, $B \oplus 1 = B$. The full-adder receives the value of B , the input carry is 0

and the circuit performs $A+B$, when $B \oplus 1 = B'$ and $C_1=1$. The B inputs are complemented and a 1 is through the input carry. The circuit performs the operation A plus the 2's complement of B.

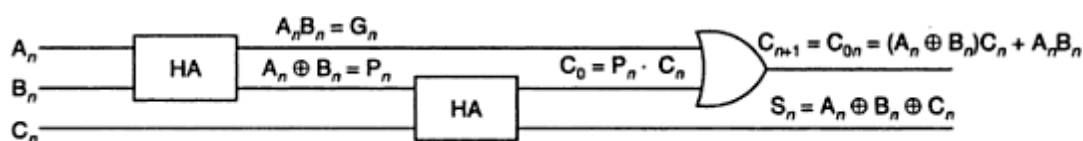


The Look-Ahead –Carry Adder:

In parallel-adder, the speed with which an addition can be performed is governed by the time required for the carries to propagate or ripple through all of the stages of the adder. The look-ahead carry adder speeds up the process by eliminating this ripple carry delay. It examines all the input bits simultaneously and also generates the carry-in bits for all the stages simultaneously.

The method of speeding up the addition process is based on the two additional functions of the full-adder, called the carry generate and carry propagate functions.

Consider one full adder stage; say the n th stage of a parallel adder as shown in fig. we know that is made by two half adders and that the half adder contains an X-OR gate to produce the sum and an AND gate to produce the carry. If both the bits A_n and B_n are 1s, a carry has to be generated in this stage regardless of whether the input carry C_{in} is a 0 or a 1. This is called generated carry, expressed as $G_n = A_n \cdot B_n$ which has to appear at the output through the OR gate as shown in fig.



A full adder (n th stage of a parallel adder).

There is another possibility of producing a carry out. X-OR gate inside the half-adder at the input produces an intermediary sum bit- call it P_n –which is expressed as $P_n = A_n \oplus B_n$. Next P_n and C_n are added using the X-OR gate inside the second half adder to produce the final



sum bit and $S_n = P_n \oplus C_n$ where $P_n = A_n \oplus B_n$ and output carry $C_0 = P_n \cdot C_n = (A_n \oplus B_n) \cdot C_n$ which becomes carry for the (n+1) th stage.

Consider the case of both P_n and C_n being 1. The input carry C_n has to be propagated to the output only if P_n is 1. If P_n is 0, even if C_n is 1, the and gate in the second half-adder will inhibit C_n . the carry out of the nth stage is 1 when either $G_n=1$ or $P_n \cdot C_n=1$ or both G_n and $P_n \cdot C_n$ are equal to 1.

For the final sum and carry outputs of the nth stage, we get the following Boolean expressions.

$$S_n = P_n \oplus C_n \text{ where } P_n = A_n \oplus B_n$$

$$C_{n+1} = G_n + P_n \cdot C_n \text{ where } G_n = A_n \cdot B_n$$

Observe the recursive nature of the expression for the output carry at the nth stage which becomes the input carry for the (n+1)st stage .it is possible to express the output carry of a higher significant stage is the carry-out of the previous stage.

Based on these , the expression for the carry-outs of various full adders are as follows,

$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

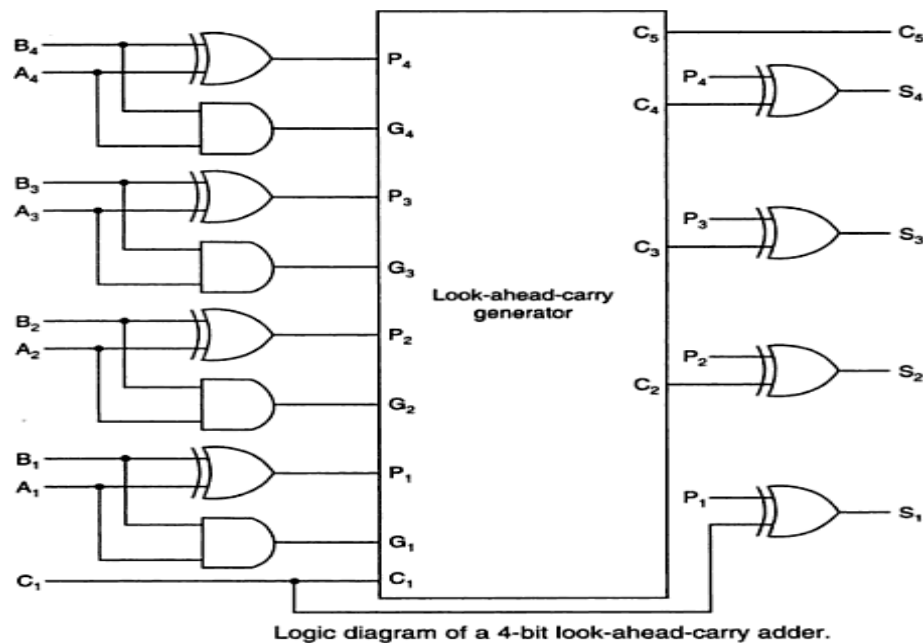
$$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

$$C_4 = G_3 + P_3 \cdot C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

The general expression for n stages designated as 0 through $(n - 1)$ would be

$$C_n = G_{n-1} + P_{n-1} \cdot C_{n-1} = G_{n-1} + P_{n-1} \cdot G_{n-2} + P_{n-1} \cdot P_{n-2} \cdot G_{n-3} + \dots + P_{n-1} \cdot \dots \cdot P_0 \cdot C_0$$

Observe that the final output carry is expressed as a function of the input variables in SOP form. Which is two level AND-OR or equivalent NAND-NAND form. Observe that the full look-ahead scheme requires the use of OR gate with (n+1) inputs and AND gates with number of inputs varying from 2 to (n+1).



2's complement Addition and Subtraction using Parallel Adders:

Most modern computers use the 2's complement system to represent negative numbers and to perform subtraction operations of signed numbers can be performed using only the addition operation, if we use the 2's complement form to represent negative numbers.

The circuit shown can perform both addition and subtraction in the 2's complement. This adder/subtractor circuit is controlled by the control signal ADD/SUB'. When the ADD/SUB' level is HIGH, the circuit performs the addition of the numbers stored in registers A and B. When the ADD/Sub' level is LOW, the circuit subtracts the number in register B from the number in register A. The operation is:

When ADD/SUB' is a 1:

1. AND gates 1,3,5 and 7 are enabled, allowing B_0, B_1, B_2 and B_3 to pass to the OR gates 9,10,11,12. AND gates 2,4,6 and 8 are disabled, blocking $B_0', B_1', B_2',$ and B_3' from reaching the OR gates 9,10,11 and 12.
2. The two levels B_0 to B_3 pass through the OR gates to the 4-bit parallel adder, to be added to the bits A_0 to A_3 . The sum appears at the output S_0 to S_3 .
3. Add/SUB' = 1 causes no carry into the adder.

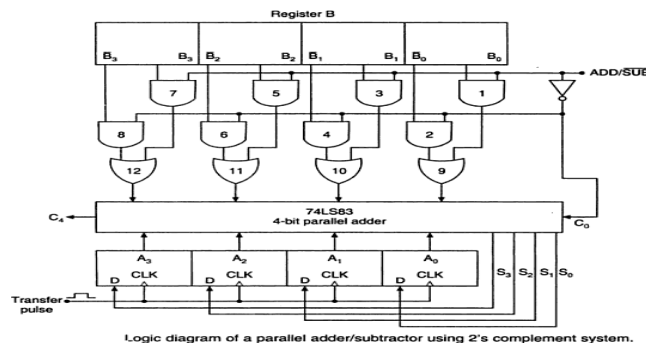
When ADD/SUB' is a 0:

1. AND gates 1,3,5 and 7 are disabled, allowing B_0, B_1, B_2 and B_3 from reaching the OR gates 9,10,11,12. AND gates 2,4,6 and 8 are enabled, blocking $B_0', B_1', B_2',$ and B_3' from reaching the OR gates.



2. The two levels B_0' to B_3' pass through the OR gates to the 4-bit parallel adder, to be added to the bits A_0 to A_3 . The C_0 is now 1. thus the number in register B is converted to its 2's complement form.
3. The difference appears at the output S_0 to S_3 .

Adders/Subtractors used for adding and subtracting signed binary numbers. In computers, the output is transferred into the register A (accumulator) so that the result of the addition or subtraction always end up stored in the register A. This is accomplished by applying a transfer pulse to the CLK inputs of register A.



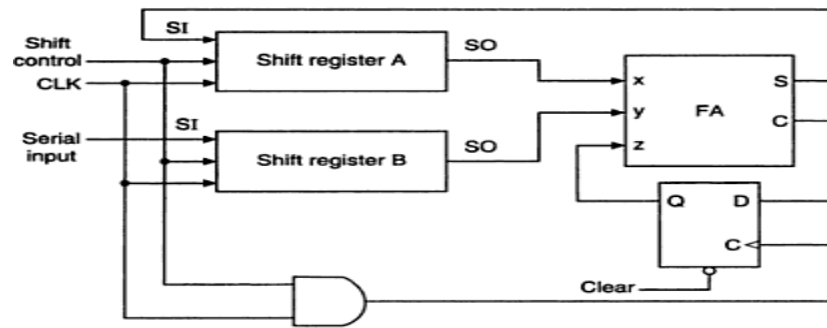
Serial Adder:

A serial adder is used to add binary numbers in serial form. The two binary numbers to be added serially are stored in two shift registers A and B. Bits are added one pair at a time through a single full adder (FA) circuit as shown. The carry out of the full-adder is transferred to a D flip-flop. The output of this flip-flop is then used as the carry input for the next pair of significant bits. The sum bit from the S output of the full-adder could be transferred to a third shift register. By shifting the sum into A while the bits of A are shifted out, it is possible to use one register for storing both augend and the sum bits. The serial input register B can be used to transfer a new binary number while the addend bits are shifted out during the addition.

The operation of the serial adder is:

Initially register A holds the augend, register B holds the addend and the carry flip-flop is cleared to 0. The outputs (SO) of A and B provide a pair of significant bits for the full-adder at x and y. The shift control enables both registers and carry flip-flop, so, at the clock pulse both registers are shifted once to the right, the sum bit from S enters the left most flip-flop of A, and the output carry is transferred into flip-flop Q. The shift control enables the registers for a number of clock pulses equal to the number of bits of the registers. For each succeeding clock pulse a new sum bit is transferred to A, a new carry is transferred to Q, and both registers are shifted once to the right. This process continues until the shift control is disabled. Thus the addition is accomplished by passing each pair of bits together with the previous carry through a single full adder circuit and transferring the sum, one bit at a time, into register A.

Initially, register A and the carry flip-flop are cleared to 0 and then the first number is added from B. While B is shifted through the full adder, a second number is transferred to it through its serial input. The second number is then added to the content of register A while a third number is transferred serially into register B. This can be repeated to form the addition of two, three, or more numbers and accumulate their sum in register A.



Logic diagram of a serial adder.

Difference between Serial and Parallel Adders:

The parallel adder registers with parallel load, whereas the serial adder uses shift registers. The number of full adder circuits in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full adder circuit and a carry flip-flop. Excluding the registers, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit. The sequential circuit in the serial adder consists of a full-adder and a flip-flop that stores the output carry.

BCD Adder:

The BCD addition process:

1. Add the 4-bit BCD code groups for each decimal digit position using ordinary binary addition.
2. For those positions where the sum is 9 or less, the sum is in proper BCD form and no correction is needed.
3. When the sum of two digits is greater than 9, a correction of 0110 should be added to that sum, to produce the proper BCD result. This will produce a carry to be added to the next decimal position.

A BCD adder circuit must be able to operate in accordance with the above steps. In other words, the circuit must be able to do the following:

1. Add two 4-bit BCD code groups, using straight binary addition.



- Determine, if the sum of this addition is greater than 1101 (decimal 9); if it is, add 0110 (decimal 6) to this sum and generate a carry to the next decimal position.

The first requirement is easily met by using a 4-bit binary parallel adder such as the 74LS83 IC. For example, if the two BCD code groups $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are applied to a 4-bit parallel adder, the adder will output $S_4S_3S_2S_1S_0$, where S_4 is actually C_4 , the carry-out of the MSB bits.

The sum outputs $S_4S_3S_2S_1S_0$ can range anywhere from 00000 to 100109 when both the BCD code groups are 1001=9. The circuitry for a BCD adder must include the logic needed to detect whenever the sum is greater than 01001, so that the correction can be added in. Those cases, where the sum is greater than 1001 are listed as:

S_4	S_3	S_2	S_1	S_0	Decimal number
0	1	0	1	0	10
0	1	0	1	1	11
0	1	1	0	0	12
0	1	1	0	1	13
0	1	1	1	0	14
0	1	1	1	1	15
1	0	0	0	0	16
1	0	0	0	1	17
1	0	0	1	0	18

Let us define a logic output X that will go HIGH only when the sum is greater than 01001 (i.e., for the cases in table). If examine these cases, see that X will be HIGH for either of the following conditions:

- Whenever $S_4=1$ (sum greater than 15)
- Whenever $S_3=1$ and either S_2 or S_1 or both are 1 (sum 10 to 15)

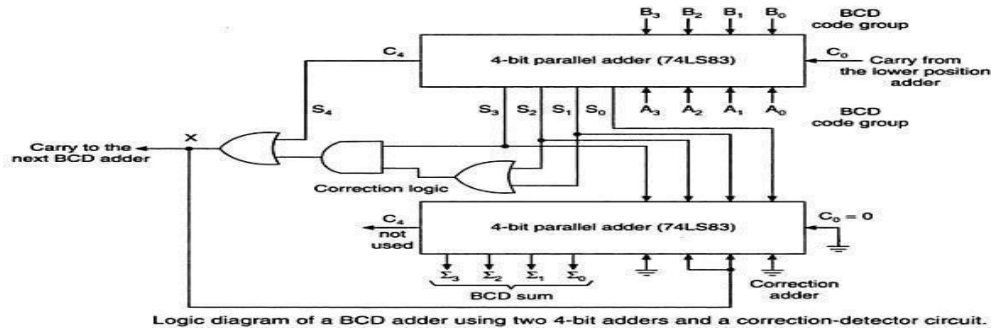
This condition can be expressed as

$$X = S_4 + S_3(S_2 + S_1)$$

Whenever $X=1$, it is necessary to add the correction factor 0110 to the sum bits, and to generate a carry. The circuit consists of three basic parts. The two BCD code groups $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$ are added together in the upper 4-bit adder, to produce the sum $S_4S_3S_2S_1S_0$. The logic gates shown implement the expression for X. The lower 4-bit adder will add the correction 0110 to the sum bits, only when $X=1$, producing the final BCD sum output represented by $\sum_3\sum_2\sum_1\sum_0$. The X is also the carry-out that is produced when the sum is greater than 01001. When $X=0$, there is no carry and no addition of 0110. In such cases, $\sum_3\sum_2\sum_1\sum_0 = S_3S_2S_1S_0$.



Two or more BCD adders can be connected in cascade when two or more digit decimal numbers are to be added. The carry-out of the first BCD adder is connected as the carry-in of the second BCD adder, the carry-out of the second BCD adder is connected as the carry-in of the third BCD adder and so on.



EXCESS-3(XS-3) ADDER:

To perform Excess-3 additions,

1. Add two xs-3 codegroups
2. If carry=1, add 0011(3) to the sum of those two codegroups

If carry =0, subtract 0011(3) i.e., add 1101 (13 in decimal) to the sum of those two code groups.

Ex: Add 9 and 5

	1100	9 in Xs-3
	+1000	5 in xs-3

1	0100	there is a carry
+0011	0011	add 3 to each group
-----	-----	
0100	0111	14 in xs-3
(1)	(4)	

EX:

(b)	0 1 1 1	4 in XS-3
	+ 0 1 1 0	3 in XS-3

	1 1 0 1	no carry
	+ 1 1 0 1	Subtract 3 (i.e. add 13)

Ignore carry	1 1 0 1 0	7 in XS-3
	(7)	

Implementation of xs-3 adder using 4-bit binary adders is shown. The augend ($A_3A_2A_1A_0$) and addend ($B_3B_2B_1B_0$) in xs-3 are added using the 4-bit parallel adder. If the carry is a 1, then 0011(3) is added to the sum bits $S_3S_2S_1S_0$ of the upper adder in the lower 4-bit parallel

adder. If the carry is a 0, then 1101(3) is added to the sum bits (This is equivalent to subtracting 0011(3) from the sum bits. The correct sum in xs-3 is obtained

Excess-3 (XS-3) Subtractor:

To perform Excess-3 subtraction,

1. Complement the subtrahend
2. Add the complemented subtrahend to the minuend.
3. If carry = 1, result is positive. Add 3 and end around carry to the result. If carry = 0, the result is negative. Subtract 3, i.e., and take the 1's complement of the result.

Ex: Perform 9-4

	1100	9 in xs-3
	+1000	Complement of 4 in Xs-3

(1)	0100	There is a carry
	+0011	Add 0011(3)

	0111	
	1	End around carry

	1000	5 in xs-3

The minuend and the 1's complement of the subtrahend in xs-3 are added in the upper 4-bit parallel adder. If the carry-out from the upper adder is a 0, then 1101 is added to the sum bits of the upper adder in the lower adder and the sum bits of the lower adder are complemented to get the result. If the carry-out from the upper adder is a 1, then 3=0011 is added to the sum bits of the lower adder and the sum bits of the lower adder give the result.

Binary Multipliers:

In binary multiplication by the paper and pencil method, is modified somewhat in digital machines because a binary adder can add only two binary numbers at a time.

In a binary multiplier, instead of adding all the partial products at the end, they are added two at a time and their sum accumulated in a register (the accumulator register). In addition, when the multiplier bit is a 0, 0s are not written down and added because it does not affect the final result. Instead, the multiplicand is shifted left by one bit.

The multiplication of 1110 by 1001 using this process is

Multiplicand 1110

Multiplier 1001

1110

The LSB of the multiplier is a 1; write down the multiplicand; shift the multiplicand one position to the left (1 1 1 0 0)

1110

The second multiplier bit is a 0; write down the previous result 1110; shift the multiplicand to the left again (1 1 1 0 0 0)

0 0)



+1110000

The fourth multiplier bit is a 1 write down the new multiplicand add it to the first partial product to obtain the final product.

1111110

This multiplication process can be performed by the serial multiplier circuit , which multiplies two 4-bit numbers to produce an 8-bit product. The circuit consists of following elements

X register: A 4-bit shift register that stores the multiplier --- it will shift right on the falling edge of the clock. Note that 0s are shifted in from the left.

B register: An 8-bit register that stores the multiplicand; it will shift left on the falling edge of the clock. Note that 0s are shifted in from the right.

A register: An 8-bit register, i.e, the accumulator that accumulates the partial products.

Adder: An 8-bit parallel adder that produces the sum of A and B registers. The adder outputs S_7 through S_0 are connected to the D inputs of the accumulator so that the sum can be transferred to the accumulator only when a clock pulse gets through the AND gate.

The circuit operation can be described by going through each step in the multiplication of 1110 by 1001. The complete process requires 4 clock cycles.

1. Before the first clock pulse: Prior to the occurrence of the first clock pulse, the register A is loaded with 00000000, the register B with the multiplicand 00001110, and the register X with the multiplier 1001. Assume that each of these registers is loaded using its asynchronous inputs(i.e., PRESET and CLEAR). The output of the adder will be the sum of A and B,i.e., 00001110.

2 First Clock pulse: Since the LSB of the multiplier (X_0) is a 1, the first clock pulse gets through the AND gate and its positive going transition transfers the sum outputs into the accumulator. The subsequent negative going transition causes the X and B registers to shift right and left, respectively. This produces a new sum of A and B.

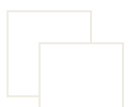
3 Second Clock Pulse: The second bit of the original multiplier is now in X_0 . Since this bit is a 0, the second clock pulse is inhibited from reaching the accumulator. Thus, the sum outputs are not transferred into the accumulator and the number in the accumulator does not change. The negative going transition of the clock pulse will again shift the X and B registers. Again a new sum is produced.

4 Third Clock Pulse: The third bit of the original multiplier is now in X_0 ; since this bit is a 0, the third clock pulse is inhibited from reaching the accumulator. Thus, the sum outputs are not transferred into the accumulator and the number in the accumulator does not change. The negative going transition of the clock pulse will again shift the X and B registers. Again a new sum is produced.

5 Fourth Clock Pulse: The last bit of the original multiplier is now in X_0 , and since it is a 1, the positive going transition of the fourth pulse transfers the sum into the accumulator. The accumulator now holds the final product. The negative going transition of the clock pulse shifts X and B again. Note that, X is now 0000, since all the multiplier bits have been shifted out.

Code converters:

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be



inserted between the two systems if each uses different codes for the same information. Thus a code converter is a logic circuit whose inputs are bit patterns representing numbers (or character) in one code and whose outputs are the corresponding representation in a different code. Code converters are usually multiple output circuits.

To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates.

For example, a binary –to-gray code converter has four binary input lines B_4, B_3, B_2, B_1 and four gray code output lines G_4, G_3, G_2, G_1 . When the input is 0010, for instance, the output should be 0011 and so forth. To design a code converter, we use a code table treating it as a truth table to express each output as a Boolean algebraic function of all the inputs.

In this example, of binary –to-gray code conversion, we can treat the binary to the gray code table as four truth tables to derive expressions for G_4, G_3, G_2 , and G_1 . Each of these four expressions would, in general, contain all the four input variables B_4, B_3, B_2 , and B_1 . Thus, this code converter is actually equivalent to four logic circuits, one for each of the truth tables.

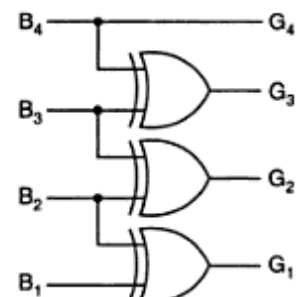
The logic expression derived for the code converter can be simplified using the usual techniques, including ‘don’t cares’ if present. Even if the input is an unweighted code, the same cell numbering method which we used earlier can be used, but the cell numbers --must correspond to the input combinations as if they were an 8-4-2-1 weighted code. s

Design of a 4-bit binary to gray code converter:

$$\begin{aligned} G_4 &= \Sigma m(8, 9, 10, 11, 12, 13, 14, 15) & G_4 &= B_4 \\ G_3 &= \Sigma m(4, 5, 6, 7, 8, 9, 10, 11) & G_3 &= \bar{B}_4 B_3 + B_4 \bar{B}_3 = B_4 \oplus B_3 \\ G_2 &= \Sigma m(2, 3, 4, 5, 10, 11, 12, 13) & G_2 &= \bar{B}_3 B_2 + B_3 \bar{B}_2 = B_3 \oplus B_2 \\ G_1 &= \Sigma m(1, 2, 5, 6, 9, 10, 13, 14) & G_1 &= \bar{B}_2 B_1 + B_2 \bar{B}_1 = B_2 \oplus B_1 \end{aligned}$$

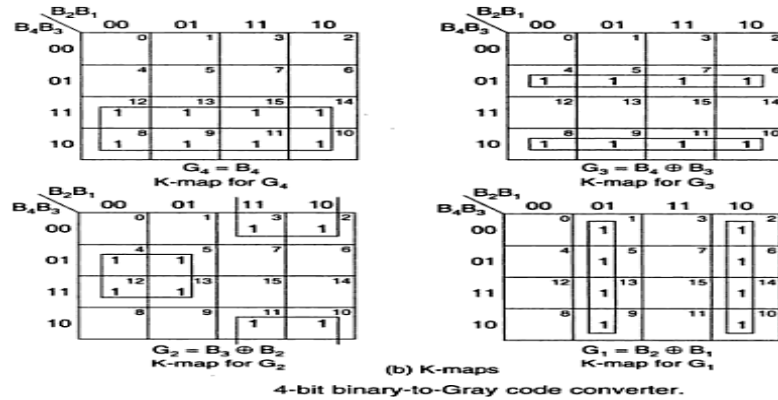
4-bit binary				4-bit Gray			
B_4	B_3	B_2	B_1	G_4	G_3	G_2	G_1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

(a) Conversion table



(c) Logic diagram

4-bit binary-to-Gray code converter



Design of a 4-bit gray to Binary code converter:

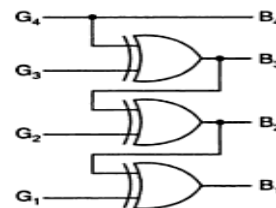
$$\begin{aligned}
 B_4 &= \Sigma m(12, 13, 15, 14, 10, 11, 9, 8) = \Sigma m(8, 9, 10, 11, 12, 13, 14, 15) \\
 B_3 &= \Sigma m(6, 7, 5, 4, 10, 11, 9, 8) = \Sigma m(4, 5, 6, 7, 8, 9, 10, 11) \\
 B_2 &= \Sigma m(3, 2, 5, 4, 15, 14, 9, 8) = \Sigma m(2, 3, 4, 5, 8, 9, 14, 15) \\
 B_1 &= \Sigma m(1, 2, 7, 4, 13, 14, 11, 8) = \Sigma m(1, 2, 4, 7, 8, 11, 13, 14)
 \end{aligned}$$

$$\begin{aligned}
 B_4 &= G_4 \\
 B_3 &= \bar{G}_4 G_3 + G_4 \bar{G}_3 = G_4 \oplus G_3 \\
 B_2 &= \bar{G}_4 G_3 \bar{G}_2 + \bar{G}_4 \bar{G}_3 G_2 + G_4 \bar{G}_3 \bar{G}_2 + G_4 G_3 G_2 \\
 &= \bar{G}_4 (G_3 \oplus G_2) + G_4 (\bar{G}_3 \oplus \bar{G}_2) = G_4 \oplus G_3 \oplus G_2 = B_3 \oplus G_2 \\
 B_1 &= \bar{G}_4 \bar{G}_3 \bar{G}_2 G_1 + \bar{G}_4 \bar{G}_3 G_2 \bar{G}_1 + \bar{G}_4 G_3 G_2 G_1 + \bar{G}_4 G_3 \bar{G}_2 \bar{G}_1 + G_4 G_3 \bar{G}_2 G_1 \\
 &\quad + G_4 G_3 G_2 \bar{G}_1 + G_4 \bar{G}_3 G_2 G_1 + G_4 \bar{G}_3 \bar{G}_2 \bar{G}_1
 \end{aligned}$$

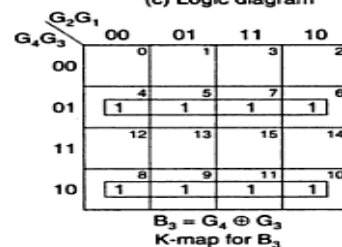
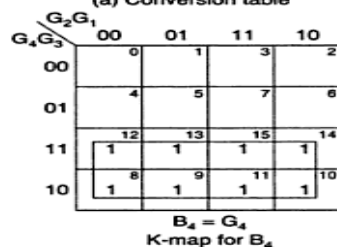
$$\begin{aligned}
 &= \bar{G}_4 \bar{G}_3 (G_2 \oplus G_1) + G_4 G_3 (G_2 \oplus G_1) + \bar{G}_4 G_3 (\bar{G}_2 \oplus \bar{G}_1) + G_4 \bar{G}_3 (\bar{G}_2 \oplus \bar{G}_1) \\
 &= (G_2 \oplus G_1) (\bar{G}_4 \oplus G_3) + (\bar{G}_2 \oplus \bar{G}_1) (G_4 \oplus G_3) \\
 &= G_4 \oplus G_3 \oplus G_2 \oplus G_1
 \end{aligned}$$

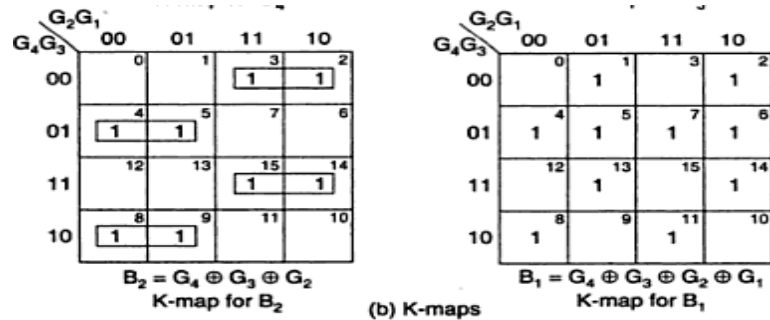
4-bit Gray				4-bit binary			
G_4	G_3	G_2	G_1	B_4	B_3	B_2	B_1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	0
0	1	1	0	0	0	1	1
0	1	1	1	0	1	0	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	0	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	0	1	1
1	1	1	1	1	0	1	0
1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	0
1	0	0	0	1	1	1	1

(a) Conversion table



(c) Logic diagram





Design of a 4-bit BCD to XS-3 code converter:

8421 code				XS-3 code			
B ₄	B ₃	B ₂	B ₁	X ₄	X ₃	X ₂	X ₁
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	1	0
0	0	1	1	0	1	1	1
0	1	0	0	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

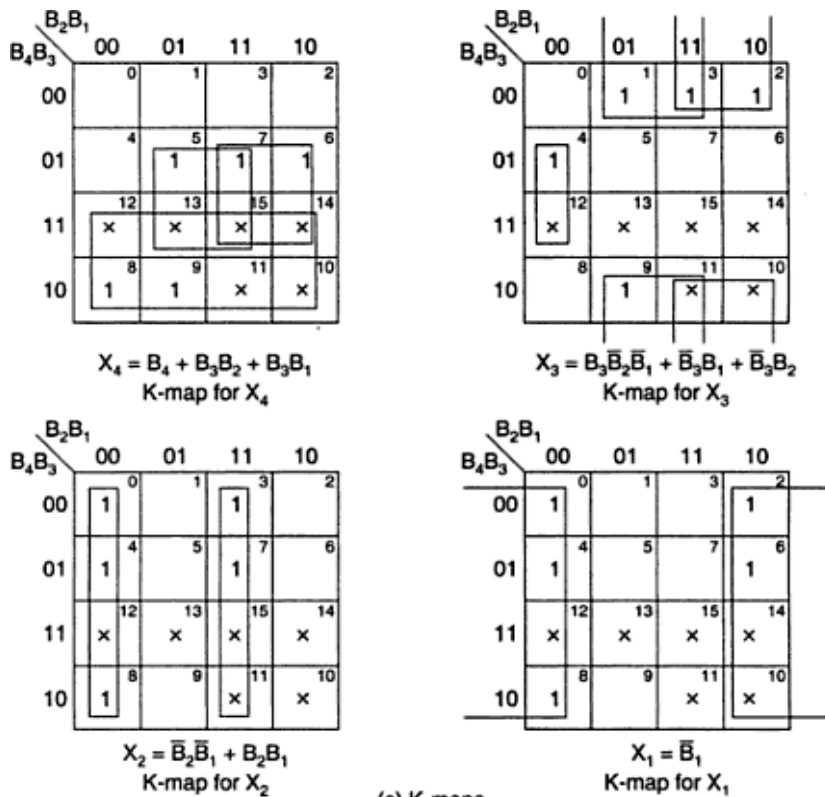
(a) Conversion table

$$\begin{aligned}
 X_4 &= \sum m(5, 6, 7, 8, 9) + d(10, 11, 12, 13, 14, 15) \\
 X_3 &= \sum m(1, 2, 3, 4, 9) + d(10, 11, 12, 13, 14, 15) \\
 X_2 &= \sum m(0, 3, 4, 7, 8) + d(10, 11, 12, 13, 14, 15) \\
 X_1 &= \sum m(0, 2, 4, 6, 8) + d(10, 11, 12, 13, 14, 15)
 \end{aligned}$$

The minimal expressions are
 $X_4 = B_4 + B_3B_2 + B_3B_1$
 $X_3 = B_3\bar{B}_2\bar{B}_1 + \bar{B}_3B_1 + \bar{B}_3B_2$
 $X_2 = \bar{B}_2\bar{B}_1 + B_2B_1$
 $X_1 = \bar{B}_1$

(b) Minimal expressions

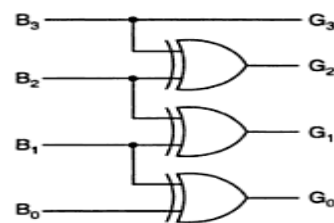
4-bit BCD-to-XS-3 code converter



Design of a BCD to gray code converter:

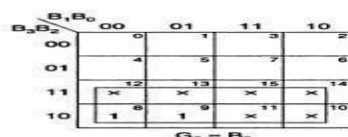
BCD code				Gray code			
B ₃	B ₂	B ₁	B ₀	G ₃	G ₂	G ₁	G ₀
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

(a) BCD-to-Gray code conversion table

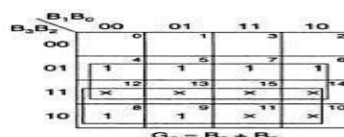


(b) Logic diagram

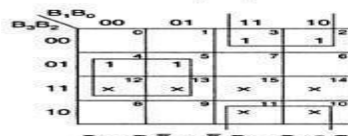
BCD-to-Gray code converter.



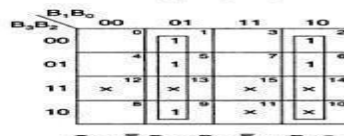
$$G_3 = B_3$$



$$G_2 = B_2 + B_3$$



$$G_1 = B_2B_3 + B_2B_1 = B_2 \oplus B_1$$



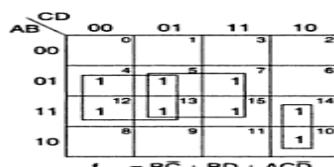
$$G_0 = B_1B_3 + B_1B_0 = B_1 \oplus B_0$$

K-maps for a BCD-to-Gray code converter.

Design of a SOP circuit to Detect the Decimal numbers 5 through 12 in a 4-bit gray code Input:

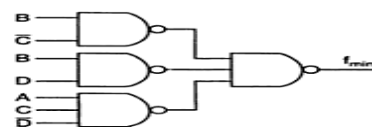
Decimal number	4-bit Gray code				Output f
	A	B	C	D	
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	1	0
3	0	0	1	0	0
4	0	1	1	0	0
5	0	1	1	1	1
6	0	1	0	1	1
7	0	1	0	0	1
8	1	1	0	0	1
9	1	1	0	1	1
10	1	1	1	1	1
11	1	1	1	0	1
12	1	0	1	1	0
13	1	0	1	0	0
14	1	0	0	1	0
15	1	0	0	0	0

(a) Truth table



$$f_{min} = BC + BD + ACD$$

(b) K-map



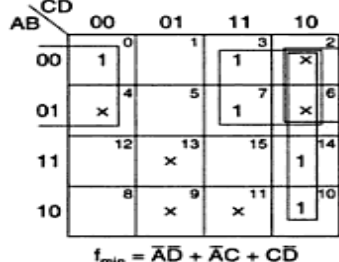
(c) NAND logic

Truth table, K-map and logic diagram for the SOP circuit.

Design of a SOP circuit to detect the decimal numbers 0,2,4,6,8 in a 4-bit 5211 BCDcode input:

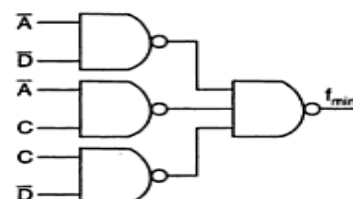
Decimal number	5211 code				Output f
	A	B	C	D	
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	1	1
3	0	1	0	1	0
4	0	1	1	1	1
5	1	0	0	0	0
6	1	0	1	0	1
7	1	1	0	0	0
8	1	1	1	0	1
9	1	1	1	1	0

(a) Truth table



$$f_{min} = ADB + AC + CD$$

(b) K-map



(c) Logic diagram

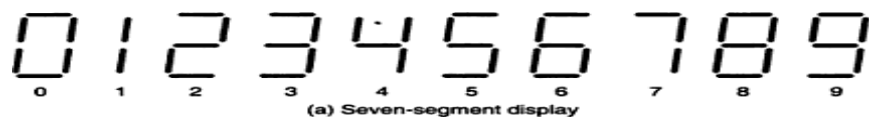
Truth table, K-map and logic diagram for the SOP circuit.

Design of a Combinational circuit to produce the 2's complement of a 4-bit binary number:

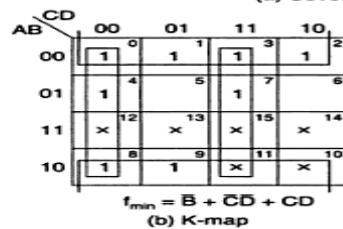
Input				Output			
A	B	C	D	E	F	G	H
0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	1	0	1	1	1	0
0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	0
0	1	0	1	1	0	1	1
0	1	1	0	1	0	1	0
0	1	1	1	1	0	0	1
1	0	0	0	1	0	0	0
1	0	0	1	0	1	1	1
1	0	1	0	0	1	1	0
1	0	1	1	0	1	0	1
1	1	0	0	0	1	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	0	1	0
1	1	1	1	0	0	0	1

(a) Conversion table

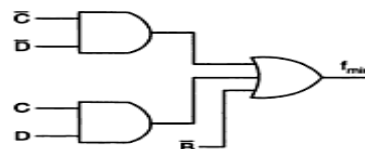
Conversion table and K-maps for the circuit



(a) Seven-segment display



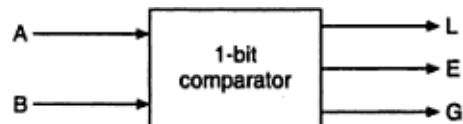
(b) K-map



(c) Logic diagram

Comparators:

$$\text{EQUALITY} = (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)(A_0 \odot B_0)$$



Block diagram of a 1-bit comparator.

The logic for a 1-bit magnitude comparator: Let the 1-bit numbers be $A = A_0$ and $B = B_0$.
If $A_0 = 1$ and $B_0 = 0$, then $A > B$.

Therefore,

$$A > B: G = A_0 \bar{B}_0$$

If $A_0 = 0$ and $B_0 = 1$, then $A < B$.

Therefore,

$$A < B: L = \bar{A}_0 B_0$$

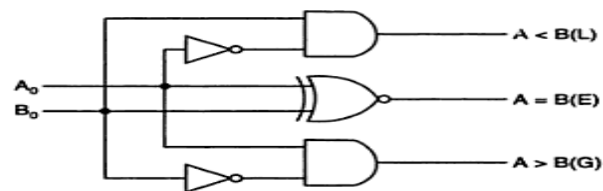
If A_0 and B_0 coincide, i.e. $A_0 = B_0 = 0$ or if $A_0 = B_0 = 1$, then $A = B$.

Therefore,

$$A = B: E = A_0 \odot B_0$$

A_0	B_0	L	E	G
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

(a) Truth table



(b) Logic diagram
1-bit comparator.

1. Magnitude Comparator:

1-bit Magnitude Comparator:

The logic for a 2-bit magnitude comparator: Let the two 2-bit numbers be $A = A_1 A_0$ and $B = B_1 B_0$.

1. If $A_1 = 1$ and $B_1 = 0$, then $A > B$ or

2. If A_1 and B_1 coincide and $A_0 = 1$ and $B_0 = 0$, then $A > B$. So the logic expression for $A > B$ is

$$A > B: G = A_1 \bar{B}_1 + (A_1 \odot B_1) A_0 \bar{B}_0$$

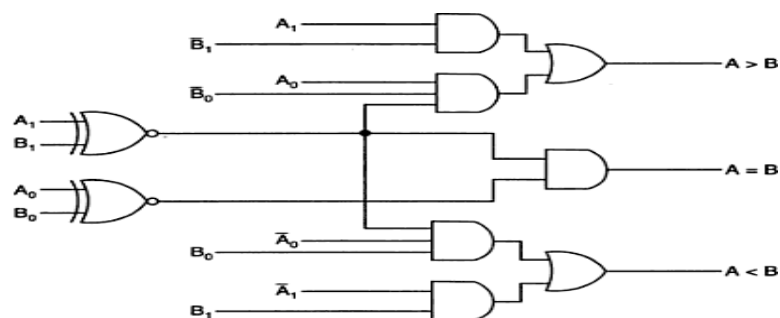
1. If $A_1 = 0$ and $B_1 = 1$, then $A < B$ or

2. If A_1 and B_1 coincide and $A_0 = 0$ and $B_0 = 1$, then $A < B$. So the expression for $A < B$ is

$$A < B: L = \bar{A}_1 B_1 + (A_1 \odot B_1) \bar{A}_0 B_0$$

If A_1 and B_1 coincide and if A_0 and B_0 coincide then $A = B$. So the expression for $A = B$ is

$$A = B: E = (A_1 \odot B_1)(A_0 \odot B_0)$$



Logic diagram of a 2-bit magnitude comparator.

4- Bit MagnitudeComparator:

The logic for a 4-bit magnitude comparator: Let the two 4-bit numbers be $A = A_3A_2A_1A_0$ and $B = B_3B_2B_1B_0$.

1. If $A_3 = 1$ and $B_3 = 0$, then $A > B$. Or
2. If A_3 and B_3 coincide, and if $A_2 = 1$ and $B_2 = 0$, then $A > B$. Or
3. If A_3 and B_3 coincide, and if A_2 and B_2 coincide, and if $A_1 = 1$ and $B_1 = 0$, then $A > B$. Or
4. If A_3 and B_3 coincide, and if A_2 and B_2 coincide, and if A_1 and B_1 coincide, and if $A_0 = 1$ and $B_0 = 0$, then $A > B$.

From these statements, we see that the logic expression for $A > B$ can be written as

$$(A > B) = A_3\bar{B}_3 + (A_3 \odot B_3)A_2\bar{B}_2 + (A_3 \odot B_3)(A_2 \odot B_2)A_1\bar{B}_1 + (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)A_0\bar{B}_0$$

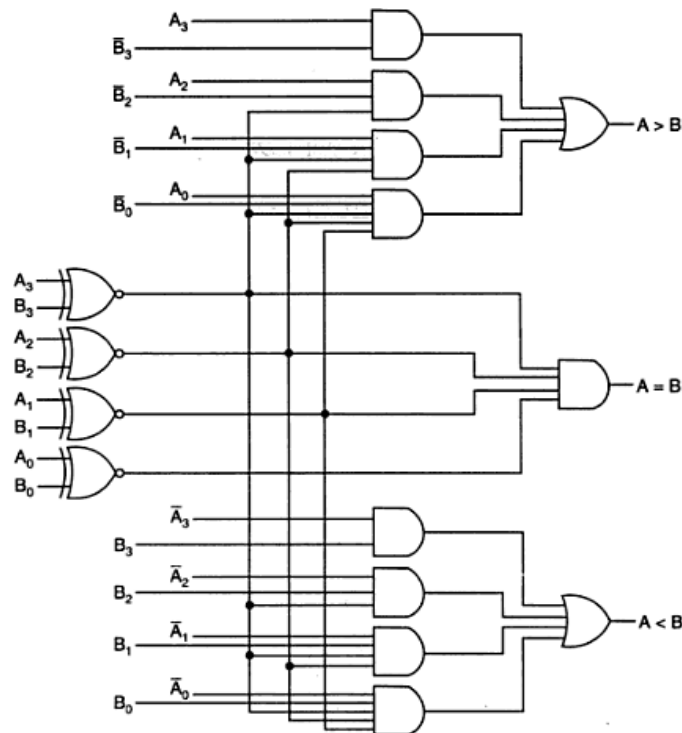
Similarly, the logic expression for $A < B$ can be written as

$$A < B = \bar{A}_3B_3 + (A_3 \odot B_3)\bar{A}_2B_2 + (A_3 \odot B_3)(A_2 \odot B_2)\bar{A}_1B_1 + (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)\bar{A}_0B_0$$

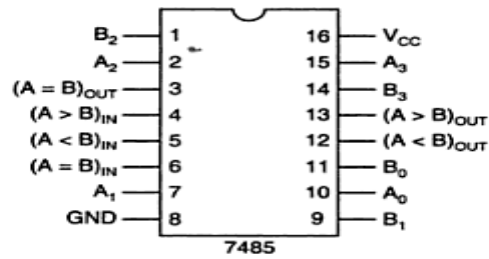
If A_3 and B_3 coincide and if A_2 and B_2 coincide and if A_1 and B_1 coincide and if A_0 and B_0 coincide, then $A = B$.

So the expression for $A = B$ can be written as

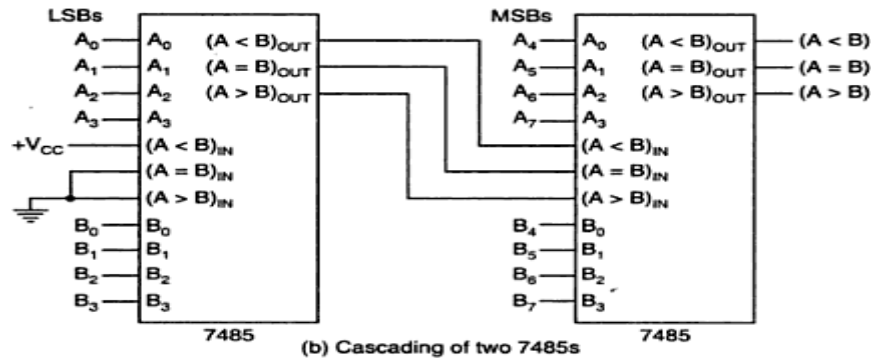
$$(A = B) = (A_3 \odot B_3)(A_2 \odot B_2)(A_1 \odot B_1)(A_0 \odot B_0)$$



IC Comparator:



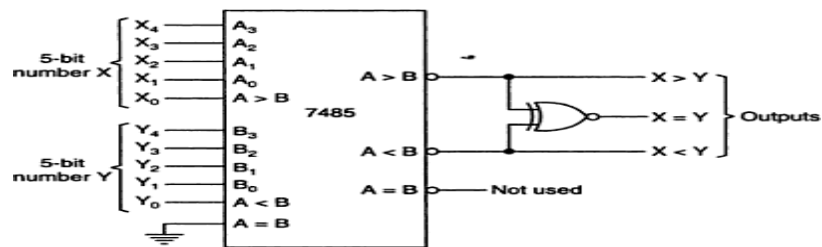
(a) Pin diagram of 7485



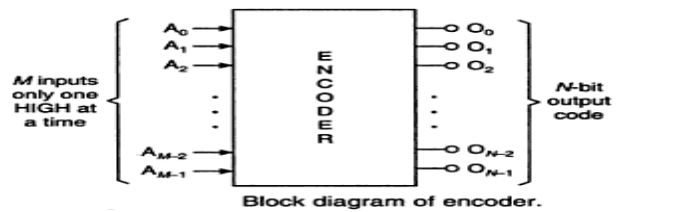
(b) Cascading of two 7485s

Pin diagram and cascading of 7485 4-bit comparators.

ENCODERS:



Use of 7485 as a 5-bit comparator.

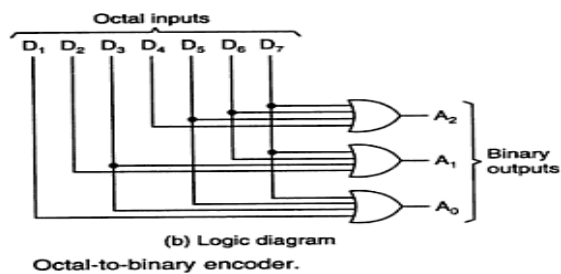


Block diagram of encoder.

Octal to Binary Encoder:

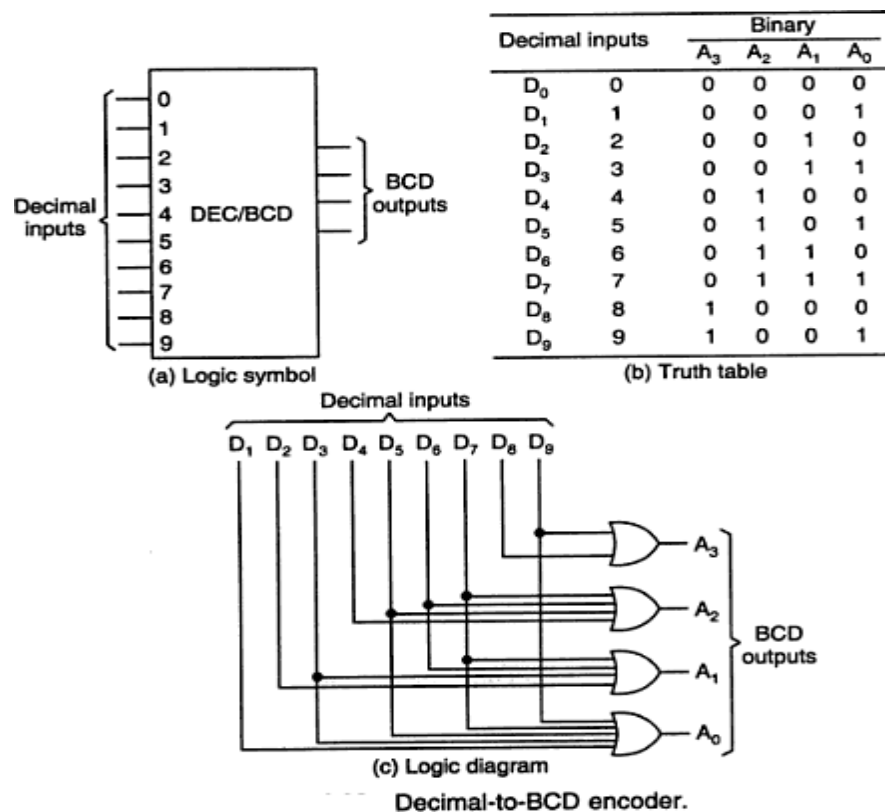
Octal digits	Binary		
	A ₂	A ₁	A ₀
D ₀	0	0	0
D ₁	1	0	0
D ₂	2	0	1
D ₃	3	0	1
D ₄	4	1	0
D ₅	5	1	0
D ₆	6	1	1
D ₇	7	1	1

(a) Truth table



(b) Logic diagram
Octal-to-binary encoder.

Decimal to BCD Encoder:



Tristate bus system:

In digital electronics **three-state**, **tri-state**, or **3-state** logic allows an output port to assume a high impedance state in addition to the 0 and 1 logic levels, effectively removing the output from the circuit.

This allows multiple circuits to share the same output line or lines (such as a bus which cannot listen to more than one device at a time).

Three-state outputs are implemented in many registers, bus drivers, and flip-flops in the 7400 and 4000 series as well as in other types, but also internally in many integrated circuits. Other typical uses are internal and external buses in microprocessors, computer memory, and peripherals. Many devices are controlled by an active-low input called OE (Output Enable) which dictates whether the outputs should be held in a high-impedance state or drive their respective loads (to either 0- or 1-level).



INPUT		OUTPUT
A	B	C
0	1	0
1	1	1
X	0	Z (high impedance)